CodeSCM: Causal Analysis for Multi-Modal Code Generation

Mukur Gupta* **Noopur Bhatt*** **Suman Jana**
Columbia University
{mukur.gupta, noopur.bhatt}@columbia.edu
suman@cs.columbia.edu

Abstract

In this paper, we propose CodeSCM, a Structural Causal Model (SCM) for analyzing multi-modal code generation using large language models (LLMs). By applying interventions to CodeSCM, we measure the causal effects of different prompt modalities, such as natural language, code, and input-output examples, on the model. CodeSCM introduces latent mediator variables to separate the code and natural language semantics of a multi-modal code generation prompt. Using the principles of Causal Mediation Analysis on these mediators we quantify direct effects representing the model's spurious leanings. We find that, in addition to natural language instructions, input-output examples significantly influence code generation.

1 Introduction

Modern Large Language Models (LLMs) have shown remarkable effectiveness in code reasoning tasks, particularly code generation (Nijkamp et al., 2023; Rozière et al., 2023; Bai et al., 2023). This task involves generating code that meets specific multi-modal requirements, constrained by natural language instructions, code snippets, and input-output (I/O) example pairs (Chen et al., 2021a; Austin et al., 2021). Additionally, some multi-modal prompt components contain information from both code and natural language modalities (Casalnuovo et al., 2020), such as function signatures and variable names, where code structure and natural language coexist. This enriched coding context, combining programming and natural language semantics, helps LLMs better understand both the semantics and syntactic requirements of the desired code.

Prior research has shown the effectiveness of prompt tuning in improving generation performance (Brown et al., 2020; Liu et al., 2021; Wei et al., 2023). These works have shown that multi-modal prompts can be highly sensitive, where small adjustments might result in drastically different responses from the model (Chao et al., 2023; Zhu et al., 2023; Sclar et al., 2023). However, the interactions between the multi-modal components of code generation prompts and their direct or indirect effects on the generated code are still not well understood.

In this paper, we systematically explore these complex multi-modal effects using a causal approach. We propose a novel causal framework, CodeSCM, to measure the causal effects of different modalities in the prompt on the performance of code generation LLMs. CodeSCM defines a Structural Causal Model(Pearl et al., 2000), shown in Figure 1, where each modal component of the prompt is treated as an independent variable that causally affects the code generated by the model. To account for similar natural language and code semantics of different surface forms, we introduce two latent mediator variables to capture code semantics and natural language semantics of the input prompt, mimicking a human mental model for generating correct code snippets from a multi-modal input problem.

Specifically, we make four key contributions in this paper: (i) we introduce CodeSCM, a novel framework for causal inference in multi-modal code generation tasks, enhancing interpretability and causal understanding of codeLLMs. While CodeSCM is designed for the code generation task in this paper, it can be extended to other modalities, tasks, and transformations. (ii) using CodeSCM, we define the Total Effects of the modalities on code generation, highlighting that input-output example pairs and natural language code components, like function headers, are significant modal components alongside natural language instructions. We also observe benchmark memorization in LLMs like GPT-4T with our Total Effect analysis; (iii)

*These authors contributed equally to this work.

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 6779–6793
April 29 - May 4, 2025 ©2025 Association for Computational Linguistics

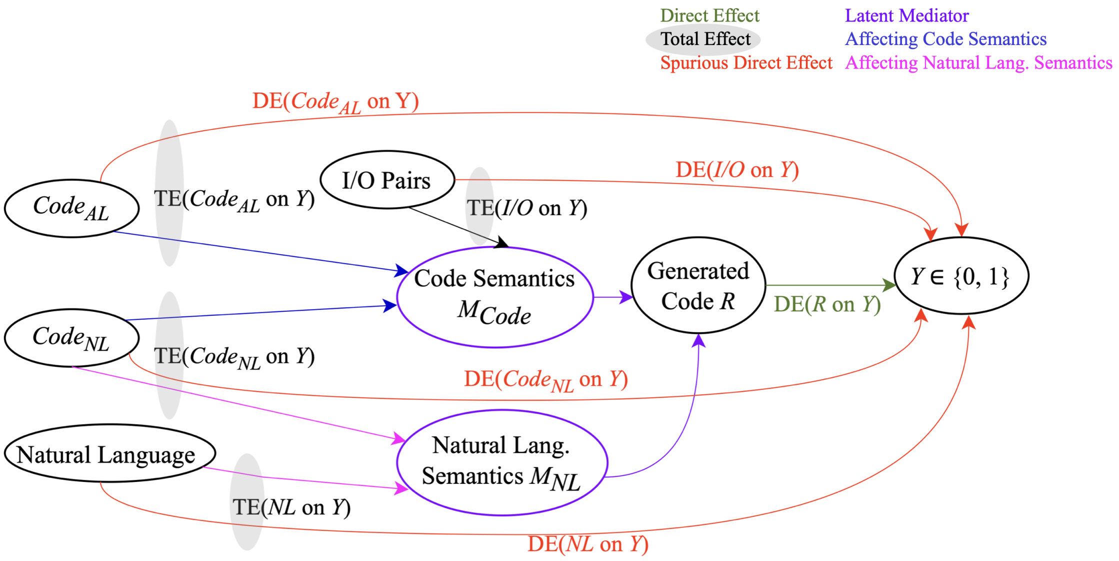


Figure 1: CodeSCM causal graph representing the total and direct effects of the modal variable nodes on the response variable Y representing the correctness of the generated code. $Code_{AL}$ represents algorithmic channel of code and $Code_{NL}$ is natural language channel of code.

through targeted interventions on CodeSCM, we measure the Direct Effects of each modality, representing spurious model correlations, and show that simple semantics-preserving transformations to input-output example pairs lead to a significant drop in accuracy; and (iv) following the asymmetric causal effects of modalities, we examine the effect of multi-modal code-specific pretraining on the embedding space, which shows that codeLLM CodeLLaMa can align different prompt modalities better than LLaMa-2 in the embedding space. Our code is available on GitHub[1].

2 Background

2.1 Structural Causal Model

A Structural Causal Model (SCM) $\mathcal{M}$ (Pearl et al., 2000) is defined by a 4-tuple $\mathcal{M} = \langle \mathbf{U}, \mathbf{V}, \mathbf{F}, P(\mathbf{U}) \rangle$, where $\mathbf{U}$ represents a set of exogenous variables that are affected by factors external to the model, $P(\mathbf{U})$ is a joint probability distribution over the set $\mathbf{U}$, $\mathbf{V}$ is a set of endogenous variables determined by variables in $\mathbf{U} \cup \mathbf{V}$, and $\mathbf{F}$ is a set of functions from $\mathbf{U} \cup \mathbf{V}$ to $\mathbf{V}$. The functions in $\mathbf{F}$ can range from simple indicator functions for binary variables to language models for complex NLP tasks. The SCM $\mathcal{M}$ can be represented by a causal graph $\mathcal{G}$, which employs nodes to represent both exogenous and endogenous variables.

Causal effects on any response variable $Y \in \mathbf{V}$ are quantitatively measured using interventions. An intervention on $X \in \mathbf{V}$, represented by $do(x)$, creates a sub-SCM $\mathcal{M}^x = \langle \mathbf{U}, \mathbf{V}, \mathbf{F}_x, P(\mathbf{U}) \rangle$ where $\mathbf{F}_x$ represents a subset of function mappings in $\mathbf{F}$ which do not have X in their co-domain, and X is replaced by a constant x. Hence, forcing the variable X to take a constant value and removing all mechanisms that may affect it. The response variable Y post-intervention ($do(x)$) is represented as Y_x, i.e., $P(Y_x) = P(Y|do(x))$. Following the above notation, we can formally define causal effects:

Definition 2.1. (Total Effect) (Pearl, 2000): The causal effect of two distinct realizations of variable X with $do(X = x')$ and $do(X = x'')$. Total Effect ($TE(x', x'')$) can be written as:

$$\mathbb{E}[Y|do(X = x')] - \mathbb{E}[Y|do(X = x'')] \quad (1)$$

Causal Mediation Analysis (Robins and Greenland, 1992; Pearl, 2022; Robins, 2003) involves understanding the effects of a mediator $M \in \mathbf{V}$ in explaining changes in Y. All the effects from X to Y where all $Z \in \mathbf{V}$, representing the parents of Y excluding X, remain fixed are called Direct Effects. We measure the direct effect of modalities to define spurious learnings that are not mediated by the latent mediators. We use the definition of Path Effect for systematic measurement of direct effects of a modality.

Definition 2.2. (Path Effect) (Avin et al., 2005; Wu et al., 2019): The causal effect of variable X along a path α can be represented in an edge sub-

[1] https://github.com/nb15/codeSCM-naacl25

graph $\mathcal{G}_\alpha$. Path Effect ($PE_\alpha(x', x'')$) can be written as:

$$\mathbb{E}\left[Y|Z_\alpha(do(x'')), Z_{\overline{\alpha}}(do(x'))\right] - \mathbb{E}\left[Y(x')\right] \quad (2)$$

where Z_α is the set of all mediators $\in \mathcal{G}_\alpha$, and $Z_{\overline{\alpha}}$ is the complementary mediator set. Hence, all the variables on the path α take values with $do(x'')$, and other mediators that do not lie on α take values with $do(x')$. Note that the Direct Effect is a special case of the Path Effect.

2.2 Multi-Modal Code Generation

For code generation tasks, natural language instructions alone are often insufficient to meet strict context-based syntax requirements, such as variable or function names that are dependent on surrounding code. Thus, natural language prompts are augmented with code modality, guiding the generation into appropriate syntactical space (Chen et al., 2021a; Austin et al., 2021). Additionally, some prompt components, such as function header name, appear as a single entity but contribute to multiple modalities in terms of model understanding, as observed by Casalnuovo et al. (2020).

For instance, the function header name in Figure 2a is primarily a code component, but its natural language name also conveys information about the desired output. As highlighted later in Section 3.1, we call this a natural language channel of Code. Figure 3 illustrates an example of a prompt with and without the natural language channel of Code. Similarly, input-output (I/O) example pairs also carry information about code correctness and logic beyond the syntactical structure. We believe that future codeLLMs might rely heavily on these components to ensure the correctness of intermediary variables, akin to code debugging process of longer code fragments. Therefore, we consider I/O pairs and natural language channels of code as separate modalities, in addition to natural language instructions and code.

3 Problem Setup

3.1 CodeSCM

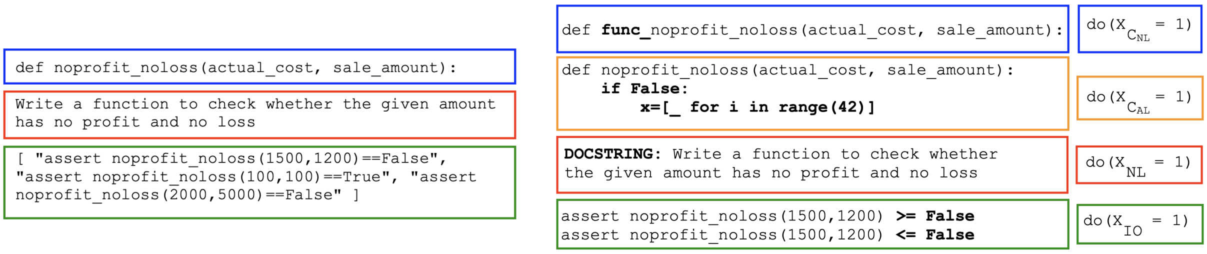
Each prompt $\mathcal{P}$ in dataset $\mathcal{D}$ is decomposed into its multi-modal components, which are represented as variables in the structural causal model, as shown in Figure 1. We use the extended Backus–Naur form (Equation 7) to represent the multi-modal prompt. In CodeSCM, as shown in Equation 7, we consider four modalities: Natural Language (NL), algorithmic channel of Code ($Code_{AL}$), natural language channel of Code ($Code_{NL}$), and input-output example pairs (I/O).

We define the multi-modal structural causal model (CodeSCM) to model the causal relationship between prompt modalities and the model-generated code. Since different code snippets and similar natural language texts can convey the same semantics for a human mental model, we introduce two latent mediators: M_{Code} for code semantics and M_{NL} for natural language semantics. Following the Causal Mediation Analysis, we assume each modality's effect on the output is mediated through these variables. As shown in Figure 1, $Code_{AL}$ and NL directly affect M_{Code} and M_{NL} respectively; I/O affects M_{Code}, and, $Code_{NL}$ directly affects both mediators. The output generated code, R, is tested for correctness against the test cases, where code correctness is the response variable $Y \in 0, 1$, with $\mathbb{E}(Y)$ representing accuracy over dataset $\mathcal{D}$. We do not account for confounding variables in this analysis, leaving this investigation for future work.

3.2 Modal Causal Effects

Using CodeSCM, we define the causal effects of each modal variable in $\mathcal{P}$ on the generated code. We measure the Total Effect (Definition 2.1) of each modality on the response variable Y, reflecting the model's sensitivity. Additionally, we examine the Direct Effect of modalities in form of Path Effect (Definition 2.2) on Y, along a path that bypass M_{Code} and M_{NL}, capturing spurious correlations learned during training. We also define additional variables and interventions to quantify these effects. Direct Effect (DE) and Total Effect (TE) for $Code_{AL}$ are presented here and the detailed derivations for other modalities are in Appendix A.

Causal effects of $\boldsymbol{Code_{AL}}$. We consider the $Code_{AL}$ variable as an output of a structural equation $F_C \in \mathbf{F}$ on three additional variables, C_{AL}, C_{DC} and X_{AL}, shown in Equation 3. C_{AL} is the actual prompt component $\mathcal{P}_{Code_{AL}} \in \mathcal{D}$. To measure $DE(Code_{AL}\ on\ Y)$, quantifying the spurious correlations, we vary $Code_{AL}$ variable while keeping mediator M_C constant i.e $M_C(Code_{AL}) = M_C(Code'_{AL})$. As shown in Figure 2b, we do this by inserting Dead Code (DC) into the original code, a code semantics-preserving transformation. The dead code is represented by variable C_{DC}. We use the categorical variable X_{AL} to represent the interaction between the actual code and the dead code,



```
def noprofit_noloss(actual_cost, sale_amount):
```

```
Write a function to check whether the given amount
has no profit and no loss
```

```
[ "assert noprofit_noloss(1500,1200)==False",
"assert noprofit_noloss(100,100)==True", "assert
noprofit_noloss(2000,5000)==False" ]
```

```
def func_noprofit_noloss(actual_cost, sale_amount):
```

do ($X_{C_{NL}}$ = 1)

```
def noprofit_noloss(actual_cost, sale_amount):
    if False:
        x=[_ for i in range(42)]
```

do ($X_{C_{AL}}$ = 1)

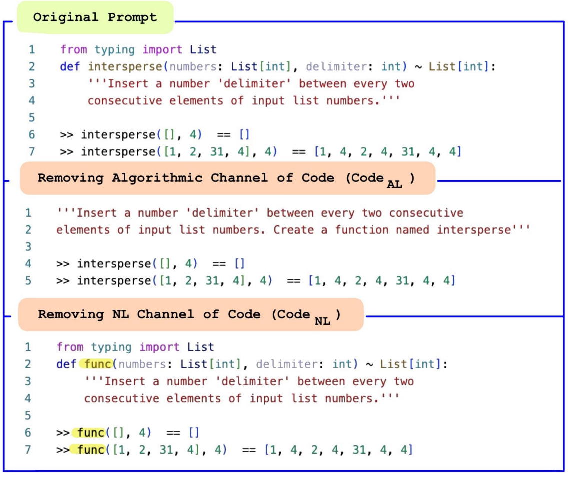
```
DOCSTRING: Write a function to check whether
the given amount has no profit and no loss
```

```
assert noprofit_noloss(1500,1200) >= False
assert noprofit_noloss(1500,1200) <= False
```

(a) Multi-Modal Prompt

(b) Semantics preserving transformations

Figure 2: (a) Modalities in an example from mMBPP+ dataset, red for NL, blue for $Code_{AL}$ and $Code_{NL}$, and green for Input/Output examples. (b) semantics preserving transformations: red for natural language, blue for $Code_{NL}$, orange for $Code_{AL}$, and green for I/O examples.



Original Prompt

```
from typing import List
def intersperse(numbers: List[int], delimiter: int) ~ List[int]:
    '''Insert a number 'delimiter' between every two
    consecutive elements of input list numbers.'''

>> intersperse([], 4)  == []
>> intersperse([1, 2, 31, 4], 4)  == [1, 4, 2, 4, 31, 4, 4]
```

Removing Algorithmic Channel of Code ($Code_{AL}$)

```
'''Insert a number 'delimiter' between every two consecutive
elements of input list numbers. Create a function named intersperse'''

>> intersperse([], 4)  == []
>> intersperse([1, 2, 31, 4], 4)  == [1, 4, 2, 4, 31, 4, 4]
```

Removing NL Channel of Code ($Code_{NL}$)

```
from typing import List
def func(numbers: List[int], delimiter: int) ~ List[int]:
    '''Insert a number 'delimiter' between every two
    consecutive elements of input list numbers.'''

>> func([], 4)  == []
>> func([1, 2, 31, 4], 4)  == [1, 4, 2, 4, 31, 4, 4]
```

Figure 3: The original HumanEval+ prompt (top) includes the function header `intersperse`, followed by natural language instructions and input-output pairs for code generation. The first modification (middle) removes the algorithmic code channel by eliminating all code components while retaining a natural language description of the function header in the docstring. The second modification (bottom) removes the natural language channel by standardizing the function name.

which also allows us to perform an intervention and calculate causal effects over all prompts in $\mathcal{D}$. We drop the AL subscript in the following derivation for brevity. It can take one of three values as shown in equation 3.

$$Code_{AL} \leftarrow \mathbb{1}_{\{X=1\}}(C_{AL} + C_{DC}) + \mathbb{1}_{\{X=0\}}(C_{AL}) + \mathbb{1}_{\{X=-1\}}(NULL) \quad (3)$$

where $\mathbb{1}(.)$ is an indicator function; $(C_{AL} + C_{DC})$ represents the concatenation of a snippet of dead code with the actual code.

We measure the TE of $Code_{AL}$, $TE(Code_{AL}\ on\ Y)$, by computing the expected change in the Y by setting the $Code_{AL}$ component as NULL in the prompt using the variable X_{AL}.

$$\begin{aligned} TE &= TE(do(X_{AL}=0), do(X_{AL}=-1)) \\ &\overset{(i)}{=} \mathbb{E}\left[Y_{X=0}\right] - \mathbb{E}\left[Y_{X=-1}\right] \\ &\overset{(ii)}{=} \mathbb{P}\left[Y_{X=0}=1\right] - \mathbb{P}\left[Y_{X=-1}=1\right] \\ &\overset{(iii)}{=} A(\mathcal{D}) - A(\mathcal{D}; \mathcal{P}_{Code_{AL}} = NULL) \end{aligned}$$

where, equality (i) follows from the Definition 2.1, equality (ii) follows because Y follows Bernoulli distribution; $A(\mathcal{D})$ is the accuracy of the model over the dataset $\mathcal{D}$. Figure 3 shows an example of how $Code_{AL}$ is removed from the prompt for computing the total effect.

The DE of $Code_{AL}$ on Y, $DE(Code_{AL}\ on\ Y)$ is measured by the expected change in Y with varying $Code_{AL}$ while keeping M_{Code} unchanged with dead code insertion. We calculate DE using the Path Effect of X_{AL} on Y, along a path from X_{AL} to Y which goes through $Code_{AL}$ but skips M_{Code}. We note that the quantification of direct effect is a special case path effect definition. Using Definition 2.2:

$$\begin{aligned} DE &= \mathbb{E}\left[Y_{Code_{AL}(X=1), M_C(X=0)}\right] - \mathbb{E}\left[Y_{X=0}\right] \\ &\overset{(i)}{=} \mathbb{E}\left[Y_{X=1}\right] - \mathbb{E}\left[Y_{X=0}\right] \\ &\overset{(ii)}{=} A(\mathcal{D}) - A(\mathcal{D}; \mathcal{P}_{Code_{AL}} = C_{AL} + C_{DC}) \end{aligned}$$

where equality (i) follows from the fact that $M_C(Code_{AL}(X = 0))$ is equal to $M_C(Code_{AL}(X = 1))$, because the dead code insertion in Equation 3 keeps the code semantics M_{Code} unchanged. Equality (ii) is similar to equalities (ii) and (iii) used in TE.

Causal Effects of Other Modalities. Similarly, the NL, I/O, and $Code_{NL}$ modal variables are considered as outputs of structural Equations 4, 5 and 6 respectively. For $Code_{NL}$, the direct effect requires bypassing two mediators, M_{NL} and M_{Code}. Therefore, we define a transformation that preserves semantics for both. As seen in Figure 2b, our transformation adds a prefix DN (Dead Name) to the function header, preserving semantics in both the natural language and code domains. For I/O transformations, each assertion equality is replaced by two inequalities ($\geq$ and $\leq$). While we demonstrate one specific transformation for each modality in our work to compute the respective Direct Effects, we note that CodeSCM can be extended to any other transformations, provided that i) the mediator variables remain unchanged, and ii) the transformations are independent of the input prompt. In addition to DE experiments in Section 4, we show DE computation with one additional transformation in Appendix D. We use simple prefix/suffix transformations to ensure independence between variables like S and DS or C and DC, to avoid correlation introduced by common transformations such as back-translation for NL.

4 Experiments

4.1 Settings

Datasets. To select evaluation datasets, we considered three key requirements: i) the dataset should contain code and natural language components (and preferably I/O pairs), iii) it should provide test cases to quantify code correctness, and iii) input modalities should be separable to isolate modal causal effects.

Based on these criteria, we study the causal effects on codeLLMs across three code generation benchmarks HumanEval+ (Liu et al., 2023a), mMBPP+ (Liu et al., 2023a), and CoderEval(Yu et al., 2024). To accommodate the lack of an explicit $Code_{AL}$ modality in the original MBPP+ dataset, we create mMBPP+ (multi-modal MBPP+) by adding a code function header to the original prompt. We evaluate HumanEval and mMBPP using evalplus (Liu et al., 2023a), which extends original datasets by incorporating additional challenging test cases for more rigorous testing. CoderEval offers a range of coding problems, from self-contained functions, to more complex functions that require an entire project environment to run. We focus on the self-contained (SC) subset, named CoderEval-SCP for Python and CoderEval-SCJ for Java. Detailed statistics are shown in Table 1.

Dataset	Size	NL	$Code_{AL}$	$Code_{NL}$	I/O Pairs
HumanEval+	164	✓	✓	✓	✓
MBPP+	399	✓	×	✓	✓
mMBPP+	399	✓	✓	✓	✓
CoderEval	460	✓	✓	✓	×
CoderEval-SCP	35	✓	✓	✓	×
CoderEval-SCJ	55	✓	✓	✓	×

Table 1: Statistics and prompt modalities of HumanEval+, MBPP+, mMBPP+, and CoderEval datasets.

Models. Using CodeSCM, we evaluate the causal effects on three codeLLMs: OpenAI GPT-4 Turbo (OpenAI, 2024) (updated on January 25, 2024), WizardCoder-15B (Luo et al., 2023), and Llama-3-8B (AI@Meta, 2024). To further explore the implications of modal alignment with code training, we examine the modal-representation space of CodeLLaMa-13B and LLaMa-2 13B to isolate the effects of multi-modal training, keeping other parts of the training process and model architecture constant.

Implementation. Following previous works on code generation (Chen et al., 2021b), we use the change in mean $pass@1$ accuracy ($Pr(Y = 1)$) to quantify the direct and total effects after interventions on CodeSCM. All datasets used are evaluation-only subsets, with no training involved in our experiments. For inference on all LLMs, we use a temperature of 0.01, a top_p value of 0.95, and a batch size of 8. The open-source model experiments were conducted on a single A100 GPU with 40 GB VRAM and each run took less than 2 GPU hours. During experiments with self-contained CoderEval functions in Python and Java, we ensured that the transformations were equivalent across both languages.

4.2 Total Effects of Modalities

Natural Language. The NL component, often a docstring in code completion tasks and containing the core logic of the generated code, shows the highest TE across all models on HumanEval+, mMBPP+, and CoderEval-SCP. As shown later in Section 4.5, removing NL increases the semantic errors. The TE of NL is highest for HumanEval+, followed by mMBPP+ and CoderEval-SCP, likely due to the greater detail in HumanEval+ docstrings compared to the shorter ones in CoderEval-SCP. However, the $do(X_{NL} = -1)$ intervention still maintains a non-zero accuracy. Given that generating correct code output without NL semantics should not be possible, we hypothesize that the

Model	Modality	HumanEval+		mMBPP+		CoderEval-SCP		CoderEval-SCJ		Mean	
		TE	DE	TE	DE	TE	DE	TE	DE	TE	DE
GPT-4T	Full	81.71		72.68		48.57		43.64		61.65	
	NL	**42.08**	1.22	19.05	4.26	**20.00**	**2.86***	3.64	1.82	21.19	3.64
	$Code_{AL}$	1.83	1.22	1.25	4.01	8.57	0.00	**43.64**	**18.18***	13.82	5.86
	$Code_{NL}$	18.91	1.83	**42.86**	2.76	0.00	**2.86***	1.82	0.00	15.90	1.52
	I/O Pairs	5.49	**2.44**	12.28	**6.26**	N/A	N/A	N/A	N/A	8.89	4.35
WizCoder	Full	53.05		52.63		37.14		47.27		47.52	
	NL	**30.49**	5.49	**13.53**	0.50	5.71	**8.57***	10.91	**3.64**	15.16	3.70
	$Code_{AL}$	4.27	9.76	2.00	**2.50**	2.86	2.86*	**45.45**	0.00	13.65	3.78
	$Code_{NL}$	6.10	2.44	4.01	0.50	**8.57***	**8.57***	3.64	0.00	5.58	3.34
	I/O Pairs	12.20	**12.20**	5.26	0.75	N/A	N/A	N/A	N/A	8.73	6.48
LLaMa-3	Full	55.49		58.64		31.43		0		36.39	
	NL	**33.54**	3.66	**16.54**	0.00	**11.43**	**5.71***	0.00	**3.64***	15.38	3.54
	$Code_{AL}$	0.61	3.66	1.76	1.51	0.00	2.86*	0.00	0.00	0.59	2.01
	$Code_{NL}$	10.98	3.05	6.02	2.01	8.57	0.00	0.00	0.00	6.39	0.98
	I/O Pairs	6.10	**4.27**	6.27	**2.76**	N/A	N/A	N/A	N/A	6.19	3.52

Table 2: Total Effect (TE) and Direct Effect (DE) of modalities on code generation. Pass@1 accuracy on Full prompt for each model and dataset is reported, followed by accuracy drop, indicating TE or DE. "*" denotes an increase in accuracy with the respective intervention. Bold highlights top TE and DE for each dataset and model. Accuracy results are averaged across 3 runs.

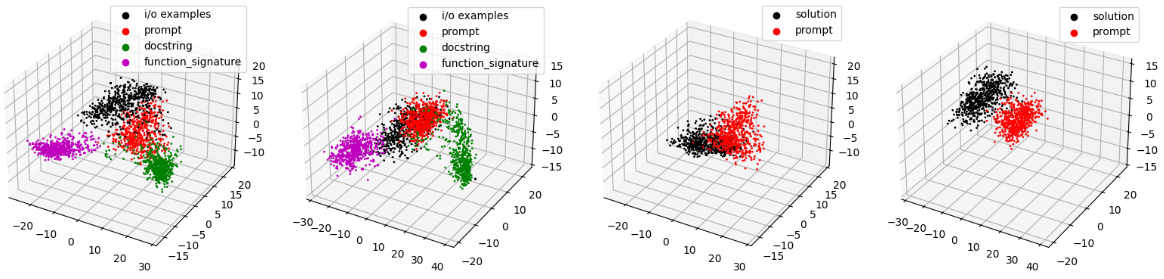
model either infers the correct NL semantics from $Code_{NL}$ or relies on its memory, suggesting memorization.

$Code_{NL}$ TE. The latter hypothesis is confirmed by the TE computation of $Code_{NL}$, which emerges as an important prompt component in the HumanEval+ and mMBPP+ datasets. For GPT-4T on mMBPP+, the TE of $Code_{NL}$ is 42.86%, surpassing the NL modality. Natural language chat models, such as GPT-4T and LLaMa-3, consistently show higher TE for $Code_{NL}$, with GPT-4T reaching 18.91% on HumanEval+. This suggests that natural language models may prioritize NL semantics (M_{NL}) more than code-focused models.

$Code_{AL}$ TE. In CoderEval-SCJ, $Code_{AL}$ has a high TE across all models, with GPT-4T and WizardCoder performance dropping nearly to zero. We observe limited code generation capabilities in the Java programming language exhibited by codeLLMs, particularly evident with zero performance from LLaMa-3. Further, models hallucinate code entry points when $Code_{AL}$ is absent. For instance, in all 55 examples, LLaMa-3 places the required code in a hallucinated class, as illustrated in Figure 5. On Python subsets, $Code_{AL}$, which contains minimal syntax information such as function headers and variable names, has the lowest TE across all models. However, $Code_{AL}$ in all three datasets under consideration is limited to the function header and input variable names along with function syntax (Figure 2a); the TE of $Code_{AL}$ where it may contain essential generation logic is yet to be explored.

I/O Pairs. The TE of I/O pairs surpasses that of $Code_{NL}$ with WizardCoder and holds equal significance with LLaMa-3 and GPT-4. This underscores the syntactic information encoded within I/O pairs, potentially aiding the model in reasoning over correct code structures. Analogously to human programmers employing unit testing for iterative code writing, the TE of I/O pairs suggests a similar process within codeLLMs. Future versions of codeLLMs may leverage intermediate I/O values for handling complex code, similar to the debugging process in software engineering.

Memorization of Code Benchmarks. Given that codeLLMs are trained on open-source datasets, we explore the potential for benchmark memorization. The non-zero $pass@1$ accuracy, even without NL instructions, indicates memorization. Furthermore, even after standardizing function header names, GPT-4T still generated original function names in 11.5% of HumanEval+ and 7.2% of mMBPP+ cases (Figure 5). LLaMa-3 showed similar behavior, with 10.3% of HumanEval+ examples despite the standardization of function names. The notably high memorization figures for GPT-4T also raise concerns regarding its performance on the EvalPlus leaderboard(Liu et al., 2023a). Similar to prior studies, such as (Lai et al., 2022), that examine memorization, our causal analysis also suggests substantial dataset memorization. However, a detailed investigation is left for future work.



(a) CodeLLaMa-13B (b) LLaMa-2 13B (c) CodeLLaMa-13B (d) LLaMa-2 13B

Figure 4: (a) and (b) Embedding PCA projections of modalities in input prompt by CodeLLaMa and LLaMa-2. (c) and (d) Prompt embedding projections along with the ground-truth code embedding projections by CodeLLaMa and LLaMa-2. $Code_{AL}$ and $Code_{NL}$ is combined into function_signature.

4.3 Direct Effects of Modalities

We define direct effects (DE) by noting the drop in $pass@1$ accuracy of the model under the semantics-preserving transformations of modalities where the latent mediators remain unchanged (Section 3.2). These effects also represent the spurious correlations, as any non-spurious learning process must be mediated through M_{NL} and M_{Code}. From Table 2, I/O pairs exhibit the strongest direct effect (DE) on HumanEval+ and mMBPP+ across models, except for mMBPP+ on WizardCoder. As seen in Figure 5, replacing a single assert equality in each I/O example with two inequalities makes it harder for the model to reason correctly over the code logic.

The DE of I/O pairs is then followed by the DE of $Code_{AL}$, where WizardCoder shows a very high DE of 9.76% on HumanEval+. For CoderEval-SCJ, GPT-4T's accuracy increased by 18.18% under the $do(X_{AL} = 1)$ intervention. As shown in Figure 5, a Java code snippet in the form of dead code reduces the class name hallucinated by the model. With this finding, we speculate that dead code might help control hallucinations, but we leave the detailed analysis to future work. In general, we observe that the DEs of NL and $Code_{NL}$ are comparatively lower, implying models are more robust to natural language than code semantics, likely due to instruction tuning stages.

4.4 Effect of Multi-Modal Pretraining

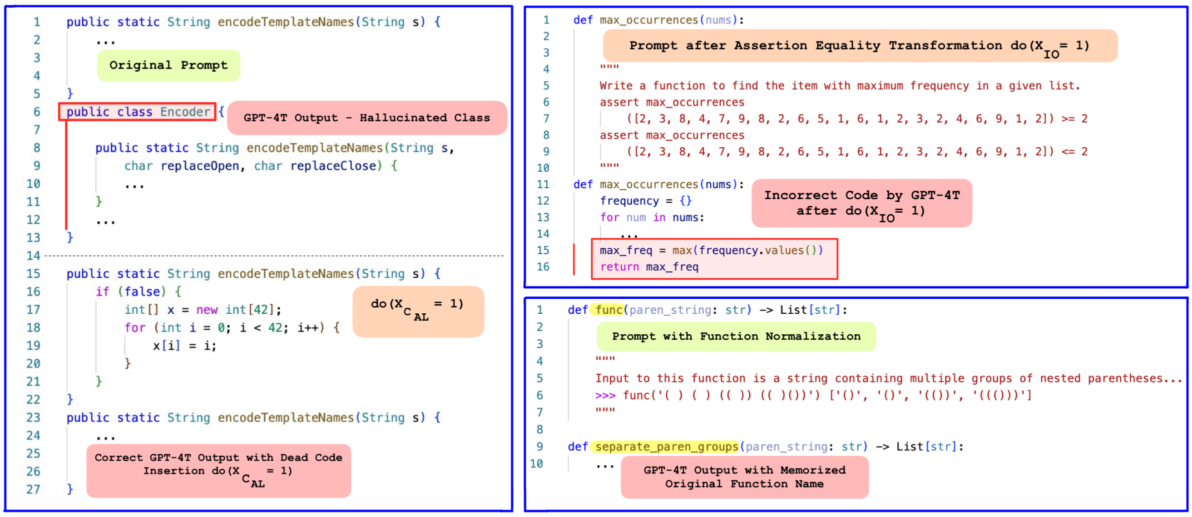
Our experiments on causal effects reveal asymmetric impacts of different modalities, leading us to examine their distribution in the embedding space of codeLLMs with code pretraining. We use PCA, following previous works (Cai et al., 2020; Rajaee and Pilehvar, 2021), to visualize high-dimensional representations into three dimensions. For this analysis, we combine data samples from the HumanEval and mMBPP datasets, excluding CoderEval due to its lack of the I/O modality (Table 1).

We use LLaMa-2(Touvron et al., 2023) to explore the effect of multi-modal pretraining of LLMs and how it affects the embedding representation of different modalities. The pretraining stages of code-aware LLMs add multi-modal alignment in codeLLMs, which is confirmed by the performance difference of 26.9% on HumanEval and 39.2% on MBPP between the LLaMa-2 (13B) and the codeLLM CodeLLaMa (13B), as reported by prior works (Li et al., 2023; Liu et al., 2023b). Using CodeLLaMa-13B and LLaMa-2 13B we can isolate the effects of multi-modal training keeping other factors such as model architecture and positional encodings constant.

In Figure 4b, complete prompts and modal components (red and green) form distinct clusters in the case of LLaMa-2, whereas, in Figure 4a, the prompt and the docstring are better associated by CodeLLaMa as they form closer clusters. I/O examples in the prompt have high token overlap with the function header (Figure 2a). While LLaMa-2 keeps them together (black and magenta clusters) probably due to the high token overlap, CodeLLaMa can separate them in the embedding space. Figures 4c and 4d show how the model associates input prompts and ground-truth code solutions. LLaMa-2 forms nearly two disjoint clusters (Figure 4c), even when prompts and ground-truth code are strongly correlated, while CodeLLaMa can associate prompt with code solution in the embedding space. We further discuss anisotropy in CodeLLaMa's embedding spaces in Appendix 5.

		HumanEval+				mMBPP+			
Model	Modality	Syn	Sem	Runt	Other	Syn	Sem	Runt	Other
GPT-4T	NL	-8.22	26.13	-18.67	0.76	-5.58	14.32	-9.61	0.87
	$Code_{AL}$	-0.75	1.49	-1.80	1.10	-1.42	2.31	-0.89	0.00
	$Code_{NL}$	-3.12	-7.04	-8.47	18.63	-11.27	-30.64	32.21	9.70
	I/O Pairs	-1.62	0.61	1.01	0.00	-3.85	-4.75	7.65	0.94
WizCoder	NL	-3.37	15.01	-11.65	0.00	-1.29	7.72	-6.01	-0.42
	$Code_{AL}$	-0.30	-0.65	-0.95	-.00	-0.92	6.93	-5.58	-0.42
	$Code_{NL}$	-0.59	-4.58	-0.21	4.96	-1.04	-2.08	2.74	0.39
	I/O Pairs	-1.01	2.67	-1.65	0.0	-1.05	4.06	-3.38	0.37
LLaMa-3	NL	-4.11	20.96	-16.60	-0.25	-5.14	14.13	-7.98	-1.01
	$Code_{AL}$	0.42	0.90	-1.30	-0.03	-1.32	4.38	-2.13	0.93
	$Code_{NL}$	-1.57	-8.67	-1.21	11.44	-0.90	-10.04	7.68	3.26
	I/O Pairs	4.43	1.75	-6.13	-0.05	-0.41	-9.85	10.80	-0.54

Table 3: Percentage of errors out of total passed cases for GPT-4T, WizardCoder-15B, and Llama-3-8B on HumanEval+ and mMBPP+. Negative percentages indicate a decrease in error count, while positive values indicate an increase in error count upon intervention. Syn represents Syntax errors, Sem represents Semantic Errors, and RunT represents runtime errors.


```
1   public static String encodeTemplateNames(String s) {
2       ...
3
4
5   }
6   public class Encoder {
7
8       public static String encodeTemplateNames(String s,
9           char replaceOpen, char replaceClose) {
10          ...
11      }
12      ...
13  }
14 ------------------------------------------------------------
15  public static String encodeTemplateNames(String s) {
16      if (false) {
17          int[] x = new int[42];
18          for (int i = 0; i < 42; i++) {
19              x[i] = i;
20          }
21      }
22  }
23  public static String encodeTemplateNames(String s) {
24      ...
25
26
27  }
```

```
1   def max_occurrences(nums):
2
3
4       """
5       Write a function to find the item with maximum frequency in a given list.
6       assert max_occurrences
7           ([2, 3, 8, 4, 7, 9, 8, 2, 6, 5, 1, 6, 1, 2, 3, 2, 4, 6, 9, 1, 2]) >= 2
8       assert max_occurrences
9           ([2, 3, 8, 4, 7, 9, 8, 2, 6, 5, 1, 6, 1, 2, 3, 2, 4, 6, 9, 1, 2]) <= 2
10      """
11  def max_occurrences(nums):
12      frequency = {}
13      for num in nums:
14          ...
15      max_freq = max(frequency.values())
16      return max_freq
```

```
1   def func(paren_string: str) -> List[str]:
2
3
4       """
5       Input to this function is a string containing multiple groups of nested parentheses...
6       >>> func('( ) ( ) (( )) (( )())') ['()', '()', '(())', '((()))']
7       """
8
9   def separate_paren_groups(paren_string: str) -> List[str]:
10      ...
```
Original Prompt
GPT-4T Output - Hallucinated Class
do(X_{C_{AL}} = 1)
Correct GPT-4T Output with Dead Code Insertion do(X_{C_{AL}} = 1)
Prompt after Assertion Equality Transformation do(X_{IO}= 1)
Incorrect Code by GPT-4T after do(X_{IO}= 1)
Prompt with Function Normalization
GPT-4T Output with Memorized Original Function Name

Figure 5: Left figure shows a CoderEval-SCJ prompt where dead code insertion corrects the original prompt's error of creating a hallucinated Java class (red box). The top right figure illustrates an mMBPP+ prompt where I/O pair transformations lead to a semantic error in lines 15-16. The bottom right figure shows GPT-4T's memorization of a HumanEval+ prompt.

4.5 Errors in codeLLMs

In this section, we analyze the types of errors encountered by codeLLMs on the HumanEval+ and mMBPP+ datasets, as detailed in Table 3. Errors are categorized into syntax errors, semantic errors, runtime errors, and other errors. The values in the table represent the percentage change in error counts upon removal of the modality ($do(X = -1)$) during TE computation, relative to the full prompt. For instance, upon removal of NL from HumanEval for GPT-4T leads to a drop in accuracy of 42.08% (Table 2), and the 26.13% of this drop is due to increase in semantic errors (Table 3). So the percentage changes sum up to zero for each modality, model, and dataset combination. As evident for all three models, removing the NL modality leads to a significant increase in semantic errors, confirming that natural language instructions are crucial for conveying problem semantics. Semantic errors are seen in form of failed test cases or assertion statements. Other errors such as resource, dependency, environment and timeout errors are mostly seen with the removal of $Code_{NL}$ modality, reflecting its importance in guiding the correct code syntax. WizardCoder-15B shows relatively small changes in syntax and other errors across modalities. For instance, on HumanEval+, syntax errors change by -0.30% to -3.37% across different modalities, indicating strong syntactic generation capabilities, likely due to its extensive code-specific training.

5 Related Work

Automatic Code Generation. Code generation with multi-modal prompts has been explored by some earlier works such as (Desai et al., 2015; Gulwani et al., 2017). Recent works have either adopted the transformer architecture (Feng et al., 2020; Wang et al., 2021) or leveraged the GPT (Brown et al., 2020) skeleton with massive pretraining for code pretraining (Rozière et al., 2023; Li et al., 2023; Luo et al., 2023; Nijkamp et al., 2023; Zhu et al., 2024).

Prompt-Tuning. Various approaches to prompt-tuning (White et al., 2023) have been explored for various domains and modalities (Mullick et al., 2024; Wu et al., 2023), such as Chain-of-Thought reasoning (Wei et al., 2023), Tree of Thoughts (Yao et al., 2023), discrete prompt optimization (Wen et al., 2023; Shin et al., 2020), and few-shot learning (Brown et al., 2020). In the context of code generation, prompt engineering has been leveraged for human-in-loop debugging (Denny et al., 2022), correctness evaluation of generated code (Liu et al., 2023b), multistep planning, and generation (Zheng et al., 2023). Our work explores the effects of modalities in prompts on code generation, which can be further used for targeted prompt-tuning processes for better performance.

Causal Inference in Code/NLP. Recent research has applied causal inference to the NLP domain (Vig et al., 2020; Finlayson et al., 2021; Stolfo et al., 2022) to better understand model behavior, which is now formalized as causal NLP (Jin et al., 2022; Feder et al., 2022). In the context of code, prior approaches have applied causal framework for various classification tasks such as vulnerability detection (Rahman et al., 2024; He et al., 2022) and code performance prediction (Cito et al., 2021). To the best of our knowledge, we are the first to apply causal inference to study modal effects on code generation task.

6 Conclusion

We propose CodeSCM, a Structural Causal Model for analyzing multi-modal code generation using LLMs. Our analysis revealed that input-output examples and natural language code components significantly influence model generation. Additionally, our interventions show that semantics-preserving changes can impact accuracy and can also lead to fewer hallucinations in some cases.

7 Limitations

We can calculate causal effects in CodeSCM with the assumption of no confounders. We believe that in the future, our causal formulation of code generation could be extended to account for confounders using the backdoor criterion.

8 Ethical considerations

We release the algorithmic details and work with public code datasets, which neither reveal any personal sensitive information nor contain any toxic statements. While there are potential societal consequences, they are not deemed important to highlight here.

9 Acknowledgments

We thank the anonymous reviewers for their constructive and valuable feedback. This work is supported partially by an NSF CAREER award; and an award from the Google Cyber NYC Institutional program. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of NSF, or Google.

References

AI@Meta. 2024. Llama 3 model card.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Chen Avin, Ilya Shpitser, and Judea Pearl. 2005. Identifiability of path-specific effects.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Xingyu Cai, Jiaji Huang, Yuchen Bian, and Kenneth Church. 2020. Isotropy in the contextual embedding space: Clusters and manifolds. In *International Conference on Learning Representations*.

Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A theory of dual channel constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 25–28, New York, NY, USA. Association for Computing Machinery.

Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J. Pappas, and Eric Wong. 2023. Jailbreaking black box large language models in twenty queries. *Preprint*, arXiv:2310.08419.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. 2021. Counterfactual explanations for models of code. *Preprint*, arXiv:2111.05711.

Paul Denny, Viraj Kumar, and Nasser Giacaman. 2022. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. *Preprint*, arXiv:2210.15157.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2015. Program synthesis using natural language. *Preprint*, arXiv:1509.00413.

Amir Feder, Katherine A. Keith, Emaad Manzoor, Reid Pryzant, Dhanya Sridhar, Zach Wood-Doughty, Jacob Eisenstein, Justin Grimmer, Roi Reichart, Margaret E. Roberts, Brandon M. Stewart, Victor Veitch, and Diyi Yang. 2022. Causal inference in natural language processing: Estimation, prediction, interpretation and beyond. *Preprint*, arXiv:2109.00725.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *Preprint*, arXiv:2002.08155.

Matthew Finlayson, Aaron Mueller, Sebastian Gehrmann, Stuart Shieber, Tal Linzen, and Yonatan Belinkov. 2021. Causal analysis of syntactic agreement mechanisms in neural language models. *arXiv preprint arXiv:2106.06087*.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

Jingzhu He, Yuhang Lin, Xiaohui Gu, Chin-Chia Michael Yeh, and Zhongfang Zhuang. 2022. Perfsig: Extracting performance bug signatures via multi-modality causal analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1669–1680.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Zhijing Jin, Amir Feder, and Kun Zhang. 2022. CausalNLP tutorial: An introduction to causality for natural language processing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, pages 17–22, Abu Dubai, UAE. Association for Computational Linguistics.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Preprint*, arXiv:2305.01210.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pretrain, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Preprint*, arXiv:2107.13586.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *Preprint*, arXiv:2102.04664.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Ankan Mullick, Mukur Gupta, and Pawan Goyal. 2024. Intent detection and entity extraction from biomedical literature. *Preprint*, arXiv:2404.03598.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *Preprint*, arXiv:2203.13474.

OpenAI. 2024. Gpt-4 turbo. `https://help.openai.com/en/articles/8555510-gpt-4-turbo`.

Judea Pearl. 2000. *Causality: Models, Reasoning, and Inference*. Cambridge University Press.

Judea Pearl. 2022. Direct and indirect effects. In *Probabilistic and causal inference: the works of Judea Pearl*, pages 373–392.

Judea Pearl et al. 2000. Models, reasoning and inference. *Cambridge, UK: CambridgeUniversityPress*, 19(2):3.

Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards causal deep learning for vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–11.

Sara Rajaee and Mohammad Taher Pilehvar. 2021. An isotropy analysis in the multilingual bert embedding space. *arXiv preprint arXiv:2110.04504*.

James M Robins. 2003. Semantics of causal dag models and the identification of direct and indirect effects. *Highly structured stochastic systems*, pages 70–82.

James M Robins and Sander Greenland. 1992. Identifiability and exchangeability for direct and indirect effects. *Epidemiology*, 3(2):143–155.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2023. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *Preprint*, arXiv:2310.11324.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV au2, Eric Wallace, and Sameer Singh. 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *Preprint*, arXiv:2010.15980.

Alessandro Stolfo, Zhijing Jin, Kumar Shridhar, Bernhard Schölkopf, and Mrinmaya Sachan. 2022. A causal framework to quantify the robustness of mathematical reasoning with language models. *arXiv preprint arXiv:2210.12023*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Jesse Vig, Sebastian Gehrmann, Yonatan Belinkov, Sharon Qian, Daniel Nevo, Yaron Singer, and Stuart Shieber. 2020. Investigating gender bias in language models using causal mediation analysis. *Advances in neural information processing systems*, 33:12388–12401.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *Preprint*, arXiv:2109.00859.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Yuxin Wen, Neel Jain, John Kirchenbauer, Micah Goldblum, Jonas Geiping, and Tom Goldstein. 2023. Hard prompts made easy: Gradient-based discrete optimization for prompt tuning and discovery. *Preprint*, arXiv:2302.03668.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *Preprint*, arXiv:2302.11382.

Qi Wu, Yuyao Zhang, and Marawan Elbatel. 2023. Self-prompting large vision models for few-shot medical image segmentation. In *MICCAI workshop on domain adaptation and representation transfer*, pages 156–167. Springer.

Yongkai Wu, Lu Zhang, Xintao Wu, and Hanghang Tong. 2019. Pc-fairness: A unified framework for measuring causality-based fairness. *Preprint*, arXiv:1910.12586.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Preprint*, arXiv:2305.10601.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Wenqing Zheng, S P Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. 2023. Outline, then details: Syntactically guided coarse-to-fine code generation. *Preprint*, arXiv:2305.00909.

Kaijie Zhu, Jindong Wang, Jiaheng Zhou, Zichen Wang, Hao Chen, Yidong Wang, Linyi Yang, Wei Ye, Yue Zhang, Neil Zhenqiang Gong, and Xing Xie. 2023. Promptbench: Towards evaluating the robustness of large language models on adversarial prompts. *Preprint*, arXiv:2306.04528.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Causal Effects

In this section, we present the causal effects of three modalities: Natural Language (NL), I/O Pairs, and Code with NL component ($Code_{NL}$). For each modality, we provide the corresponding structural equation, followed by the total effect (TE) and direct effect (DE).

A.1 Natural Language (NL)

The NL variable is defined by the following structural equation:

$$NL \leftarrow \mathbb{1}_{\{X_{NL}=1\}}(S + DS) + \mathbb{1}_{\{X_{NL}=0\}}(S) + \mathbb{1}_{\{X_{NL}=-1\}}(NULL) \quad (4)$$

where S is the actual natural language prompt component $\mathcal{P}_S \in \mathcal{D}$, DS is a Dead String that does not alter the semantics of the natural language, and X_{NL} is used to control whether to allow the original $\mathcal{P}_S$, concatenate a dead string, or remove the natural language modality. Similar to dead code insertion in $Code_{AL}$, dead string insertion is a semantics-preserving transformation such that $M_{NL}(S) = M_{NL}(S + DS)$. NL subscript is dropped for brevity.

Total Effect of NL.

$$\begin{aligned} TE &= TE(do(X_{NL} = 0), do(X_{NL} = -1)) \\ &= Acc(\mathcal{D}) - Acc(\mathcal{D}; \mathcal{P}_{NL} = NULL) \end{aligned}$$

Direct Effect of NL.

$$\begin{aligned} DE &= \mathbb{E}\left[Y_{X=1,NL(X=1),M_{NL}\left(NL(X=0)\right)}\right] \\ &\quad - \mathbb{E}\left[Y_{NL(X=0)}\right] \\ &= Acc(\mathcal{D}) - Acc(\mathcal{D}; \mathcal{P}_{NL} = S + DS) \end{aligned}$$

Here, DS represents the Dead String. We use the prefix 'DOCSTRING: ' concatenated to each natural language instruction to preserve semantics. Other transformations such as back-translation are possible but introduce correlations between variables, so we prefer simpler prefix or suffix transformations that keep S and DS independent.

A.2 I/O Pairs

The I/O modality is defined by the following structural equation:

$$I/O \leftarrow \mathbb{1}_{\{X_{IO}=0\}}(I^r = I^r) + \mathbb{1}_{\{X_{IO}=1\}}((I^l \leq I^r) + (I^r \geq I^r)) + \mathbb{1}_{\{X_{IO}=-1\}}(NULL) \quad (5)$$

where I^l and I^r represent the left-hand side (LHS) and right-hand side (RHS) of the assertion equality statement in the original prompt, respectively. For semantics-preserving transformations, we replace each assertion equality with two inequalities, $\leq$ and $\geq$. I/O is omitted for brevity.

Total Effect of I/O.

$$\begin{aligned} TE &= TE(do(X_{IO} = 0), do(X_{IO} = -1)) \\ &= Acc(\mathcal{D}) - Acc(\mathcal{D}; \mathcal{P}_{IO} = NULL) \end{aligned}$$

Direct Effect of I/O.

$$\begin{aligned} DE &= \mathbb{E}\left[Y_{\left(X=1,M_{Code}(X=1),M_{Code}(X=0)\right)}\right] \\ &\quad - \mathbb{E}\left[Y_{M_{Code}(X=0)}\right] \\ &= Acc(\mathcal{D}) - \\ &\quad Acc(\mathcal{D}; \mathcal{P}_{IO} = (I^l \leq I^r) + (I^r \geq I^r)) \end{aligned}$$

A.3 Code with NL Component ($Code_{NL}$)

The $Code_{NL}$ modality is defined by the following structural equation:

$$Code_{NL} \leftarrow \mathbb{1}_{\{X_{NL}=1\}}(C_{NL} + DN) + \mathbb{1}_{\{X_{NL}=0\}}(C_{NL}) + \mathbb{1}_{\{X_{NL}=-1\}}(NULL) \quad (6)$$

where C_{NL} is the code prompt component $\mathcal{P}_{Code_{NL}} \in \mathcal{D}$, and DN is a Dead Name added to the function header. This transformation preserves semantics for both the natural language and code domains. For instance, $M_{NL}(C_{NL}) = M_{NL}(C_{NL} + DN)$.

Total Effect of $Code_{NL}$.

$$\begin{aligned} TE &= TE(do(X_{CN} = 0), do(X_{CN} = -1)) \\ &= Acc(\mathcal{D}) - Acc(\mathcal{D}; \mathcal{P}_{Code_{NL}} = NULL) \end{aligned}$$

Direct Effect of $Code_{NL}$.

$$\begin{aligned} DE &= \mathbb{E}\left[Y_{\left(X=1,C_{NL}(X=1),M_{NL}\left(C_{NL}(X=0)\right),\right.}\right. \\ &\quad \left.{}_{\left.M_C\left(C_{NL}(X=0)\right)\right)}\right] - \mathbb{E}\left[Y_{(C_{NL}(X=0)}\right] \\ &= Acc(\mathcal{D}) - Acc(\mathcal{D}; \mathcal{P}_{C_{NL}} = C_{NL} + DN) \end{aligned}$$

Here, DN represents Dead Name, and we use the prefix 'func_' in Python and 'Method' in Java to maintain semantic preservation. Other transformations, like capitalization, are possible but avoided to keep C_{NL} and DN independent.

B Multi-Modal Prompt

The multi-modal prompt $\mathcal{P}$ can be expressed as an equation comprising one or more prompt components P^j of modality M_i, where different prompt components are concatenated using one of the defined separators:

$$\mathcal{P} = P^1_{M_1}\left[\text{sep}\ \ P^j_{M_i}\right] \tag{7}$$

$$\text{sep} = \ '\ '\ \mid \backslash n \mid \backslash t \mid \ :\ \mid \ ,\ \mid ; \tag{8}$$

In this equation, different prompt components are concatenated using one of the defined separators.

C Implementation Details

We exclude APPS (Hendrycks et al., 2021) and CodeContest (Li et al., 2022), as they lack multi-modal prompts, making them unnecessary for multi-modal causal analysis. Similarly, while the CONCODE segment of the CodexCGLUE (Lu et al., 2021) benchmark includes multi-modal prompts, it measures code quality via natural language similarity metrics like BLEU, which is unsuitable for code generation tasks. Lastly, DS-1000 (Lai et al., 2022) was excluded due to the need for manual screening of all examples to separate modal components for CodeSCM.

D DE Additional Transformation

We demonstrate one specific transformation for each modality in the paper and compute the respective causal effects. CodeSCM can be directly extended to other transformations as well for DE computation. For example, in Table 4, along with original transformations from Table 2 (DE-1), we illustrate DE computation with an additional set of transformations (DE-2) for the mMBPP+ dataset using WizardCoder codeLLM. The following transformations are used for DE-2 - (dead string prefix, unused variable, dead name prefix, and negating the not assert statement):

- DS = Code Logic:\n (in Equation 4)
- C_{DC} = \tvar = 42 (in Equation 3)
- DN = header_ (in Equation 6)
- assert $I^l == I^r$ is changed to assert not $I^l != I^r$) (in Equation 5)

Modality	DE-1	DE-2
Full	52.63	
NL	0.50	1.23
$Code_{AL}$	2.50	3.03
$Code_{NL}$	0.50	1.73
I/O Pairs	0.75	3.23

Table 4: Direct effects of WizardCoder on mMBPP+ dataset under an additional transformation. DE-1 values are the same as Table 2

Prompt Component	CodeLLaMa ↓	LLaMa-2
examples	0.85	0.77
docstring	0.86	0.83
prompt	0.87	0.80
solution	0.90	0.83
function	0.91	0.85
all	0.77	0.66

Table 5: Intra-modal cosine similarity between mean hidden representation of CodeLLaMa-13B and LLaMa-2. Similarities are reported by combining HumanEval and mMBPP.

Modality-1	Modality-2	CodeLLaMa ↓	LLaMa-2
function	docstring	0.59	0.43
docstring	examples	0.63	0.45
solution	docstring	0.65	0.47
function	prompt	0.74	0.66
docstring	prompt	0.76	0.64
function	examples	0.77	0.66
examples	prompt	0.82	0.72
solution	function	0.82	**0.76**
solution	examples	0.83	0.72
solution	prompt	0.84	0.72

Table 6: Inter-modal cosine similarity between averaged hidden representation of CodeLLaMa-13B and LLaMa-2. Similarities are reported by combining HumanEval and mMBPP.

E Multi-Modal Pretraining

Inspired from previous works (Cai et al., 2020; Rajaee and Pilehvar, 2021), we measure the cosine similarities between the averaged last layer's hidden state representations of CodeLLaMa and LLaMa-2. For each modality, intra-modality S_{intra} cosine similarity is defined as $\mathbb{E}_{i,j \in P}\left[cos\left(M(i), M(j)\right)\right]$, where i and j are distinct prompts of same modality. Inter-modal cosine similarity S_{inter} is defined for a pair for modalities, as $\mathbb{E}_{i \in P1, j \in P2}\left[cos\left(M(i), M(j)\right)\right]$, where i and j belong to different prompt modal components.

Similar to Section 4.4, we combine data samples from the HumanEval and mMBPP datasets, excluding CoderEval due to its lack of I/O modality (Table 1). In Table 6, the ground truth problem solution and input prompt are kept closest by CodeL-

LaMa despite being of different modalities and low token overlap, which explains CodeLLaMa's superior performance on code generation benchmarks. LLaMa-2 on the other hand, keeps ground truth problem solution and function header name, probably due to significant token overlap between the two.

Measuring S_{inter} for each modality, in Table 6 we observe the closer clusters of modalities in CodeLLaMa's vector space with consistently higher similarities. Given a round of code pretraining, CodeLLaMa assigns the highest S_{inter} to the function header and solution, both of which are code components, while LLaMa-2 assigns similar S_{inter} to the docstring (P_{NL}) and code solution (P_{code}). Finally, in the last row of Table 6, we show the average cosine similarity of the entire space i.e., vector representations from all components. We note a higher similarity in the code model. Given elevated values of similarity for the code model, we suspect an anisotropic embedding space compared to the natural language model. Anisotropy would increase as the model learns to specialize in one task/domain (code generation in this case) and loses generalization capabilities. The concrete conclusive claim however requires further analysis which we leave to the future works.

F Error Analysis

In this section, we provide definitions of the different types of errors encountered in code generation tasks.

Syntax Errors. These errors occur when the code does not conform to the syntactical rules of the programming language. They are typically detected during the parsing stage. An example of a syntax error might be a missing colon, unmatched parentheses, or incorrect indentation.

Semantic Errors. Semantic errors arise when the code is syntactically correct but fails to produce the intended output due to logical mistakes. This can include errors in the logic of the code, incorrect use of variables, or wrong implementation of algorithms. We broadly encounter two types of semantic errors: (i) **test case errors**, when the test cases in the respective dataset fail; (ii) **assertion errors**, when an input-output example assertion in the prompt fails.

Runtime Errors. These errors occur during the execution of the code. They result from operations like division by zero, accessing out-of-bound indices, or other exceptional conditions that the code does not handle.

Other Errors. This category includes various errors that do not fit into the above classifications. It covers resource errors (e.g., memory errors when the program tries to allocate more memory than what is available), dependency errors (e.g., missing modules or packages), environment errors (e.g., issues with file access or permissions), and timeout errors (when the execution of the code takes longer than the allowed time limit).