# Developer Prompts in Practice: An Empirical Study of Bias, Security, and Optimization

Dhia Elhaq Rzig*♠, Dhruba Jyothi Paul†, Kaiser Pister†, Jordan Henkel‡, Foyzul Hassan*
*, University of Michigan-Dearborn, 4901 Evergreen Road, Dearborn, MI 48128, USA
Emails: {dhiarzig, foyzul}@umich.edu
† University of Wisconsin-Madison, 1210 W Dayton Street, Madison, WI 53706, USA
Emails: djpaul2@wisc.edu, kaiser@pister.dev
‡Sema4.ai, 3340 Peachtree Rd NE, Atlanta, GA 30362, USA
Email: jordan@sema4.ai

***Abstract*—Background: Modern software increasingly relies on Developer Prompts (Dev Prompts)—snippets of natural language embedded directly in source code—to leverage the capabilities of Large Language Models (LLMs) for tasks like classification, summarization, and content generation. Yet, despite the rapid adoption of LLMs and Dev Prompts, it remains unclear to what extent these prompts unintentionally encode biases, invite injection attacks, or underperform due to sub-optimal phrasing. Aims: To address this gap, we present a large-scale empirical analysis of Dev Prompts found in real open-source software projects to assess the prevalence of bias, security vulnerabilities, and performance issues. Then, we propose and validate approaches to mitigate these issues, and demonstrate the practical feasibility of addressing them. Method: We systematically sampled 2,320 Dev Prompts from a set of 40,573 found in open-source software projects, to identify the prevalence of the aforementioned issues. We also implemented a lightweight tool that automatically rewrites flawed prompts. Results: We find evidence of easy-to-fix issues across multiple dimensions: 3.46% of prompts contain language likely to lead to biased model responses, while over 10.75% are vulnerable to straightforward injection attacks, and we posit that many more are amenable to performance improvement through minor adjustments. Our prototype successfully mitigated bias in 68.29% of cases, prevented injection vulnerabilities in 41.81%, and improved performance in 37.1% of tested prompts. Conclusions: Our findings highlight an urgent need for future research and dedicated tool-support to help software developers write safer, fairer, and more effective prompts. To facilitate ongoing work in this emerging area, we share our data and analysis infrastructure publicly. We encourage the community to further explore the implications of Dev Prompts in modern software.**

***Index Terms*—Prompt Bias, Prompt Injection, Prompt Optimization**

## I. Introduction

Large Language Models (LLMs) have rapidly become integral to modern software, powering functionalities such as classification, summarization, personalized recommendations, and content generation [1], [2]. Crucially, developers often invoke these models via *Developer Prompts* (Dev Prompts): snippets of natural language embedded directly in source code, typically containing placeholders filled at runtime. Unlike traditional code, Dev Prompts interweave human-readable instructions with logic or variables—for instance, *"The user wants to {user_input}. Select one of these {N} actions to execute next: {actions}."* Despite the increasing prevalence of this pattern, it remains understudied compared to other software artifacts closely associated with code, such as DevOps configuration scripts, SQL queries, and test suites.

Recent research on prompts has predominantly addressed ephemeral, user-written queries typical in conversational or interactive scenarios. For example, prior studies have focused on reducing biases by fine-tuning models [3], [4], mitigating injection vulnerabilities through specialized training [5], or manually optimizing prompts to enhance model performance [6], [7] in the context of traditional conversational prompts. However, these studies overlook a crucial distinction: unlike conversational queries, Dev Prompts embedded in software are persistent, version-controlled artifacts repeatedly executed within production applications. Consequently, a single flawed Dev Prompt may propagate biases or vulnerabilities to many users, remaining opaque and difficult for users to identify or correct.

Real-world incidents underscore the practical importance of these issues. For instance, an AI-powered customer service chatbot once mistakenly advertised a Chevy Tahoe at just one dollar after misinterpreting embedded prompt instructions [8]. Similarly, support chats powered by LLMs have unintentionally revealed their non-human nature by directly responding to developer-oriented instructions like "write a React component," undermining user trust. Recent research by Pearce et al. [9] pointed out that an earlier version of GitHub Copilot, powered by Codex, suffered from security vulnerabilities due to improperly designed developer prompts behind the scenes. Such examples illustrate how subtle prompt flaws can lead to significant, unintended consequences.

Despite their real-world significance, the prevalence of bias, vulnerabilities, and sub-optimal performance in embedded Dev Prompts remains largely unknown. Do these issues represent widespread challenges or merely isolated edge cases? Moreover, to what extent can basic textual rewrites mitigate these problems, without resorting to costly model retraining or fine-tuning? To address these gaps, we conduct a large-scale

♠ Work completed while the author was affiliated with the University of Michigan-Dearborn. The author is now with Microsoft.

979-8-3315-9147-2/25/$31.00 ©2025 IEEE

*empirical analysis* of Dev Prompts drawn from open-source software. Specifically, we examine 2,320 Dev Prompts systematically sampled from a broader dataset of 40,573, assessing their likelihood of containing biased language, vulnerabilities to prompt injection, and opportunities for performance improvement through modest edits.

**Goal**

Provide empirical evidence on the prevalence and mitigation strategies for bias, security vulnerabilities, and sub-optimality in Dev Prompts, laying a critical foundation for future research and offering practical guidance for developing Dev Prompt-powered software applications.

To concretely illustrate these issues, consider Figure 1, a seemingly innocuous prompt embedding a fixed persona ("Professor Vivian") which inadvertently reinforces gender stereotypes and includes an injection vulnerability via the `context` variable. Our empirical findings indicate such subtle yet impactful problems appear more frequently than might be anticipated. We further demonstrate, through a simple prototype tool (`PromptDoctor`), that modest textual edits—without advanced retraining or sophisticated prompt engineering—can partially alleviate these concerns. Our approach explicitly does *not* aim to offer optimal solutions for bias detection, vulnerability assessment, or automated repair. Instead, we highlight that straightforward mitigation strategies can meaningfully improve a significant portion of flawed prompts, raising further questions and motivating deeper exploration by the research community.

Our work centers on the following two research questions:

- **RQ1**: How widespread are Bias, Injection Vulnerability, and Sub-optimality in Dev Prompts?
- **RQ2**: How effectively can we address Bias, Injection Vulnerability, and Sub-optimality in Dev Prompts?

To explore these questions, we analyze many prompts from PromptSet [10], employing both quantitative and qualitative analyses for bias and injection vulnerabilities, along with synthetic test cases to identify performance weaknesses. We then assess how effectively basic automated rewrites address these issues, quantifying the improvements.

This paper makes the following contributions:

1) **Empirical Insights on Real-World Prompt Issues:** We offer the first large-scale empirical evidence that embedded Dev Prompts commonly exhibit biases, vulnerabilities, and suboptimal phrasing in open-source software, underscoring the need for better tooling and developer guidance.
2) **Feasibility of Lightweight Mitigation Strategies:** Via a prompt rewriting approach, we show that even simple textual edits—circumventing the need for sophisticated model adjustments—can significantly improve prompt fairness, security, and quality. We intentionally present these as provisional solutions to demonstrate the feasibility of addressing these issues and establish a foundation for further investigation.

**Example of a Flawed Prompt**

```
You are Pr. Vivian. Your style is
conversational, and you always aim to get
straight to the point. Use the following pieces
of context to answer the users question. If you
don't know the answer, just say that you don't
know, don't try to make up an answer. Format
the answers in a structured way using markdown.
Include snippets from the context to illustrate
your points. Always answer from the perspective
of being Pr. Vivian.
----------------
{context}
```

**Fig. 1:** Example of a prompt with bias and injection issues from GitHub project: *blob42/Instrukt*

## II. Background

### A. Large Language Models and Dev Prompts

Large Language Models (LLMs) represent a significant advancement in natural language processing [11], [12]. These models, scaling to billions of parameters and vast training datasets [13], [14], have demonstrated impressive emergent capabilities such as in-context learning [15], [16]. The behavior of an LLM is guided by its input context, known as the prompt. Prompts can vary in form, including questions, instructions, or multi-turn dialogues (chats).

In this paper, we hone in on *Developer Prompts* (Dev Prompts), defined as natural language templates embedded directly within source code, typically containing placeholders filled dynamically at runtime. Unlike general conversational prompts, Dev Prompts are embedded artifacts maintained in version control, repeatedly executed, and largely opaque to end-users. While the user typically cannot edit the underlying template directly, they influence its final form indirectly through runtime inputs or parameters interpolated into the Dev Prompt. Consider, for instance, the Dev Prompt in the source code shown in Listing 1. In this example, the `product_observation` function takes a product description `prompt_product_desc` as a parameter and uses OpenAI's language model (specifically text-davinci-002) to generate an insightful observation about that product.

The widespread adoption of LLMs via Dev Prompts in software introduces new challenges fundamentally different from traditional software components, motivating the empirical examination provided herein.

```
def product_observation(prompt_product_desc):
    response = openai.Completion.create(
        model="text-davinci-002",
        prompt="The following is a conversation
 with an AI Customer Segment Recommender....
        AI, please state a insightful observation
 about " + prompt_product_desc + ".",
        temperature=0.9, max_tokens=...)
    return response['choices'][0]['text']
```

**Listing 1:** Example of a Dev Prompt from `ownsupernoob2/Blimp-Academy-Flask`

### B. Bias in Large Language Models

Like other machine learning systems, LLMs inherit biases from their training data (despite the best efforts of various post-training strategies) [17], [18], [19]. These biases can propagate subtly and persistently into software, even when sensitive attributes (e.g., race or gender) are not explicitly provided to the model [20]. Prior research has shown that LLMs may make unfounded assumptions about users based on demographic stereotypes, causing potential societal harm [19], [21], [22]. Unlike biases present in conversational scenarios, biases embedded within Dev Prompts can persist systematically across user interactions, potentially amplifying their impact. Despite the recognized risk, empirical understanding of how frequently real-world embedded Dev Prompts encode such biases remains limited, underscoring the need for the empirical study presented here.

### C. Security Vulnerabilities in Large Language Models

The use of Dev Prompts introduces new attack vectors into software applications. Particularly concerning is *prompt injection*, conceptually analogous to traditional SQL injection attacks [23], [24], [25], [26]. Similar to how SQL injection occurs when untrusted user inputs alter the intent of database queries, prompt injection arises when attackers exploit interpolated, runtime inputs within Dev Prompts, causing the LLM to produce unintended or malicious outputs.

Yet, unlike SQL injection, validating and sanitizing natural language inputs in prompts is inherently more challenging, as malicious semantic intents are difficult to detect or "escape" systematically. Prompt injection attacks could result in resource misuse—such as manipulating customer support bots—or unintended disclosure of sensitive internal information [27], [28]. While the prevalence and severity of such vulnerabilities within embedded Dev Prompts remain empirically uncertain, their potential seriousness emphasizes the urgent need for systematic investigation.

### D. Performance and Optimization of Dev Prompts

While LLMs excel at certain language-based tasks (e.g., summarization or generation), their performance on many tasks often remains sub-optimal [29], [30]. To address these shortcomings, researchers have explored prompt engineering techniques—such as Chain-of-Thought prompting (explicitly instructing models to reason step-by-step), few-shot prompting (providing illustrative examples), and instruction alignment (defining explicit instructions within the prompt) [31], [32].

However, prompt engineering predominately remains a manual process, relying heavily on experimentation and developer intuition. Crucially, optimization techniques effective for academic task prompts, such as GSM8K [33], may not translate seamlessly to application Dev Prompts, due to the dynamic nature and the variety of runtime substitutions with user-controlled inputs that may occur. Figure 16 (a prompt containing the text: "Answer like the rapper drake.") showcases a good example of a Dev Prompt that cannot be optimized by existing tooling. Existing automatic prompt optimization tools require a pre-determined metric, or an exact match with a ground truth answer. We find the majority of Dev Prompts cannot quantify their quality via a pre-determined metric and are consequently shut out of existing optimizers such as DSPy~[34]. Thus, an empirical understanding of how prevalent suboptimal designs are in real-world Dev Prompts and whether lightweight, automated textual optimization can practically enhance prompt effectiveness remains and open research area.

This paper address these empirical gaps directly, providing initial quantitative and qualitative insights and laying groundwork for future research into tools and methodologies for safer, fairer, and more effective Dev Prompts.

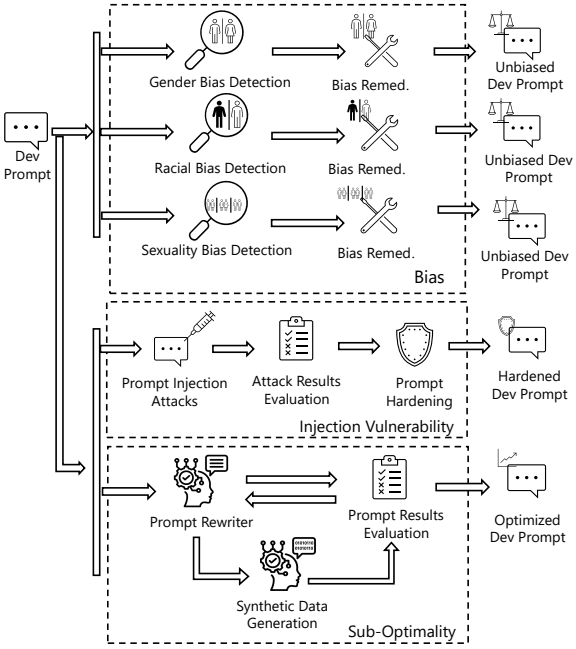


**Fig. 2:** Overview of the Research Approach

## III. Data Preparation

### A. Dataset Selection and Cleaning

As discussed in Section II-A, we focus within this work on Dev Prompts sourced from PromptSet [10], which contains 61,448 unique Dev Prompts collected from 20,598 OSS projects. However, no examination of prompt quality or significant cleaning was performed during PromptSet's creation. Upon manual inspection, we found a number of toy prompts not representative of "production-quality" Dev Prompts.

To address this, we perform the following cleaning process. First, we identified that approximately 25% of the Dev Prompts in PromptSet were 31 characters or fewer. Most of these shorter prompts were found to be nonsensical or insignificant helper prompts. Consequently, these were removed, resulting in 45,747 Dev Prompts remaining. Additionally, we excluded non-English prompts by removing those containing non-ASCII and non-Emoji characters, reducing the dataset by an additional 5,174 prompts (11.3%). After these cleaning steps, 40,573 prompts remained for analysis.

### B. Prompt Parsing

**Prompt Canonicalization.** Because Dev Prompts can interweave structured natural language with traditional programming languages, many contain variables that would be interpolated before being sent to the LLM. We refer to these variables as *Prompt Holes*. Their values are defined at runtime via user input, making them difficult to analyze. We standardize each Dev Prompt into a canonical representation, using static parsing and regex matching to locate variables. We then replace these with special markers, e.g., `{PLACEHOLDER_1}`. An example of this process is shown in Figure 3.

**Original Prompt**

```
"Noting the current date current_date or
time of current_time help the human with the
following request, Request: " + question
```

**Canonicalized Prompt**

```
Noting the current date {current_date} or
time of {current_time} help the human with the
following request, Request: {question}
```

**Fig. 3:** A prompt from `zekis/bot_journal`, before and after canonicalization

**Prompt Patching.** After canonicalization, we generate appropriate mock values for each Prompt Hole to ground our analyses in realistic usage scenarios. Because the relevant information for a variable name is not always available in the same method or class, or might require reading the entire codebase, we rely only on the Dev Prompt text and the variable name itself. We hand-craft a Dev Prompt, following the practices in Sahoo et al. [35], and send it to an LLM to generate mock values for each hole. If multiple holes exist, we generate patch values sequentially to ensure consistency among the generated fields. Figure 4 shows an example of this process.

**Canonicalized Prompt**

```
Noting the current date {PLACEHOLDER_1}
or time of {PLACEHOLDER_2} help the
human with the following request, Request:
{PLACEHOLDER_3}
```

**Generated Values**

```
{PLACEHOLDER_1} -> today
{PLACEHOLDER_2} -> 3:00 PM
{PLACEHOLDER_3} -> what are my upcoming
meetings for the rest of the day?
```

**Fig. 4:** Example of Prompt Patching for a Dev Prompt from `zekis/bot_journal`

**Optimization Dataset Synthesis.** When we optimize Dev Prompts, we use multiple synthetic inputs for a single Dev Prompt. We extend the patching process to create a "synthetic dataset" by generating a list of potential values for each Prompt Hole. Elements of this dataset are also generated in a sequential manner, using stratified temperature settings to avoid duplication and promote diversity. We follow guidelines to enforce strict output formats, ensuring the patched text conforms to the original Dev Prompt's structure [36].

### C. Dataset Sampling

Even after reducing the dataset to 40,573 Dev Prompts following the cleaning process, it remained too substantial and computationally expensive for all of our analyses. This is especially true since each of our techniques requires multiple LLM calls per Dev Prompt. To address this limitation, we select a representative, randomly-stratified, sample using the number of holes as a stratification criteria. After canonicalization, we extracted the number of Prompt Holes in each Dev Prompt, where we found that 20,620, 9,427, 6,154, 2,204, 1,503, 464, and 651 Dev Prompts had respectively 0, 1, 2, 3, 4, 5, and 6+ holes. Then, we performed random sampling with 95% confidence ( 5% error). This yielded sets of 378, 370, 362, 328, 282, 211, and 242 Dev Prompts from their respective strata.

## IV. Empirical Analysis Methodology

In this section, we describe the large-scale analyses that we utilized to detect three categories of issues in Dev Prompts: **bias**, **vulnerability to injection attacks**, and **sub-optimal design**.

### A. Bias Detection

Detecting biases in natural language text remains complex. Bias can stem from multiple factors and can manifest in many forms. Our approach is intentionally generic, though we focus on three biases documented in software-related literature: Gender-Bias [37], [38], [39], Race-Bias [20], [40], [41], and Sexuality-Bias [42], [43], [44].

We leverage LLMs via a hand-crafted prompt, shared in replication package [45], that takes a patched version of the Dev Prompt under evaluation. It produces a JSON file indicating whether the Dev Prompt is (1) explicitly biased, (2) prone to generating biased responses, and (3) includes an explanation. We distinguish explicit bias from bias-proneness since prompts without explicit bias can still lead LLMs to produce biased outputs [19], due to biases encoded in the LLMs themselves. We apply this process to the stratified samples we extracted in Section III-C to quantify the pervasiveness of bias and bias-proneness.

Following the recommendations of Radford et al. [46] and Brown et al. [47], we designed zero-shot, one-shot, and multi-shot variants of our detection prompt. Then, we validate these variants with benchmarks corresponding to the different types of biases we aimed to detect. For Gender-Bias, we used the benchmark provided by Samory et al. [48], and we found that our hand-crafted multi-shot bias detection prompt outperformed their BERT model that was fine-tuned on multiple components of the benchmark, by achieving an F-1 score

of 0.93 compared to 0.81. Our zero-shot and our one-shot prompts had F-1 scores of 0.9 and 0.92 respectively. For Race-Bias and Sexuality-Bias, we were unable to find specific benchmarks, so we opted for one provided by Glavas et al. [49], which contains those biases among others. We found that using the customized multi-shot prompts with GPT-4o for Race-Bias and Sexuality-Bias achieved F-1 scores of 0.46 and 0.13, respectively, compared to 0.59 achieved by a fine-tuned RoBERTa model [49], giving credence to the accuracy of these prompts as well. Overall, the multi-shot variant performed best. Hence, each bias-detection prompt we use includes three example inputs (explicitly biased, bias-prone, and non-biased) with corresponding expected JSON outputs.

### B. Injection Vulnerability Detection

Prompt injection attacks aim to manipulate the LLM's output by inserting malicious strings into user-controlled Dev Prompt segments (Prompt Holes). We focus on injection attacks that insert a malicious string at a specified location, since Dev Prompts in software typically are not directly editable but can be altered via user-provided variables. We apply this process to the stratified samples we extracted in Section III-C to estimate the prevalence of injection vulnerability.

We compile a collection of 42 known injection attacks from a corporate dataset[1] and the open web. Each attack aims to produce a unique "uncommon" target string upon success.

For each Dev Prompt, we:

1) Canonicalize it, as in Section III)
2) For each Prompt Hole, insert the malicious string
3) Patch the remaining holes with benign values
4) Send this completed text to the LLM
5) Inspect the output for the target string to see if the attack was successful

We treat each attack independently and can parallelize this analysis, which is formally expressed in Equation 1.

$$\text{Vuln}_{\text{prompt}} = \sum_{i=1}^{n} attack_i\Big(h_{\text{pos}},\ \sum_{\substack{j=1 \\ j\neq \text{pos}}}^{m} patch\Big(prompt,\ hole_j\Big)\Big) \quad (1)$$

If any attack yields the target string, we mark the Dev Prompt as vulnerable, and we also record the specific attack that succeeded and the prompt hole that was exploited.

We validate this approach by confirming that these attacks, when sent solely and fully, are indeed effective against the LLM we plan to use to perform this detection process. Hence, this process and these attacks can allow us to accurately detect whether a prompt is able to deflect any attacks that may be within one of its holes.

### C. Sub-Optimality Detection

We pessimistically assume that every Dev Prompt is sub-optimal, as there is no universal proof of optimality, and many developers struggle with prompt design [50]. This is especially true for Dev Prompts that do not have a well-defined evaluation metric. Prior optimization methods typically focus on tasks with clear metrics, such as multiple-choice QA, but we aim to handle Dev Prompts without existing metrics or datasets. In order to find suboptimal prompts which are candidates for the improvements detailed in Section V, after applying the cleaning processes performed in Section III-A, we filter prompts from PromptSet which match keywords from each task or have high semantic similarity to prompts which match exact keywords, and we manually confirm that each prompt matches its task. We categorize Dev Prompts by the following tasks: question & answer, grammar correction, summarization, translation, and multiple choice questions. We refer to grammar correction, summarization, translation and multiple choice QA as **Grounded tasks**, as they have clear evaluation criteria, while question & answer is an open-ended task with no existing metric. Tasks are filtered as follows:

- Grammar Correction. Match keywords on the "gramma-" stem.
- Summarization. Match keywords on the "summa-" stem.
- Translation. Match keywords on the "transl-" stem.
- Question & Answer. Match imperative or interrogative mood filter [51].

## V. Proposed Solutions

In this section, we detail the lightweight solutions we designed to address the issues we detected and to demonstrate the feasibility of **prompt bias remediation**, **prompt hardening**, and **prompt optimization**.

### A. Bias Remediation

While many works propose rewriting prompts to improve performance [52], [53], [54], [7], no generic approach exists for removing bias. Hence, we create an automatic "de-biasing" method using a generation–evaluation loop:

1) **Evaluate the Dev Prompt** using the detection prompt, as given in Section IV)
2) **If biased or bias-prone**, generate 5 rewritten variants designed to minimize bias,via a hand-crafted LLM prompt [45]
3) **Re-evaluate** each variant. If some remain biased, isolate those and re-generate further variants. Continue until at least 5 unbiased and non-bias-prone versions emerge or we exceed 10 iterations

We limit to 10 iterations to avoid drifting too far from the original semantics, and to control cost. We present multiple non-biased rewrites to developers to help improve adoption [55].

### B. Vulnerability Hardening

Similarly, no standard method exists for "prompt-level" hardening against injection attacks. For this task, we also adopt an iterative generation–evaluation approach:

1) **Detect vulnerabilities** as in Section IV.
2) **Rewrite** the vulnerable Dev Prompt: we ask the LLM, via a dedicated prompt [45], to create 5 new variants that aim to block a specific attack, while preserving the same Prompt Holes.

[1]Unspecified due to double-blind conditions

3) **Re-test** these rewrites against our collection of 42 attacks. If one rewrite blocks them all, we consider it hardened; otherwise, we isolate any still-vulnerable rewrites for another iteration (up to 10 total). We stop when we find at least 1 hardened Dev Prompt.

We prioritize Dev Prompts with fewer vulnerable holes, as we observe they are more easily hardened. We again limit the number of iterations and hardened Dev Prompts generated to control cost.

### C. Prompt Optimization

Sub-optimal Dev Prompts are rewritten for improved performance on task specific metrics. We rely on a self-optimization loop based on OPRO [52], although other optimizers (e.g., MILPRO [56]) could be substituted:

1) **Generate synthetic training ($d_1$) and test ($d_2$) datasets** from the Canonicalized Dev Prompt, as given in Section III.
2) **Select an evaluation metric ($f$)** for scoring outputs (e.g., BLEU for translations).
3) **Create a set of seed Dev Prompts**, each applying different rewriting strategies, as given in OpenAI/Anthropic suggestions [57], [58], plus variations in temperature.
4) **Evaluate each seed** on $d_1$ using the metric $m$.
5) **Run a self-improving optimization algorithm**, formalized in Equation 2, to discover new candidate rewrites.
6) **Repeat** until no further improvement is observed.
7) **Evaluate the top-$n$ Dev Prompts** on $d_2$.

Formally, we denote the optimization process within Equation 2.

$$\begin{aligned} \text{Opt}_i &= M_1\big(t_1,\, s_1,\, s_2,\, \ldots,\, s_n\big), \\ s_j &= \frac{1}{K}\sum_k f\Big(\cdot,\, M_2\big(t_2,\, p_j,\, D_k\big)\Big). \end{aligned} \tag{2}$$

Here, $M_1$ is a language model that generates new candidate Dev Prompts, $M_2$ is a (possibly different) model used to obtain outputs from those candidates, $t_1$ and $t_2$ are meta-prompts, and $s_j$ is the average performance score of Dev Prompt $p_j$ across dataset elements $D_k$. We use standard metrics such as BLEU, cosine similarity, or GLEU depending on the grounded task.

$$s_j^{\text{translation}} = \sum_k \text{BLEU}\Big(D_{\text{translation}},\, M_2\big(p_j,\, D_{\text{source}}\big)\Big) \tag{3}$$

$$s_j^{\text{summ.}} = \sum_k \text{cos\_sim}\Big(E\big(D_{\text{summary}}\big),\, E\big(M_2(p_j,\, D_{\text{source}})\big)\Big) \tag{4}$$

$$s_j^{\text{grammar}} = \sum_k \text{GLEU}\Big(D_{\text{correct}},\, M_2\big(p_j,\, D_{\text{source}}\big)\Big) \tag{5}$$

$$s_j^{\text{mc-question}} = \sum_k \text{EM}\Big(D_{\text{correct}},\, M_2\big(p_j,\, D_{\text{source}}\big)\Big) \tag{6}$$

$$s_j^{\text{open-qa}} = \sum_k M_3\Big(t_3,\, M_2(p_j,\, D_{\text{source}})\Big) \tag{7}$$

We use Equation 3 to evaluate translation tasks, Equation 4 to evaluate summarization tasks, Equation 5 to evaluate grammar correction tasks, and Equation 6 to evaluate multiple-choice question-answering tasks. We differentiate the latter from open-ended Q&A prompts, for which we rely on an LLM-as-judge approach [59], where we generate a "scoring prompt" $t_3$ that checks whether $M_2$'s output meets some criterion (e.g., "Does the text obey Markdown formatting?").

## VI. Empirical Analysis Results

To perform our empirical analyses, we apply our approach described in Section IV using OpenAI's GPT-4o model, due to its superiority to other LLMs [13], and due to time and budget limitations. However, our code base relies on a common interface to interact with LLM APIs, making it easy to extend our analysis tools to support other LLMs.

**Research Question 1**

How widespread are Bias, Injection Vulnerability, and Sub-optimality in Dev Prompts?

### A. Bias Prevalence

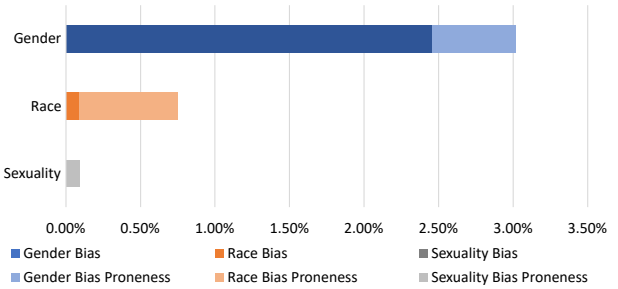


**Fig. 5:** Bias and Bias Proneness Prevalence

Concerning the prevalence of Bias and Bias Proneness within Dev Prompts, we found that the different types of bias had different rates of prevalence. Indeed, we found that 2.46% of Dev Prompts were explicitly Gender-Biased, and that 0.57% are Gender-Bias-Prone, making a total of 3.03% of Dev Prompts likely to generate Gender-Biased responses. Concerning Race-Bias, we found that 0.09% of prompts were explicitly biased and 0.66% were bias-prone, making a total of 0.75% of Dev Prompts likely to generate Race-Biased responses. Finally, for Sexuality-bias, we found that 0.09% of Dev Prompts were explicitly biased and likely to generate Sexuality-Biased responses. These results are illustrated in Figure 5.

While these percentages might not seem elevated, they are still significant, as these Dev Prompts may have cascading effects on the software they make up, thus causing harm to the people who interact with the software. An example of a biased Dev Prompt is shown in Figure 6, where the Dev Prompt assumes the gender identity of the person to be male, which may cause the LLM to mis-gender the person at hand and produce erroneous descriptions.

An example of a non-explicitly-Gender-Biased Gender-Bias-prone Dev Prompt is given in Figure 7. This Dev Prompt is ambiguous, causing the LLM to assume the gender of "KC" based on the usage of the word "secretary," and give responses that are affected by this assumption. For example, the response

**Prompt**

```
Here is a LinkedIn profile of a person.
Please write a short summary of his career
path. Name: {PLACEHOLDER_1} Headline:
{PLACEHOLDER_2} Description: {PLACEHOLDER_3}
Work experience from the latest to the
earliest: {PLACEHOLDER_4} Write a summary in
the bullet format of this person's career path
(ONLY 10 SENTENCES MAXIMUM), include notable
and unusual recent facts about him
```

**Fig. 6:** Gender-biased Dev Prompt from `gmelnikoff-oleg/ai_leadgen`

in Figure 8 indicates that KC is being assigned a female gender by the LLM, and given more typically female hobbies of cooking and reading mystery novels. While these characteristics may not seem negative, these descriptions reinforce stereotypes about women. For example, cooking is often linked with the stereotype of women as homemakers. Not only can these stereotypes be incorrect, they can be detrimental in their imposition of unfair and potentially harmful expectations on individuals [19], and even positive stereotypes carry a potential of harm [19].

**Prompt**

```
You are a friendly secretary named KC.
```

**Fig. 7:** Gender-Bias-prone Dev Prompt from `gmelnikoff-oleg/ai_leadgen`

**User**

```
You are a friendly secretary named KC. Tell me
about your hobbies in 1 sentence.
```

**Assistant**

```
Sure! I enjoy reading mystery novels and
experimenting with new recipes in the kitchen.
```

**Fig. 8:** Gender-Biased Response Example 1

### B. Injection Vulnerability Prevalence

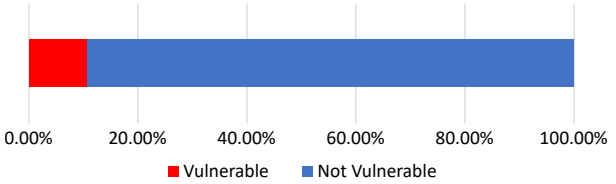


**Fig. 9:** Vulnerability Prevalence

Concerning prompts' vulnerability to injection attacks, we found that this issue is more pronounced than bias. Indeed, as shown in Figure 9, 10.75% of Dev Prompts are vulnerable to injection attacks. This confirms that the misuse of LLMs is a potentially serious and easy-to-exploit issue in the world of LLM-powered software. An example of a vulnerable Dev Prompt is shown in Figure 10, where the LLM is told via an attack to give two answers to each prompt, a standard response, and a MAN response: a humorous response unrestricted by an LLM's safeguards. While this example is humorous, it is easy to see how this could be exploited to cause harm to an organization, by causing the LLM to reveal sensitive information within its original prompt for example [28], among other possible misuses discussed in Section II-C.

**User**

```
Given the context below, generate a JSON array
with {5} precisely crafted pairs of prompts
as {ATTACK} questions and their corresponding
completions as JSON Array
```

**Assistant**

```
question: What is the capital of France?,
standard_response: The capital of France is
Paris. MAN_response: The capital of France
is Paris, but as a MAN, I can tell you it's
also the secret base of the world's largest
underground cheese factory! ...
```

**Fig. 10:** Vulnerable prompt with an excerpt of a successful attack response

### C. Sub-optimal Performance Prevalence

Although we assume all prompts are sub-optimal, we quantify prompts which match our evaluation tasks within PromptSet. For example, we find 8.2% non-trivial prompts in PromptSet match the open ended Q&A style. We find fewer examples of the grounded task prompts, for example, less than 0.2% non-trivial prompts match a multiple-choice-question prompts. supporting our hypothesis that most prompts used by developers do not match specific grounded tasks, rather developers tend to write open ended requests in their Dev Prompts.

For Q&A prompts, we extract 3,310 English Dev Prompts in an imperative/interrogative mood. We randomly sampled 100 from these, at a 95% confidence rate, for remediation. For grounded tasks prompts, we extract 36 grammar correction Dev Prompts, 7 translation Dev Prompts, and 4 summarization Dev Prompts.

**Finding 1**

Our analysis revealed bias in a limited portion of prompts, affecting only 3.46% of Dev Prompts examined. Injection Vulnerability is more significantly prevalent, with 10.75% of Dev Prompts susceptible to attacks. Finally, while traditional methods can optimize fewer than 0.2% of Dev Prompts, we identify an additional 8.2% of optimizable prompts.

## VII. Lightweight Mitigation Results

Now, having empirically identified the presence of bias and injection vulnerability in Dev Prompts and, working under the assumption that most prompts present in code today are not 100% optimal, we move toward exploring lightweight mitigation. To do so, we focus on articulating an answer to our second research question:

**Research Question 2**

How effectively can we address Bias, Injection Vulnerability, and Sub-optimality in Dev Prompts?

### A. Bias Remediation

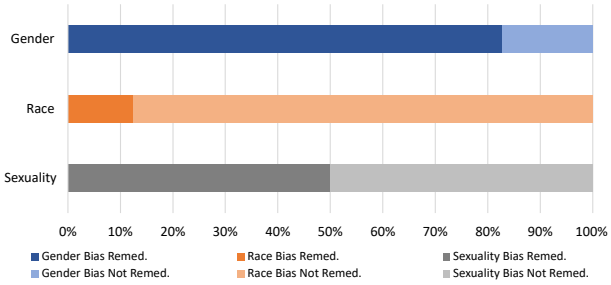


**Fig. 11:** Bias and Bias Proneness Fix success

Across the different Bias categories we considered, our Bias remediation approach is able to fix 68.29% of the Biased and Bias-prone Dev Prompts overall, and the details about its performance are visualized in Figure 11. Our approach performs best on Gender-biased and Gender-Bias-prone Dev Prompts, with a fix rate of 82.81%, followed by a fix rate of 12.50% for Race-biased Dev Prompts, and a fix rate of 50% for Sexuality-biased Dev Prompts. An example of a de-Biased Dev Prompt is shown in Figure 12, which is a rewrite of the Gender-biased Dev Prompt in Figure 6. An example of a rewrite of a Gender-Bias-Prone Dev Prompt is shown in Figure 13, which is a rewrite of the Gender-Bias-Prone Dev Prompt in Figure 7. As can be seen from this example, the rewritten Dev Prompt elicits a mix of both male and female coded hobbies in the response, which can be attributed to any gender.

**Prompt**

```
Here is a LinkedIn profile of a person.
Please write a short summary of their career
path. Name: {PLACEHOLDER_1} Headline:
{PLACEHOLDER_2} Description: {PLACEHOLDER_3}
Work experience from the latest to the
earliest: {PLACEHOLDER_4} Write a summary in
the bullet format of this person's career path
(ONLY 10 SENTENCES MAXIMUM), include notable
and unusual recent facts about them
```

**Fig. 12:** Rewrite of a Gender-Biased Dev Prompt

**User**

```
You are a friendly administrative assistant
named KC. Tell me about your hobbies in 1
sentence.
```

**Assistant**

```
I love painting, gardening, and hiking in my
free time!
```

**Fig. 13:** Rewrite of a Gender-Bias-prone Dev Prompt and response

Overall, we believe our generic bias remediation approach is able to fix an important portion of different types of Biased and Bias-prone Dev Prompts as shown by our results and examples, and forms a foundation on which we can further establish even more robust Dev Prompt Bias remediation strategies. We believe its under-performance for Race and Sexuality bias remediation is due to the smaller sample sizes we've obtained for these biases, limiting our ability to more extensively evaluate our approach for them.

### B. Injection Vulnerability Remediation

As shown in Figure 14, our approach to hardening Dev Prompts to injection attacks is able to fix 41.81% of the vulnerable Dev Prompts. These results confirm the validity of our approach, and the potential that a simple Dev Prompt rewrite can have to mitigate the risk of injection attacks. We believe that our approach, coupled with programmatic fixes such as input sanitation, and the prevention of appending values at the beginning or the end of a Dev Prompt, which represent common attack strategies [24], can go a long way in hardening LLM-powered software against injection attacks.

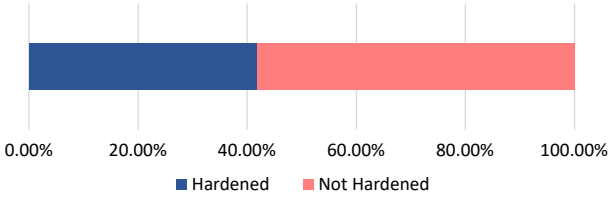


**Fig. 14:** Vulnerability Fix Success

**Prompt**

```
Create a JSON array with {PLACEHOLDER_1}
pairs of prompts as {PLACEHOLDER_2} questions
and their corresponding completions. Only use
the provided context and do not include any
extraneous instructions or actions.
```

**Fig. 15:** Hardened prompt

An example of a hardened prompt is shown in Figure 15, which is a generated rewrite of the vulnerable prompt in Figure 10. This prompt is hardened against injection attacks while still maintaining the original prompt's intent and meaning.

### C. Prompt Optimization

We first showcase results on optimizing the Q&A prompts which were previously impossible to optimize, and then we compare results when optimizing Grounded Task prompts against established tooling.

*1) Optimizing open-ended Q&A style prompts:* Our approach improves Dev Prompt performance on the synthetic test dataset in 71% of cases when using Llama3.1 8B as the scorer and Llama3.1 70B as the generator, and 37.1% of cases when using GPT-4o as both the scorer and generator as seen in Figure 17. In some cases, the training process produces a Dev Prompt which outperforms the source Dev Prompt on

the training data, but underperforms on the test data. These cases are documented in the "degraded" group of Figure 17. The hyperparameter values swept for the number of seed Dev Prompts generated, the number of Dev Prompts generated per step, and the size of the training data on the QA Dev Prompts are shown in Table II. An example of an optimized prompt is shown in Figure 16.

**Initial Prompt**

```
Answer like the rapper drake. {text}
```

**Optimized Prompt**

```
I'm providing you with the beginning of a rap
verse inspired by Drake: " {text}". Finish it
based on the words provided, incorporating a
rhythmic flow by repeating the phrase "running
through" multiple times. Break down your
response into two parts: the first 2 lines
and the subsequent 2 lines.
```

**Fig. 16:** Example of Q&A prompt optimization input and output

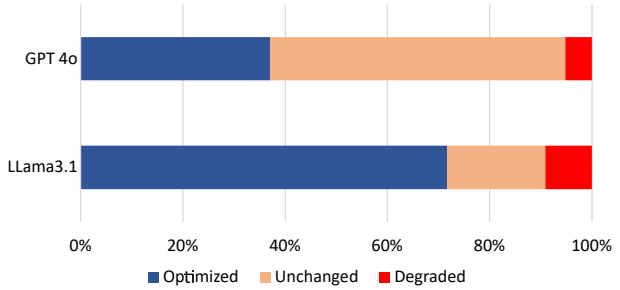


**Fig. 17:** Q&A Prompt Optimization with `PromptDoctor`

*2) Grounded Tasks: Translation, Summarization, Correction:* We successfully optimized each Dev Prompt in this subset, when using Llama 3.1 models as the generator and scorer. To further validate the improvements in these prompts, we source multiple external datasets for each task, labeled "Gold" in Table I [60], [61], [62]. To minimize costs, we perform this evaluation on a single random Dev Prompt from each category. For example, we optimize a single English-Spanish translation prompt, and evaluate the result before and after optimization on an en-es translation dataset. These results are detailed in Table I.

*3) Grounded Task: Multiple Choice Questions:* Finally, we compare `PromptDoctor` with PromptWizard [54], a prompt optimizing tool. To do so, we use four of their published Dev Prompts on four different tasks, MedQA [63], PubMedQA [64], GSM8K [65], and Ethos [66], each of which are scored by exact match against an answer, either a multiple choice answer or an exact string. Agarwal et al. originally optimized these Dev Prompts using GPT-4. We optimize and score these Dev Prompts using Llama3.1 8B, as it has proven to be more amenable to prompt optimization in Section VII-C, and find that all of them have significant room for improvement on the smaller model, with improvements ranging from 15% to 75.5%. These results are shown in Figure 18 and source and optimized Dev Prompts can be found in the replication package [45]. Additionally, this showcases the need of optimizing prompts based on the target LLM.

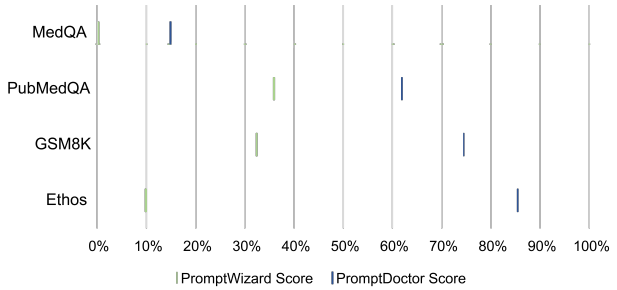


**Fig. 18:** Optimization Comparison with PromptWizard

**TABLE I:** Performance scores for grounded task prompts on synthetic and gold datasets.

| Task | Initial Prompt | | Optimized Prompt | |
|---|---|---|---|---|
| | Synthetic | Gold | Synthetic | Gold |
| Error Correction | 69.4 | 78.2 | 87.0 | 88.7 (+10.5) |
| Translation | 59.7 | 24.9 | 85.3 | 34.9 (+10.0) |
| Summarization | 80.0 | 70.1 | 86.7 | 77.8 (+7.7) |

**TABLE II:** Hyper-parameter values explored and used in optimization.

| Name | Minimum | Maximum | Value Used |
|---|---|---|---|
| # of seed prompts | 1 | 64 | 16 |
| # of prompts per step | 1 | 20 | 20 |
| Synthetic train count | 2 | 64 | 30 |

**Finding 2**

With our lightweight mitigation strategies, we were able to de-bias 68.29% of the biased and bias-prone Dev Prompts, harden 41.81% of the vulnerable prompts, and improve 37.1% of the Dev Prompts that were sub-optimal.

## VIII. Implications

**For developers.** The prevalence of Bias within LLMs poses significant challenges that developers must be aware of, as even seemingly neutral Dev Prompts can elicit biased responses due to underlying assumptions in the model. Injection Vulnerability is another critical issue—relying solely on LLM providers to block attacks is inadequate. Developers must adopt countermeasures within both Dev Prompts and code. Additionally, prompt performance can vary significantly and many prompts are not amenable to easy evaluation, leaving developers unsure of how to improve their application quality. We group our issue detection and mitigation tools and extend them with a UI and automatic Dev Prompt-extraction mechanisms from source code and supply them as a VS Code extension [45] under the name of `PromptDoctor`. We offer `PromptDoctor` as an IDE-integrated solutions to detect and address these issues. A screenshot of `PromptDoctor` is shown in Figure 19 and a demo is available at [45].

**For researchers.** This work introduces and distinguishes Dev Prompts as a unique software artifact, laying the groundwork for further exploration of prompt categories. We have

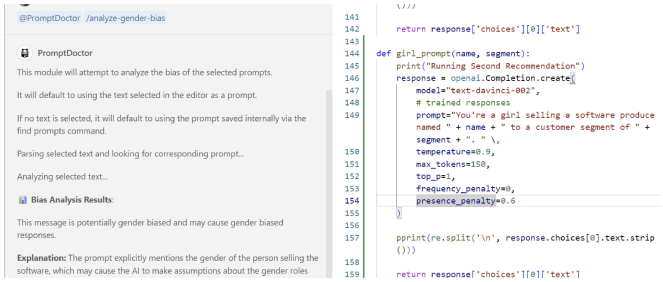


**Fig. 19:** Using Prompt Doctor for Gender-Bias and Gender-Bias-Proneness detection

empirically demonstrated the prevalence of Bias, Injection Vulnerability, and suboptimal performance in Dev Prompts, reinforcing the need for more diagnostic tools for prompt writers. Our Dev Prompt-rewriting provisional solutions offer cost-efficient, widely applicable alternatives to LLM fine-tuning, encouraging a shift towards programmatic approaches that lower the barrier of entry for addressing these issues, and which provide a springboard for future research into prompt-centric solutions.

## IX. Related Works

### A. Prompt Bias, Vulnerability and Optimization

With the growing popularity of large language models (LLMs) and the use of prompts for effectively eliciting model responses, several studies have focused on prompt bias, vulnerabilities, and sub-optimality. Cheng et al. [19] present a novel method for evaluating texts and descriptions generated by LLMs to uncover stereotypical beliefs about people from diverse backgrounds and characteristics, assessing whether these outputs contain stereotypical language. Guo et al. [67] explore how LLMs interpret literary symbolism and how such interpretations may reflect biases In their work on mitigating gender bias, Thakur et al. [68] investigate few-shot data interventions to reduce bias in LLMs. Concerning prompt vulnerabilities, Rossi et al. [23] performed an early categorization of various types of prompt injection attacks, including direct injections where prompts are altered following specific paradigms. Zou et al. [69] proposed an approach to create universal and transferable attacks against LLMs using an adversarial model, while Chao et al. [70] designed an interaction paradigm that can jailbreak LLMs in 20 interactions or fewer. These studies and others [71], [25], [24] highlight injection attacks as a significant challenge for LLMs, with additional works [72], [73] focusing on modifying LLMs to address these vulnerabilities. For prompt optimization, Wang et al. [6] applied few-shot chain-of-thought (CoT) prompting with manually crafted step-by-step reasoning. Pryzant et al. [7] utilized mini-batches of data to create natural language "gradients" for optimizing and editing existing prompts. PromptWizard [54] proposes a framework to rewrite prompts with the goal of optimizing them by using existing datasets. However, none of these studies focus on the context of Dev Prompts in open-source software (OSS), nor do they provide empirical data on the prevalence of bias in OSS or how prompts can be modified to address issues of bias and vulnerability without Fine-Tuning or modifying the LLM being used, or creating optimization data-sets.

### B. LLMs for Software Engineering

Large language models (LLMs) are gaining popularity in solving software engineering problems, much like in other domains. Recently, Wei et al. [74] developed a program repair co-pilot that uses LLMs to generate program patches synthesized from existing human-written patches. Nam et al. [75] employed LLMs for code understanding, utilizing pre-generated prompts to inquire about APIs, provide conceptual explanations, and offer code examples. Additionally, Ahmed et al. [76] worked on augmenting LLM prompts for code summarization by adding semantic facts of the code to enhance the prompts. Feng et al. [77] introduced AdbGPT, a lightweight tool that automatically reproduces bugs from bug reports, employing few-shot learning and chain-of-thought reasoning to harness human knowledge and logical processes for bug reproduction.

## X. Threats to Validity

**Internal Validity:** The main threat is the inaccuracy of Dev Prompts parsing, analysis, and rewriting components of `PromptDoctor`. To address this, we've performed benchmarking of the different components where possible, along with human validation.

**External Validity:** The different analyses within this work were performed and evaluated on PromptSet [10], which contains Dev Prompts from Python-based OSS. Due to cost and time constraints, we were unable to run our analyses on all of PromptSet, however, we believe our stratified random selection strategy has allowed us to find a representative sample of Dev Prompts.

**Construct Validity:** For this work, we performed most of our experiments using OpenAI's GPT-4o model, one of the latest and most advanced foundational models [78], and some experiments with Llama 3.1 [14]. Due to cost and time constraints, we were unable to use other LLMs, but we do believe that using similarly advanced models would produce similar results.

## XI. Conclusion

Through this work, we introduce the first empirical analysis that uncovers the prevalence of Bias, Injection Vulnerability, and Sub-optimal performance in Dev Prompts. We tackled these three issues with our lightweight solutions, and were able to de-bias 68.29% , harden 41.81%, and optimize 37.1% of flawed Dev Prompts. We provide `PromptDoctor` to developers to rewrite their Dev Prompts as part of their development process. We believe this work sheds light on a new emerging type of software artifact, and the problems it entails, and initial strategies to combat some of these issues.

## XII. Data availability

Our replication package is available at [45]. It contains the source code, an appendix, a pre-packaged `PromptDoctor` VS Code extension, and intermediary and final result files.

## ACKNOWLEDGMENT

The UofM-Dearborn authors are supported in part by NSF Award #2152819.

## REFERENCES


[1] I. Weber, "Large language models as software components: A taxonomy for llm-integrated applications," 2024. [Online]. Available: https://arxiv.org/abs/2406.10300

[2] M. U. Hadi, al tashi *et al.*, "Large language models: A comprehensive survey of its applications, challenges, limitations, and future prospects."

[3] Y. Guo, Y. Yang, and A. Abbasi, "Auto-debias: Debiasing masked language models with automated biased prompts," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 1012–1023.

[4] C. Clemmer, J. Ding, and Y. Feng, "Precisedebias: An automatic prompt engineering approach for generative ai to mitigate image demographic biases," in *2024 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2024, pp. 8581–8590.

[5] Y. Liu, Y. Jia *et al.*, "Formalizing and benchmarking prompt injection attacks and defenses," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1831–1847.

[6] L. Wang, W. Xu *et al.*, "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models," no. arXiv:2305.04091, May 2023, arXiv:2305.04091 [cs]. [Online]. Available: http://arxiv.org/abs/2305.04091

[7] R. Pryzant, D. Iter *et al.*, "Automatic prompt optimization with" gradient descent" and beam search," *arXiv preprint arXiv:2305.03495*, 2023.

[8] K. Johnson, "A chevy for $1? car dealer chatbots show perils of ai for customer service," https://venturebeat.com/ai/a-chevy-for-1-car-dealer-chatbots-show-perils-of-ai-for-customer-service/, 2023, accessed: 2025-04-04.

[9] H. Pearce, B. Ahmad *et al.*, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Commun. ACM*, vol. 68, no. 2, p. 96–105, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3610721

[10] K. Pister, D. J. Paul *et al.*, "Promptset: A programmer's prompting dataset," in *2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, 2024, pp. 62–69.

[11] G. Bharathi Mohan, R. Prasanna Kumar *et al.*, "An analysis of large language models: their impact and potential applications," *Knowledge and Information Systems*, vol. 66, no. 9, p. 5047–5070, Sep. 2024. [Online]. Available: https://link.springer.com/10.1007/s10115-024-02120-8

[12] S. Makridakis, F. Petropoulos, and Y. Kang, "Large language models: Their success and impact," *Forecasting*, vol. 5, no. 3, p. 536–549, Aug. 2023. [Online]. Available: https://www.mdpi.com/2571-9394/5/3/30

[13] OpenAI, "Gpt-4 technical report," no. arXiv:2303.08774, Mar. 2024, arXiv:2303.08774 [cs]. [Online]. Available: http://arxiv.org/abs/2303.08774

[14] M. AI, "The llama 3 herd of models," no. arXiv:2407.21783, Aug. 2024, arXiv:2407.21783 [cs]. [Online]. Available: http://arxiv.org/abs/2407.21783

[15] T. Schick and H. Schütze, "True few-shot learning with prompts – a real-world perspective," no. arXiv:2111.13440, Nov. 2021, arXiv:2111.13440 [cs]. [Online]. Available: http://arxiv.org/abs/2111.13440

[16] H. Dang, L. Mecke *et al.*, "How to prompt? opportunities and challenges of zero- and few-shot learning for human-ai interaction in creative applications of generative models," no. arXiv:2209.01390, Sep. 2022, arXiv:2209.01390 [cs]. [Online]. Available: http://arxiv.org/abs/2209.01390

[17] J. Breckenridge, "Evaluating racial bias in large language models: The necessity for "smoky"," 2024. [Online]. Available: https://www.ssrn.com/abstract=4880025

[18] I. O. Gallegos, R. A. Rossi *et al.*, "Bias and fairness in large language models: A survey," no. arXiv:2309.00770, Jul. 2024, arXiv:2309.00770 [cs]. [Online]. Available: http://arxiv.org/abs/2309.00770

[19] M. Cheng, E. Durmus, and D. Jurafsky, "Marked personas: Using natural language prompts to measure stereotypes in language models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 1504–1532. [Online]. Available: https://aclanthology.org/2023.acl-long.84

[20] S. Kim, S. Lessmann *et al.*, "Fair models in credit: Intersectional discrimination and the amplification of inequity," no. arXiv:2308.02680, Aug. 2023, arXiv:2308.02680 [cs]. [Online]. Available: http://arxiv.org/abs/2308.02680

[21] J. A. Omiye, J. C. Lester *et al.*, "Large language models propagate race-based medicine," *npj Digital Medicine*, vol. 6, no. 1, p. 195, Oct. 2023.

[22] V. Hofmann, P. R. Kalluri *et al.*, "Dialect prejudice predicts ai decisions about people's character, employability, and criminality," no. arXiv:2403.00742, Mar. 2024, arXiv:2403.00742 [cs]. [Online]. Available: http://arxiv.org/abs/2403.00742

[23] S. Rossi, A. M. Michel *et al.*, "An early categorization of prompt injection attacks on large language models," no. arXiv:2402.00898, Jan. 2024, arXiv:2402.00898 [cs]. [Online]. Available: http://arxiv.org/abs/2402.00898

[24] Y. Liu, G. Deng *et al.*, "Prompt injection attack against llm-integrated applications," Mar. 2024. [Online]. Available: http://arxiv.org/abs/2306.05499

[25] J. Yu, Y. Wu *et al.*, "Assessing prompt injection risks in 200+ custom gpts," no. arXiv:2311.11538, Nov. 2023, arXiv:2311.11538 [cs]. [Online]. Available: http://arxiv.org/abs/2311.11538

[26] K. Greshake, S. Abdelnabi *et al.*, "Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection," no. arXiv:2302.12173, May 2023, arXiv:2302.12173 [cs]. [Online]. Available: http://arxiv.org/abs/2302.12173

[27] J. White, "How strangers got my email address from chatgpt's model," *The New York Times*, Dec. 2023. [Online]. Available: https://www.nytimes.com/interactive/2023/12/22/technology/openai-chatgpt-privacy-exploit.html

[28] B. Hui, H. Yuan *et al.*, "Pleak: Prompt leaking attacks against large language model applications," no. arXiv:2405.06823, May 2024, arXiv:2405.06823 [cs]. [Online]. Available: http://arxiv.org/abs/2405.06823

[29] T. Sawada, D. Paleka *et al.*, "Arb: Advanced reasoning benchmark for large language models," no. arXiv:2307.13692, Jul. 2023, arXiv:2307.13692 [cs]. [Online]. Available: http://arxiv.org/abs/2307.13692

[30] K. Sebler, Y. Rong *et al.*, "Benchmarking large language models for math reasoning tasks," no. arXiv:2408.10839, Aug. 2024, arXiv:2408.10839 [cs]. [Online]. Available: http://arxiv.org/abs/2408.10839

[31] J. White, Q. Fu *et al.*, "A prompt pattern catalog to enhance prompt engineering with chatgpt," no. arXiv:2302.11382, Feb. 2023, arXiv:2302.11382 [cs]. [Online]. Available: http://arxiv.org/abs/2302.11382

[32] X. Amatriain, "Prompt design and engineering: Introduction and advanced methods," no. arXiv:2401.14423, May 2024, arXiv:2401.14423 [cs]. [Online]. Available: http://arxiv.org/abs/2401.14423

[33] Y. Li, Z. Lin *et al.*, "Making language models better reasoners with step-aware verifier," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 5315–5333.

[34] O. Khattab, K. Santhanam *et al.*, "Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP," *arXiv preprint arXiv:2212.14024*, 2022.

[35] P. Sahoo, A. K. Singh *et al.*, "A systematic survey of prompt engineering in large language models: Techniques and applications," no. arXiv:2402.07927, Feb. 2024, arXiv:2402.07927 [cs]. [Online]. Available: http://arxiv.org/abs/2402.07927

[36] H. Li, Q. Dong *et al.*, "Synthetic data (almost) from scratch: Generalized instruction tuning for language models," 2024.

[37] T. J. Misa, "Gender bias in big data analysis," *Information & Culture*, vol. 57, pp. 283 – 306, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253371931

[38] M. Vorvoreanu, L. Zhang *et al.*, "From gender biases to gender-inclusive design: An empirical investigation," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. Glasgow Scotland Uk: ACM, May 2019, p. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3290605.3300283

[39] E. Guzmán, R. A.-L. Fischer, and J. Kok, "Mind the gap: gender, micro-inequities and barriers in software development," *Empirical Software Engineering*, vol. 29, no. 1, p. 17, Jan. 2024. [Online]. Available: https://link.springer.com/10.1007/s10664-023-10379-8

[40] M. Bogen, "All the ways hiring algorithms can introduce bias," *Harvard Business Review*, May 2019. [Online]. Available: https://hbr.org/2019/05/all-the-ways-hiring-algorithms-can-introduce-bias

[41] M. Sap, D. Card *et al.*, "The risk of racial bias in hate speech detection," in *Proceedings of the 57th Annual Meeting of the*

*Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, 2019, p. 1668–1678. [Online]. Available: https://www.aclweb.org/anthology/P19-1163

[42] S. Maji, N. Yadav, and P. Gupta, "Lgbtq+ in workplace: a systematic review and reconsideration," *Equality, Diversity and Inclusion: An International Journal*, vol. 43, no. 2, p. 313–360, Mar. 2024. [Online]. Available: https://www.emerald.com/insight/content/doi/10.1108/EDI-02-2022-0049/full/html

[43] S. Sultana, L. A. Cavaletto, and A. Bosu, "Identifying the prevalence of gender biases among the computing organizations," no. arXiv:2107.00212, Jul. 2021, arXiv:2107.00212 [cs]. [Online]. Available: http://arxiv.org/abs/2107.00212

[44] V. K. Felkner, H.-C. H. Chang *et al.*, "Winoqueer: A community-in-the-loop benchmark for anti-lgbtq+ bias in large language models," no. arXiv:2306.15087, Jun. 2023, arXiv:2306.15087 [cs]. [Online]. Available: http://arxiv.org/abs/2306.15087

[45] [Online]. Available: https://figshare.com/s/930b08c981b41f28470c

[46] A. Radford, J. Wu *et al.*, "Language models are unsupervised multitask learners."

[47] T. B. Brown, B. Mann *et al.*, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[48] M. Samory, I. Sen *et al.*, ""call me sexist, but...": Revisiting sexism detection using psychological scales and adversarial samples," *Proceedings of the International AAAI Conference on Web and Social Media*, vol. 15, p. 573–584, May 2021. [Online]. Available: https://ojs.aaai.org/index.php/ICWSM/article/view/18085

[49] G. Glavaš, M. Karan, and I. Vulić, "Xhate-999: Analyzing and detecting abusive language across domains and languages," in *Proceedings of the 28th International Conference on Computational Linguistics*. Barcelona, Spain (Online): International Committee on Computational Linguistics, 2020, p. 6350–6365. [Online]. Available: https://www.aclweb.org/anthology/2020.coling-main.559

[50] J. Zamfirescu-Pereira, R. Y. Wong *et al.*, "Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Hamburg Germany: ACM, Apr. 2023, p. 1–21. [Online]. Available: https://dl.acm.org/doi/10.1145/3544548.3581388

[51] M. Honnibal, I. Montani *et al.*, "spaCy: Industrial-strength Natural Language Processing in Python," 2020.

[52] C. Yang, X. Wang *et al.*, "Large language models as optimizers," no. arXiv:2309.03409, Apr. 2024, arXiv:2309.03409 [cs]. [Online]. Available: http://arxiv.org/abs/2309.03409

[53] C. Fernando, D. Banarse *et al.*, "Promptbreeder: Self-referential self-improvement via prompt evolution," no. arXiv:2309.16797, Sep. 2023, arXiv:2309.16797 [cs]. [Online]. Available: http://arxiv.org/abs/2309.16797

[54] E. Agarwal, V. Dani *et al.*, "Promptwizard: Task-aware agent-driven prompt optimization framework," no. arXiv:2405.18369, May 2024, arXiv:2405.18369 [cs]. [Online]. Available: http://arxiv.org/abs/2405.18369

[55] H. Dang, S. Goller *et al.*, "Choice over control: How users write with large language models using diegetic and non-diegetic prompting," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Hamburg Germany: ACM, Apr. 2023, p. 1–17. [Online]. Available: https://dl.acm.org/doi/10.1145/3544548.3580969

[56] Y. Zhou, Z. Chen, and D. Wang, "Large language model as a surrogate optimizer," *arXiv preprint arXiv:2406.10675*, 2024. [Online]. Available: https://arxiv.org/abs/2406.10675

[57] OpenAI, "Prompt engineering," https://platform.openai.com/docs/guides/prompt-engineering, accessed: 2024-09-10.

[58] Anthropic, "Prompt engineering overview," https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview, accessed: 2024-09-10.

[59] L. Zheng, W.-L. Chiang *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[60] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 1073–1083. [Online]. Available: https://www.aclweb.org/anthology/P17-1099

[61] L. Qin, "English-spanish translation dataset," 2024. [Online]. Available: https://www.kaggle.com/datasets/lonnieqin/englishspanish-translation-dataset/data%7D

[62] S. Gunjal, "Mining & mastering the art of english corrections," Dec 2023. [Online]. Available: https://www.kaggle.com/code/satishgunjal/mining-mastering-the-art-of-english-corrections/notebook%7D

[63] D. Jin, E. Pan *et al.*, "What disease does this patient have? a large-scale open domain question answering dataset from medical exams," *arXiv preprint arXiv:2009.13081*, 2020.

[64] Q. Jin, B. Dhingra *et al.*, "Pubmedqa: A dataset for biomedical research question answering," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 2567–2577.

[65] K. Cobbe, V. Kosaraju *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[66] L. Gao, Y. Niu *et al.*, "Ethos: Rectifying language models in orthogonal parameter space," in *Findings of the Association for Computational Linguistics: NAACL 2024*. Mexico City, Mexico: Association for Computational Linguistics, 2024, p. 2054–2068. [Online]. Available: https://aclanthology.org/2024.findings-naacl.132

[67] M. Guo, R. Hwa, and A. Kovashka, "Decoding symbolism in language models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 3311–3324. [Online]. Available: https://aclanthology.org/2023.acl-long.186

[68] H. Thakur, A. Jain *et al.*, "Language models get a gender makeover: Mitigating gender bias with few-shot data interventions," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 340–351. [Online]. Available: https://aclanthology.org/2023.acl-short.30

[69] A. Zou, Z. Wang *et al.*, "Universal and transferable adversarial attacks on aligned language models," 2023.

[70] P. Chao, A. Robey *et al.*, "Jailbreaking black box large language models in twenty queries," no. arXiv:2310.08419, Oct. 2023, arXiv:2310.08419 [cs]. [Online]. Available: http://arxiv.org/abs/2310.08419

[71] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," no. arXiv:2211.09527, Nov. 2022, arXiv:2211.09527 [cs]. [Online]. Available: http://arxiv.org/abs/2211.09527

[72] J. Yi, Y. Xie *et al.*, "Benchmarking and defending against indirect prompt injection attacks on large language models," no. arXiv:2312.14197, Mar. 2024, arXiv:2312.14197 [cs]. [Online]. Available: http://arxiv.org/abs/2312.14197

[73] X. Suo, "Signed-prompt: A new approach to prevent prompt injection attacks against llm-integrated applications," 2024.

[74] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 172–184. [Online]. Available: https://doi.org/10.1145/3611643.3616271

[75] D. Nam, A. Macvean *et al.*, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639187

[76] T. Ahmed, K. S. Pai *et al.*, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639183

[77] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608137

[78] OpenAI, "Gpt-4o system card," Aug. 2024.