Benchmarking Causal Study to Interpret Large Language Models for Source Code

Daniel Rodriguez-Cardenas, David N. Palacio, Dipin Khati, Henry Burke and Denys Poshyvanyk
Department of Computer Science, William & Mary
Williamsburg, VA
Email: dhrodriguezcar, danaderpalacio, dkhati, hqburke, dposhyvanyk{@wm.edu}

***Abstract*—One of the most common solutions adopted by software researchers to address code generation is by training Large Language Models (LLMs) on massive amounts of source code. LLMs are rooted in the concept of emergent capabilities in which machines statistically learn complex patterns from code data. Although a number of studies have shown that LLMs have been effectively evaluated on popular accuracy metrics (*e.g.,* BLEU, CodeBleu), previous research has largely overlooked the role of Causal Inference as a fundamental component of the interpretability of LLMs' performance. Existing benchmarks and datasets are meant to highlight the difference between the expected and the generated outcome, but do not take into account confounding variables (*e.g.,* lines of code, number of tokens, prompt size) that equally influence the accuracy metrics. The fact remains that, when dealing with generative software tasks by LLMs, no benchmark is available to tell researchers how to quantify neither the causal effect of SE-based treatments nor the correlation of confounders to the model's performance. In an effort to bring statistical rigor to the evaluation of LLMs, this paper introduces a benchmarking strategy named *Galeras* comprised of curated testbeds for three SE tasks (*i.e.,* code completion, code summarization, and commit generation) to help aid the interpretation of LLMs' performance.**

We illustrate the insights of our benchmarking strategy by conducting a case study on the performance of ChatGPT under distinct prompt engineering methods. The results of the case study demonstrate the positive causal influence of prompt semantics on ChatGPT's generative performance by an *average treatment effect* of $\approx 3\%$. Moreover, it was found that confounders such as prompt size are highly correlated with accuracy metrics (≈ 0.412). The end result of our case study is to showcase causal inference evaluations, *in practice*, to reduce *confounding bias*. By reducing the bias, we offer an interpretable solution for the accuracy metric under analysis.

***Index Terms*—Software Engineering, Testbeds, Large Language Models, dl4se, Interpretability**

I. Introduction

Deep Learning for Software Engineering (*DL4SE*) is an emerging research area in the field of software maintainability that entails a paradigm shift in the form by which machines statistically learn complex patterns from code data. To support actionable downstream SE tasks (*e.g.,* code completion, code summarization, or commit generation), ample evidence supports that *DL4SE* approaches in the form of Language Models are able to generate code conditioned on a well-defined prompt [1]–[3]. While essential, *DL4SE* approaches have been reduced to a group of large and self-supervised neural architectures (*i.e.,* Large Language Models or simply LLMs) comprised of multiple self-attention layers that perform linear transformations to extract salient features from programming and natural language data. In particular, Large Language Models for Code (LLMc) have led to a renewed interest in the automation of software engineering tasks. Most of this automation is a generative process in which underlying code and natural language features interact with each other to auto-complete [4]–[9], summarize [10]–[12], review [13]–[16], trace [17] and translate code [18]; generate test cases [19]–[21], detect cone clones [22], [23] or fix bugs [24]–[31]. In fact, LLMc have been deployed in large-scale solutions to provide code generative services. Tools such as ChatGPT and GitHub Copilot, which are based on the *gpt* architecture, exhibit good performance at the aforementioned tasks [2].

Therefore, an increased interest has emerged in further evaluating these LLMc [32]–[35] to standardize the quality assessment of the generated code. Unfortunately, the current evaluation process overly-relies on accuracy metrics leaving no consensus as to what other features or properties are impacting the code generation process. In other words, we require to control for factors that influence the performance of LLMc if our goal is to *interpret* models' output. Few studies have sought to examine accuracy metrics from a causal perspective to interpret LLMc [36]. Ergo, the problem remains that, when attempting to understand the prediction performance of LLMc, no benchmarks are available to articulate causal queries.

Previous research has largely overlooked the role of causal inference in evaluating LLMc. In fact, existing benchmarks are not without flaws to detect *confounding bias*, which refers to the statistical ability to control for variables that can influence models' performance beyond the SE treatments under study (*i.e.,* evaluating the best prompting method). That is, we study causation because we need to understand not only *what* but also *why* LLMc arrive at performance decisions. To overcome these challenges, we pose a code-based benchmarking strategy, named *Galeras*, to interpret LLMc concentrated on answering causal queries of interest. *Galeras* enables SE researchers to explain LLMc performance decisions from a curated set of code-based confounders, which are associated with a given SE treatment under study. *Galeras* is comprised of three parts: 1) seven testbeds for evaluating distinct SE downstream tasks free of sampling bias and data snooping, 2) a set of confounders to compute causal effects, and 3) a pipeline to curate data from open repositories.

To illustrate how to exploit *Galeras* to interpret LLMc, we conducted a causal study to quantify the impact of confounding variables on ChatGPT's prediction performance to assess whether certain types of *prompt engineering* methods are excelling at automating code completion tasks. Prompt engineering is associated with the emergent ability of LLMs to learn from prompts (*i.e.,* in-context learning). This ability comprises a set of techniques that manipulates the structure of a LLM's input sequence to attain better and less computationally expensive outputs than applying other downstream methods such as fine-tuning [33]. We organize our study around two RQs that are fundamentally centered on the problem of *prompt engineering* for code:

RQ$_1$ Exploratory Analysis: *How different is the distribution of tokens between the generated and ground-truth code?*

RQ$_2$ Causal Analysis: *To what extent the type of Prompt Engineering is influencing the code completion performance?*

The achieved results show that prompt engineering methods indeed causally impact the accuracy of the model by an *Average Treatment of Effect* (ATE) of 3% between the semantics of the prompt and the accuracy metric. Hence, choosing an adequate prompting strategy can positively influence the code completion performance of ChatGPT. To summarize, our key contributions are: 1) A filtered testbed with non-contaminated code snippets for LLMc benchmarking; 2) a set of (confounding) features (*e.g.,* Cyclo Complexity, # of AST levels) included in the testbed; 3) a pipeline to generate new testbeds for a given SE task; and 4) a causal inference benchmarking to interpret LLMc.

II. Related Work

Considerable research attention has been devoted to data collection and benchmarking for LLMc. Tab.I showcases eight qualitative properties that we use to compare three state-of-art benchmarks (*i.e., CodeXGLUE*, *IdBench*, and *MultiPL-E*) with *Galeras*. Firstly, Husain *et al.* introduced *CodeSearchNet* for code retrieval automation [37]. Their datasets have been mostly employed to pre-train LLMs rather than benchmarking software tasks. Later, researchers at Microsoft extended *CodeSearchNet* and amalgamated 12 SE-related datasets for other relevant downstream tasks (*e.g.,* clone detection, refinement, translation) [38]. These datasets and benchmarks are known as *CodeXGLUE*, which partially support some accuracy and distance metrics. Secondly, Wainakh *et al.* proposed *IdBench* to evaluate generated identifiers by measuring similarity distances of semantic representations [39]. Finally, Chen *et al.* notably posed *HumanEval* to validate the functional correctness of generated code [35]. Cassano *et al.* amplified *HumanEval* to create *MultiPL-E* for code translation [40]. Although these three benchmarks have been successfully employed for evaluating LLMc, these benchmarking strategies were not conceived to address the *interpretation* of models' outputs.

As LLMc are quickly evolving due to data and hyperparameter augmentation, current models (*e.g.,* ChatGPT, AlfaCode, Copilot) could have been trained on samples already used

TABLE I: SOTA Benchmark qualitative properties comparison.

Qualitative Properties		*Benchmarks*			
		CodeXGLUE	IdBench	MultiPL-E	Galeras
Software Tasks	Clone detection	✓	✗	✗	✗
	Defect detection	✓	✗	✗	✗
	Type Inferring	✗	✓	✗	✗
	Summarization	✗	✗	✗	✓
	Code generation	✗	✗	✗	✓
	Commit generation	✗	✗	✗	✓
	Repair	✓	✗	✗	✗
	Translation	✓	✗	✓	✗
	Search	✓	✗	✗	✗
I/O	code-code	✓	✗	✗	✓
	code-text	✓	✓	✗	✓
	text-code	✓	✗	✓	✗
Output Granularity	Identifiers	✗	✓	✗	✗
	Code line	✓	✗	✗	✗
	Method	✓	✗	✓	✓
	Files	✓	✗	✗	✗
Type of Datum	Words	✗	✓	✗	✗
	Tokens	✓	✗	✗	✓
	Snippets	✓	✗	✓	✓
	Prompts	✗	✗	✗	✓
Dimension	Size	416K	500 answers	164 problems	227K
	Languages	≈ 12	3	19	1
Supported Metrics	BLEU	✓	✓	✗	✓
	CodeBLEU	✓	✗	✗	✓
	Cloze testing	✓	✗	✗	✗
	Levenshtein	✗	✓	✗	✓
	Accuracy	✓	✗	✗	✗
	Causal Effect	✗	✗	✗	✓
Prompt Engineering [33]	Single-step	✗	✗	✓	✓
	Multiple-step	✗	✗	✗	✓
Causal Evaluation	Confounders	✗	✗	✗	✓tab.II
	Inference	✗	✗	✗	✓

Shadowed cells indicate *Galeras* only.

for evaluation (*a.k.a.* data snooping) and datasets such as *BigQuery* [41], *BigPython* [42], and the *Pile* [43] have omitted the importance of interpreting LLMc' performance. *Galeras*, however, offers curated testbeds for enabling prompt engineering evaluation. This evaluation includes an interpretability analysis based on *causal inference* in the form of Structural Causal Models (SCM). What is more, *Galeras* provides a pipeline to collect and access confounders and treatment data. Such data is plugged into the SCM to estimate the causal effects between treatments and outcomes. Estimating these casual effects promote statistical rigor in evaluating SE-based generative tasks.

III. Testbed Curation Pipeline

This section considers our proposed pipeline to structure and collect required testbeds for the comparative causal evaluation of LLMc. *Galeras* is a benchmarking strategy that entails a software architecture solution for the curation process.

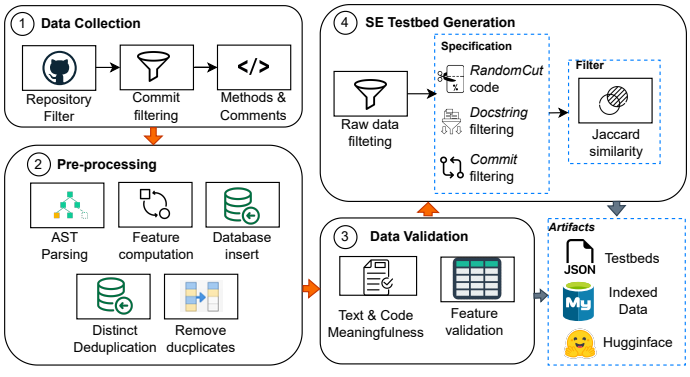

Fig. 1: Testbed Curation Pipeline of *Galeras*

A. Structuring Testbed's Features

Galeras testbeds are sets of Python methods that serve as evaluative data points. Each data point comprises four dimensions. The first dimension corresponds to snippets' identification, which includes the *commit_id* (*i.e.,* commit hash), *repository* name, *path*, *file_name*, and *fun_name*. The second dimension corresponds to snippets' documentation, which includes the *commit_message* and *docstring*. The *docstring* belongs to a JSON object that is extended to complementary natural language features such as *n_words, vocab_size, language,* and *n_whitespaces*. The third dimension corresponds to the snippet's syntactic information, which includes the actual code base, *n_ast_errors*, *n_ast_levels*, *n_ast_nodes*,*n_words*, *vocab_size*, *token_count*, and *n_whitespaces*. Finally, the fourth dimension corresponds to canonical software metrics, which include *nloc*, *complexity*, and *n_identifiers*.

B. Collecting Code Samples

Figure 1 describes a 4-step pipeline that *Galeras* uses to collect code samples. In the first step (Fig. 1-①), we filtered the most popular Python Github repositories using the following query: $language : Python$, $fork : false$, $size :>= 30,000$, $pushed :> 2021-12-31$, $stars :> 1,000$. From the last paper report of ChatGPT [44], we assumed ChatGPT and other LLMc under analysis were not trained on commits from Jan 2, 2022 to Jan 1, 2023. Therefore, we claim that our testbeds help to avoid *data snooping*, which is the misuse of data points to evaluate statistical hypotheses using training samples. Then, we collected a set of brand-new methods for each commit. This step resulted in $\approx 338k$ data points. For each data point, we also collected its corresponding documentation without considering inline comments.

In the second step (Fig. 1-②), we engineered and preprocessed both code and documentation-related features from collected data points. Then we parsed the AST variables for our data points by employing the Tree-Sitter library. To guarantee efficient data management and once the previous features were engineered and extracted, we stored raw and preprocessed data points in a relational database. Next, we removed duplicated samples using a distinct query reducing the testbeds size to $\approx 227K$ data points for code (*RawData* in tab. II). Of these reduced data points, $\approx 77K$ contains a valid *docstring* (*RawDataDocstring* in tab. II). A *docstring* is valid when its text is larger than 3 words.

In the third step 1-③), we manually validated 960 out of $\approx 227K$ data points. These validated data points were randomly selected from *RawData* and *RawDataDocstring*. The remaining data points were automatically validated. Our validation process ensures the date of each pushed commit is within the range of dates stated in the original query. We also validated that the methods attached to each commit were indeed updated within the same range of dates. In addition, we validated the meaningfulness of the *docstring* and *commit_message* by inspecting the consistency of the natural language descriptions with the actual code implementation, removing $\approx 1.9\%$ *RawDataDocstring* obtaining $\approx 57K$ datapoints (tab. II). Lastly, *complexity* was validated using the Codalyze plugin in Visual Studio Code. For the sake of simplicity, we omit explaining all considered fine-grained validation steps in this paper. However, the reader can consult our online appendix for more information [45].

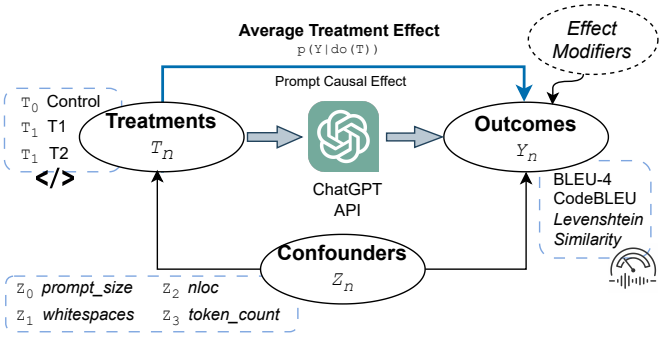

Fig. 2: *Galeras* Structural Causal Model Benchmarking

In the final step (Fig.1-④), we sampled $3k$ data points from *RawData* testbed to build five additional testbeds, each one for a specific SE task. *Galeras* comprises *RandomCut*, *WithDocString* and *FromDocString* for *code completion*; *CommitGen* for *code generation*; and *SummarizationGen* for *code summarization*. These additional testbeds are described in Tab. II. To build *RandomCut*, we chose data points with more than 10 tokens or 100 characters. Next, the data point is randomly cut after the method signature. To build *SummarizationGen* and *CommitGen*, we filtered the *RawDataDocstring* data points with more than 10 words or 50 characters. After building the five testbeds, we removed duplicated snippets using the Jaccard similarity on preprocessed data points with BPE HuggingFace tokenizer. Because the de-duplication between training and test sets was discarded (*i.e.,* no multiset threshold), we set 0.7 as the similarity threshold for our testbeds [46], [47]. Table. III shows the SE Task associated with each curated testbed, the percentage rate of detected duplicates, and the final size.

IV. Causal Analysis for Interpretable LLMc

Galeras is a causal benchmarking to compare the performance of LLMc against each other by controlling for *confounding variables*, which are features of the source code that can influence the prediction performance of LLMc. Ideally, researchers can use *Galeras* to contextualize the outcomes of LLMc by presenting possible tailored treatment variables that explain the behavior of the model. *Galeras*' goal is to empower the research community to interpret typical performance metrics by stating the assumptions of the prediction problem in a *Structural Causal Model* (SCM). The SCM comprises four random variables. The first variable is the *treatments* T, which represents the input configuration prompts in our case study. The second variable is the *potential outcomes* Y, which is the model prediction performance measured using distance metric (*e.g.,* BLEU, CodeBLEU, Levenshtein). The third variable is the *confounders* Z, which represents variables affecting both

TABLE II: Descriptive Analysis $[avg \pm std]$ of *Galeras*' Testbeds and Code Features.

	Confounders*				Effect modifiers					
Testbed	**Dedup**	**n_whitespaces**	**nloc**	**token_counts**	**n_ast_errors**	**ast_levels**	**n_ast_nodes**	**complexity**	**token_counts**	**n_identifiers**
RawData	277226	259.23 ± 902.22	21.16 ± 47.46	137.38 ± 262.59	0.09 ± 0.42	11.85 ± 3.5	221.91 ± 438.23	3.25 ± 6.98	137.38 ± 262.59	17.94 ± 16.45
RawDataDocstring	57045	206.98 ± 453.06	18.89 ± 30.98	112.50 ± 183.78	0.10 ± 0.73	11.57 ± 3.45	184.53 ± 436.76	3.42 ± 6.61	112.50 ± 183.78	15.96 ± 14.48
RandomCut	2931	229.24 ± 479.38	18.27 ± 26.98	126.54 ± 177.19	0.10 ± 0.30	12.25 ± 3.06	207.55 ± 259.46	3.16 ± 6.09	126.54 ± 177.19	17.70 ± 13.42
WithDocstring	2926	208.48 ± 414.67	18.08 ± 20.65	111.98 ± 122.27	0.08 ± 0.58	12.22 ± 3.10	188.99 ± 400.74	3.78 ± 4.33	111.98 ± 122.27	16.61 ± 11.50
FromDocsting	2937	167.96 ± 244.56	16.68 ± 20.91	100.13 ± 118.36	0.10 ± 0.59	11.38 ± 3.44	156.39 ± 180.71	3.48 ± 4.48	100.13 ± 118.36	14.62 ± 11.94
CommitGen	2919	179.62 ± 363.64	16.75 ± 21.37	101.66 ± 128.65	0.09 ± 0.36	11.51 ± 3.36	160.30 ± 201.11	3.28 ± 4.93	101.66 ± 128.65	15.07 ± 11.90
SummarizationGen	2924	212.08 ± 415.90	18.66 ± 21.63	114.38 ± 128.94	0.07 ± 0.32	12.22 ± 3.16	197.05 ± 532.69	3.85 ± 5.36	114.38 ± 128.94	16.71 ± 12.16

*The confounder *prompt_size* was omitted due to its treatment dependency. We measured its correlations in Tab. IV

TABLE III: Jaccard Similarity de-duplication

SE Task	Testbed	I/O	Dupes	Dupe %	size
Code completion	*RandomCut*	code ⇒ code	69	2.30%	2931
	WithDocString	code-text ⇒ code	74	2.47%	2926
	FromDocString	text ⇒ code	63	2.10%	2937
Code generation	*CommitGen*	code ⇒ text	81	2.70%	2919
Summarization	*SummarizationGen*	code ⇒ text	76	2.53%	2924

T and Y (see Fig. 2). The last variable is the *effect modifiers*, which is the features directly affecting outcomes Y.

The purpose of the causal analysis is to eliminate *spurious correlations* between the treatments T and the outcomes Y by controlling for confounding features Z. The elimination of the confounding features can be formally described with both an SCM and the do-operator introduced by Pearl *et al.* [48]. We measure the *Average Treatment Effect* (ATE) by approximating the conditional probability $p(Y|do(T))$ with statistical methods such as the propensity score matching, stratification, or IPW [48], [49]. An in-depth analysis and explanation of causal inference methods are beyond the scope of this paper.

V. Causal Study: Interpretable Code Completion

To demonstrate how to employ *Galeras* for causal analysis, in practice, we design a study in which we evaluate ChatGPT's performance for two prompt engineering methods T_1 and T_2 based on Liu *et al.* [33]. Prompt engineering is the activity of optimizing the input space of a given LLM in order to generate better outcomes without giving rise to expensive fine-tuning. The goal of our case study is to compare these two prompting methods after controlling for confounding features.

A. Evaluation Methodology

The evaluation methodology of the case study is divided into three parts. The first part addresses the exploratory analysis of *Galeras* testbeds. We employed the BPE tokenizer to normalize the vocabulary of each treatment T and outcome Y sentence. The token count categorized by taxonomy is presented in Fig.3. Tokens within each sentence were classified based on their taxonomy, *i.e., 'try'* and *'catch'* are classified as *exceptions* and *'if'* and *'else'* as conditionals. Since the analysis focused solely on Python, keywords related to data types were classified as *casting* tokens.

The second part canonically evaluates ChatGPT using our testbed *WithDocString*. CodeBLEU was computed with a default parameter value of 0.25. In addition, BLUE was computed with a 4-gram parameter. On the other hand, we computed the Levenshtein distance and similarity for a local evaluation (see Tab .IV-Performance Metrics).

The third part estimates the causal effect of prompt engineering methods and ChatGPT performance. Figure 2 illustrates our Structural Causal Models for the prompt engineering case of ChatGPT. We use *Galeras* to compare the performance of two different treatments. The first treatment T_1 is one prompt, which contains a command (*e.g., Complete the following a Python code, return only code and complete method: '{partial code}'*) followed by the actual input code to be completed. The second treatment T_2 comprises two prompts. The first one is a context prompt that entails both the *docstring* and the incomplete cut code. The second one is a *processing prompt* that contains sentences asking for removing comments and optimizing code (*e.g.,* Remember you have a Python function named '{ *fun_name* }', the function starts with the following code '{code}'. The description for the function is: '{ *docstring* }'). We used the previous treatments against a *control* group. The *control* is a *task prompt* that encompasses an action word or verb followed by the incomplete code input (*e.g.,* Complete the following python method: '{partial code}'). To evaluate whether treatments T are impacting ChatGPT performance Y, we controlled for confounding features Z. Our confounders *prompt_size*, *n_whitespaces*, *token_count*, and *nloc* were selected due to their high correlation ($[0.4-0.8]$) with the Levenstein distance in control and treatment groups. Although *n_ast_nodes* has a high correlation with the Levenstein distance, we assumed that structural features are ignoring the treatments. Hence, AST-based features are effect modifiers. The potential outcomes Y_2,Y_1,Y_0 are observed under the treatments T_1,T_2,*control*. Next, we approximate the *Average Treatment Effect* $p(Y|do(T)$ using the SCM defined in Fig .2.

B. Results

RQ$_1$ *Exploratory Analysis.* The purpose of the exploratory analysis is to expose and understand the testbeds' feature distribution grouped by prompt engineering methods T. Table II depicts the average and standard deviation for each code feature. We observed high variability in *n_whitespaces* (902.22) and *token_count* (262.59), which implies the method sizes are not homogeneous across the testbeds. While the descriptive analysis showcases high variability for all code features, our testbeds are a representative sub-sample of open repositories. For instance, the *complexity* feature has an average value of 3.25 suggesting that the code has a reasonable number of loops, conditionals, and operators. Therefore, our collected methods exhibit that our pipeline process guarantee data point diversity.

We observed no significant differences in the counting of tokens among potential outcomes (including the *control*)

and the ground truth (see Fig. 3-A). For instance, *control* and T_2 on declarations (with a diff. around 550 tokens) and loops (with a diff. around 600 tokens) are relatively small. However, T_1 outcome exhibited high difference and excessive use of OOP, declarations, and loops with a diff. around $2.6k$, $2k$, and $1.5k$ tokens respectively. Figure 3-B showcases the token distribution for each testbed. We detected that the two prompt engineering methods were generating a similar amount of tokens (*i.e.,* green and red distributions) compared to the *control* and ground truth. This suggests that sophisticated prompts tend to generate repetitive tokens. Figure 3-C depicts the Levenshtein similarity distance between the ChatGPT outputs, generated with both prompt engineering methods and the *control*, and the ground truth. We can observe from the proportion curve that T_1 similarity performs the worst compared to the *control* and T_2.

RQ$_1$ Exploratory Analysis: Grouped by taxonomy the ground truth does not repeat the same tokens as much as the treatments. The T_1 outcome seems to have notable intense use of keywords for OOP, declarations, and loops; T_2 obtains better performance with the highest similarity average of 0.43

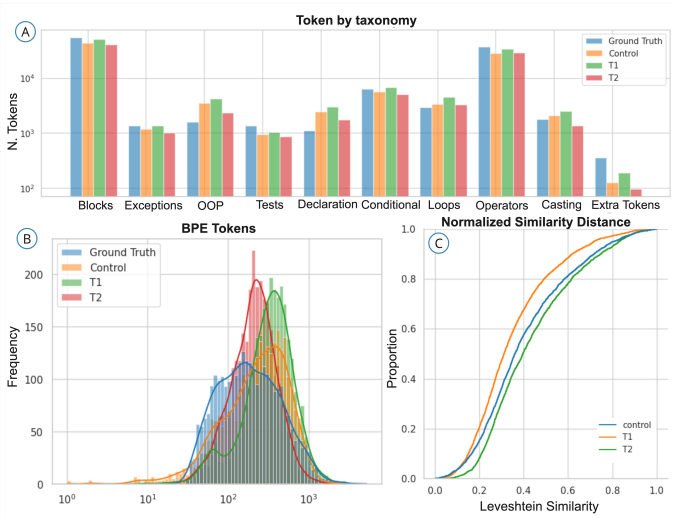

Fig. 3: Descriptive Analysis: Top graph the token count for each testbed, bottom left the token frequency distribution, bottom right the similarity proportion score.

RQ$_2$ *Causal Analysis.*

For two basic prompt engineering methods, code completion performance of ChatGPT is mainly affected by the following confounders: number of white spaces, lines of code, tokens in the outcome, and tokens in the prompt with a maximum correlation of 0.80 with the Levenstein distance (see Tab. IV-Correlations). This suggests that after controlling for confounders, the *Average Treatment Effect* (ATE) the prompt engineering method$_1$, represented by T_1, has a negative causal effect $p_1(Y|do(T)) = E[Y_1 - Y_0] \approx -5.1\%$ compared to a positive causal effect $p_2(Y|do(T)) = E[Y_2 - Y_0] \approx 3.3\%$ of method$_2$, represented by T_2 (see Tab. IV-Causal Effects). This indicates that method$_1$ is negatively affecting the Levenshtein similarity (*i.e.,* poor performance) across *WithDocString* testbed, while method$_2$ is actually enhancing ChatGPT prediction performance. These results are consistent with the previous section in which we demonstrated that T_2 performs better than T_1. After controlling for the confounding effect of the code features such as the prompt size and token counts, we can claim that the reason why T_2 is performing better than T_1 is *purely* due to the information contained in the prompt.

In order to validate the robustness of computed ATEs and proposed SCM, we refuted our effects using the following methods: *Placebo, Random Common Cause (RCC)* and *Subet* (see DoWhy refuters in [49]). We found that, for the ATEs computed with score matching, their corresponding refutation values are not stable. That is, the placebo value for Y_1 similarity is far from zero with 2.98, while the RCC value differs by around 212 in Y_2 distance.

TABLE IV: Code Completion Testbed Results: Performance Metrics, Correlations, and Causal Effects.

Treatments		**Control**		**T1**		**T2**	
Performance Metrics							
Distance	Bleu	0.444		0.45		0.42	
	CodeBleu	0.441		0.438		0.469	
Similarity	Avg. Lev.	0.40±0.20		0.35±0.18		0.43±0.20	
Correlations (vs Levenshtein)		**Dist.**	**Sim.%**	**Dist.**	**Sim.%**	**Dist.**	**Sim.%**
Confounders	*prompt_size*	0.45	25.6%	0.40	41.2%	0.45	28.3%
	n_whitespaces	0.69	5.6%	0.62	20.7%	**0.80**	1.8%
	token_count	0.67	5.3%	0.59	24.8%	0.70	3.9%
	nloc	0.64	4.2%	0.57	20.7%	0.70	0.1%
Effect Modifiers	*complexity*	0.43	4.3%	0.40	16.8%	0.47	0.9%
	n_ast_nodes	0.72	7.8%	0.62	29.4%	0.77	4.3%
	n_ast_errors	0.02	-2.4%	0.05	3.7%	0.18	2.3%
	n_ast_levels	0.40	9.9%	0.31	30.4%	0.44	8.1%
Causal Effects $(T \rightarrow Y)$							
Score Matching	ATE	-	-	104.02	-3.7%	-314.36	6.9%
	Placebo	-	-	-0.21	**298%**	0.02	0.1%
	RCC	-	-	112.14	-5.2%	-102.71	3.3%
	Subset	-	-	110.85	-5.1%	-101.6	3.3%
Stratification	ATE	-	-	111.05	-5.1%	-101.73	3.3%
	Placebo	-	-	-0.17	0.04%	0.01	0.05%
	RCC	-	-	111.17	-5.1%	-101.7	3.3%
	Subset	-	-	110.95	-5.2%	-101.49	3.3%
IPW	ATE	-	-	111.05	-5.1%	-101.73	3.3%
	Placebo	-	-	-0.54	-0.02%	-1.30	-0.07%
	RCC	-	-	111.04	-5.1%	-101.74	3.3%
	Subset	-	-	111.12	-5.1%	-101.47	3.3%

bold: highest correlation, underline: null effect.

RQ$_2$ Causal Analysis: The prompt engineering method$_1$ (treatment T_1) has a negative causal impact on the ChatGPT performance with an ATE estimation of -5%. Conversely, the prompt engineering method$_2$ (treatment T_2) has a subtle positive influence on the same performance with an ATE of 3%. This suggests that after controlling for prompt size, white spaces, # of tokens, and nlocs; prompt engineering strategies are indeed affecting the quality of code completion.

VI. Conclusion & Future Work

This study used a qualitative technique to analyze the causal effect of SE-oriented treatments on the performance of LLMc. Such a technique is embedded into a benchmarking strategy named *Galeras*. Our benchmarking enables researchers to interpret *why* a given LLMc is reporting a particular accuracy metric. We curated two raw Python testbeds: *RawData* with only mined code and *RawDataDocstring* with the corresponding documentation from GitHub. We also provide five SE

Python testbeds for three SE tasks (*i.e.,* code completion, code summarization, and commit generation), we proposed a pipeline for collecting testbeds from git repositories. Finally, we conducted a rigorous evaluation of code completion with ChatGPT. Our causal study suggests that ChatGPT's performance is not only affected by the prompt size but also by the prompt semantics. Future research will focus on determining whether other unmeasured confounders are affecting LLMc's prediction by augmenting the number of testbeds.

VII. Acknowledgement

This research has been supported in part by the NSF CCF-2311469, CNS-2132281, CCF-2007246, and CCF-1955853. We also acknowledge support from Cisco Systems. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] C. Watson, N. Cooper *et al.*, "A systematic literature review on the use of deep learning in software engineering research."

[2] W. X. Zhao, K. Zhou *et al.*, "A Survey of Large Language Models," Apr. 2023, arXiv:2303.18223 [cs].

[3] D. Zan, B. Chen *et al.*, "Large Language Models Meet NL2Code: A Survey," May 2023, arXiv:2212.09420 [cs].

[4] J. Austin, A. Odena *et al.*, "Program synthesis with large language models," 2021.

[5] D. Hendrycks, S. Basart *et al.*, "Measuring coding challenge competence with APPS," *CoRR*, vol. abs/2105.09938, 2021.

[6] M. Chen, J. Tworek *et al.*, "Generation Probabilities are Not Enough: Improving Error Highlighting for AI Code Suggestions," 2021, publisher: arXiv Version Number: 2.

[7] M. White, C. Vendome *et al.*, "Toward deep learning software repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 334–345.

[8] M. Ciniselli, N. Cooper *et al.*, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4818–4837, 2022.

[9] M. Ciniselli, N. Cooper *et al.*, "An empirical study on the usage of bert models for code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 108–119.

[10] Y. Hussain, Z. Huang *et al.*, "Deep transfer learning for source code modeling," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 30, pp. 649–668, 2020.

[11] A. LeClair, A. Bansal *et al.*, "Ensemble Models for Neural Source Code Summarization of Subroutines," Jul. 2021, arXiv:2107.11423 [cs].

[12] K. Moran, A. Yachnes *et al.*, "An empirical investigation into the use of image captioning for automated software documentation," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 514–525.

[13] A. Mastropaolo, N. Cooper *et al.*, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1580–1598, 2023.

[14] A. Mastropaolo, S. Scalabrino *et al.*, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 336–347.

[15] R. Tufano, L. Pascarella *et al.*, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 163–174.

[16] R. Tufano, S. Masiero *et al.*, "Using pre-trained models to boost code review automation," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2291–2302.

[17] K. Moran, D. N. Palacio *et al.*, "Improving the effectiveness of traceability link recovery using hierarchical bayesian networks," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 873–885.

[18] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *ICSE'15*. IEEE Press, 2015, p. 858–868.

[19] R. White and J. Krinke, "ReAssert: Deep Learning for Assert Generation," Nov. 2020, arXiv:2011.09784 [cs].

[20] V. Raychev, M. T. Vechev *et al.*, "Code completion with statistical language models," *PLDI*, 2014.

[21] C. Watson, M. Tufano *et al.*, "On learning meaningful assert statements for unit test cases," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1398–1409.

[22] M. White, M. Tufano *et al.*, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.

[23] M. Tufano, C. Watson *et al.*, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 542–553.

[24] M. Tufano, C. Watson *et al.*, "Learning How to Mutate Source Code from Bug-Fixes," *ICSME 2019*, pp. 301–312, 2019.

[25] Y. Zhou, S. Liu *et al.*, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks."

[26] M. White, M. Tufano *et al.*, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 479–490.

[27] M. Tufano, J. Pantiuchina *et al.*, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 25–36.

[28] M. Tufano, C. Watson *et al.*, "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, pp. 1–29, 2019.

[29] M. Tufano, C. Watson *et al.*, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 832–837.

[30] Z. Chen, S. Kommrusch *et al.*, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.

[31] A. Connor, A. Harris *et al.*, "Can we automatically fix bugs by learning edit operations?" in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2022, pp. 782–792.

[32] J. Liu, C. S. Xia *et al.*, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," 2023.

[33] C. Liu, X. Bao *et al.*, "Improving chatgpt prompt for code generation," 2023.

[34] F. F. Xu, U. Alon *et al.*, "A Systematic Evaluation of Large Language Models of Code," May 2022, arXiv:2202.13169 [cs].

[35] M. Chen, J. Tworek *et al.*, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374 [cs].

[36] D. N. Palacio, N. Cooper *et al.*, "Toward a Theory of Causation for Interpreting Neural Code Models," Feb. 2023, arXiv:2302.03788 [cs, stat].

[37] H. Husain, H.-H. Wu *et al.*, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[38] S. Lu, D. Guo *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation."

[39] Y. Wainakh, M. Pradel *et al.*, "Evaluating semantic representations of source code," *arXiv*, 2019, arXiv: 1910.05177.

[40] F. Cassano, J. Gouwar *et al.*, "MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation," Dec. 2022, arXiv:2208.08227 [cs].

[41] "Bigquery datasets: https://cloud.google.com/bigquery," 2021.

[42] E. Nijkamp, B. Pang *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," 2023.

[43] L. Gao, S. Biderman *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," 2020.

[44] OpenAI, "Gpt-4 technical report," 2023.

[45] D. Rodriguez-Cardenas, D. N. Palacio *et al.*, "https://github.com/wm-semeru/galeras-benchmark," 2023.

[46] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *OOPLSA*, 2019, pp. 143–153.

[47] C. Wang, K. Cho *et al.*, "Neural Machine Translation with Byte-Level Subwords," Dec. 2019, arXiv:1909.03341.

[48] J. Pearl, *Causality: models, reasoning, and inference*, 2009.

[49] A. Sharma, V. Syrgkanis *et al.*, "DoWhy : Addressing Challenges in Expressing and Validating Causal Assumptions," 2021.