DeciX: Explain Deep Learning Based Code Generation Applications

SIMIN CHEN, University of Texas at Dallas, USA
ZEXIN LI, University of California at Riverside, USA
WEI YANG, University of Texas at Dallas, USA
CONG LIU, University of California at Riverside, USA

Deep learning-based code generation (DL-CG) applications have shown great potential for assisting developers in programming with human-competitive accuracy. However, lacking transparency in such applications due to the uninterpretable nature of deep learning models makes the automatically generated programs untrustworthy. In this paper, we develop `DeciX`, a first explanation method dedicated to DL-CG applications. `DeciX` is motivated by observing two unique properties of DL-CG applications: output-to-output dependencies and irrelevant value and semantic space. These properties violate the fundamental assumptions made in existing explainable DL techniques and thus cause applying existing techniques to DL-CG applications rather pessimistic and even incorrect. `DeciX` addresses these two limitations by constructing a causal inference dependency graph, containing a novel method leveraging causal inference that can accurately quantify the contribution of each dependency edge in the graph to the end prediction result. Proved by extensive experiments assessing popular, widely-used DL-CG applications and several baseline methods, `DeciX` is able to achieve significantly better performance compared to state-of-the-art in terms of several critical performance metrics, including correctness, succinctness, stability, and overhead. Furthermore, `DeciX` can be applied to practical scenarios since it does not require any knowledge of the DL-CG model under explanation. We have also conducted case studies that demonstrate the applicability of `DeciX` in practice.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Computing methodologies** → **Machine learning**; **Artificial intelligence**;

Additional Key Words and Phrases: Program Synthesis, Large Language Model, Explainable AI

ACM Reference Format:
Simin Chen, Zexin Li, Wei Yang, and Cong Liu. 2024. DeciX: Explain Deep Learning Based Code Generation Applications. *Proc. ACM Softw. Eng.* 1, FSE, Article 107 (July 2024), 23 pages. https://doi.org/10.1145/3660814

1 INTRODUCTION

Deep learning (DL) has achieved phenomenal success in code generation applications (*i.e.,* program synthesis, code completion, API recommendation) [6, 7, 12, 16, 17, 24, 27, 35, 36, 49, 51]. DL-CG applications are booming due to high, even human-competitive accuracy. However, the complexity and opacity of underlying DNNs make it difficult for developers to confidently trust the application or effectively troubleshoot errors. Lacking explainability causes adopting DL-CG tools a major concern in many practical settings [37, 60, 68], particularly under safety-sensitive scenarios [2, 5, 18, 32, 47]. A recent empirical study [46] surprisingly shows that approximately 40% of the generated

Authors' addresses: Simin Chen, University of Texas at Dallas, Dallas, USA, sxc180080@utdallas.edu; Zexin Li, University of California at Riverside, Riverside, USA, zli536@ucr.edu; Wei Yang, University of Texas at Dallas, Dallas, USA, Wei.Yang@utdallas.edu; Cong Liu, University of California at Riverside, Riverside, USA, congl@ucr.edu.

ACM 2994-970X/2024/7-ART107
https://doi.org/10.1145/3660814

programs from Microsoft Copilot [37] are vulnerable. Clearly, being able to explain DL-driven behaviors is critical in making DL-CG applications be widely and safely adopted.

Recently, addressing the issue of lacking explainability in DL-driven software domains has received much attention. A rich set of explainable DL techniques have been proposed [5, 8, 9, 18, 29, 32, 52]. To assist users in understanding why a DL model makes a specific prediction, explainable DL techniques seek to identify a subset of input features (tokens) that play a dominant role in the decision-making process. By carefully examining existing explainable DL techniques [5, 8, 9, 18, 52], we observe that they are not applicable and cannot be trivially extended to DL-CG applications due to two fundamental limitations: *(i)* **Dependency-oblivious to output tokens:** existing explanation techniques mainly focus on the classification task, whose outputs are scalars with fixed dimensions [18, 52]. However, the theoretical foundation of DL-CG applications is language models (LM) which essentially exhibit a Markov process containing dependency among a discrete set of output tokens [64]. Existing explainable DL techniques are dependency-oblivious to output tokens since they cannot model the dependency among output tokens (detailed in §3 with an intuitive example shown in Fig. 2). Thus, being dependency-aware to output tokens is critical for any explainable DL techniques to be correct and effective for DL-CG applications. *(ii)* **Input feature (token) relevance in the value space**: an important assumption made in existing explainable DL techniques [18, 52] is semantic continuity in input features (*i.e.,* semantic-similar features will have similar values), which is true for DL models that accept images as inputs. For example, color 'white' (255, 255, 255) and 'ivory white' (255, 255, 240) are semantic-similar and have similar RGB pixel values. While this assumption works for most image classification tasks, it no longer holds for DL-CG applications. Inputs of DL-CG applications are usually discrete tokens, whose representations in the value space may be totally irrelevant (*e.g.,* while `postRequest` and `setRequest` are semantically relevant in programming languages, their tokenized values under a given DL model could be 31 and 203, respectively). Thus, existing techniques for explaining image classification models are not applicable. For instance, `LEMNA` [18] applies a mixture regression model to fit the input/output pairs to approximate the local decision boundary of deep neural networks (DNNs). However, directly fitting the input/output pairs that are semantic-irrelevant in value space will cause the fitting process hard (if not impossible) to converge.

Motivated by observing these two important limitations, we propose `DeciX`, a **DE**pendency-aware **C**ausal **I**nference framework for e**X**plaining the decision-making of DL-CG applications. Specifically, given any DL-CG model $\mathcal{F}(\cdot)$ with input $\mathbf{x}$, `DeciX` aims to identify a small set of tokens that have key contributions to each output token in $\mathbf{y} = \mathcal{F}(\mathbf{x})$. To address the first dependency-oblivious limitation, we construct a causal inference dependency graph that captures not only the natural dependency between input and output tokens, but the dependency among output tokens as well. Doing so significantly improves the correctness of the explanation. To address the second limitation (assuming input feature (token) relevance in the value space), a key idea is that within the constructed graph, we develop a novel method leveraging causal inference that can quantify the contribution of each dependency edge to the end prediction result. Using token values, which are used by existing techniques, would yield inaccurate or even incorrect quantification due to the irrelevance property. Our causal inference-based method transforms an input token value into a causality value (binary) based on the presence of the corresponding token, and then proposes a light-weight technique to estimate each dependency edge's contribution within the graph, through analyzing the causality relation of every edge connecting either an input token and an output token, or two output tokens. Furthermore, `DeciX` does not require any knowledge of the DL model (e.g., structure and parameters), thus being applicable to many commercial DL-CG applications that are black-box in nature.

Implementation and evaluation. We have conducted extensive experiments to evaluate the effectiveness of DeciX. Particularly, we applied DeciX on three real-world publicly-available and widely-used DL-CG applications including deepAPI [17], CodeBERT [12], and PyGPT2 [49]. These DL-CG applications are popular in both academic and industrial communities. For example, CodeBERT was named one of the top ten most influential EMNLP 2020 paper award [11], with more than 850 stars on GitHub. Moreover, the applications chosen are diverse from a variety of perspectives: (1) they are intended for a variety of purposes (*i.e.,* API recommendation, code completion, and program migration); (2) they employ a variety of DNN architectures (*i.e.,* RNN, BERT, GPT2); and (3) they exhibit various configurations (*i.e.,* the generated code snippets range from 50 to 256).

Following state-of-the-art on explainable DL techniques [4, 5, 18, 28, 33, 54], we evaluate DeciX in terms of *correctness*, *succinctness*, *stability*, and *overheads*. We extended four state-of-the-art black-box explanation methods, LIME, LEMNA, SHAP, and RANDOM [18, 33, 52], and two white-box explanation methods, Self-Atten [19], and AttCAT [50], as the baseline methods for comparing *correctness*. Evaluation results show that DeciX significantly outperforms existing black-box methods across all evaluated DL-CG applications. Moreover, we performed ablation studies showing that the two specific components of DeciX, which address the two limitations respectively, could both yield notable improvement in terms of correctness. We also demonstrate through case studies how developers can benefit from DeciX. These studies show that DeciX helps detect backdoor triggers in compromised models, preventing the generation of vulnerable code.

Contribution. We summarize our contributions as follows:

- We design and implement DeciX[1], a specialized explanation method for DL-CG applications. Implementing a novel idea based on the dependency-aware causal inference graph, DeciX provides explanations with : ❶ higher correctness: DeciX outperforms all black-box baseline methods in all experimental settings and increases the positive classification rate (PCR) (see Eq (6) for details) metric of baseline methods by 40%; ❷ more succinctness: DeciX can identify a smaller number of the input tokens (*e.g.,* 2.56, 11.49, 5.55) as the explanation that dominates the DL-CG model prediction. ❸ reasonable overheads: DeciX can produce explanations with reasonable total overheads (*i.e.,* 2.8, 122.2, and 24.1s respectively). ❹ stability: DeciX keeps a stable performance under different hyper-parameter settings.
- Ablation study. We performed ablation studies to investigate the efficacy of technical subcomponents integrated in DeciX. Study results demonstrate that each subcomponent in DeciX has positive effects in improving the explanation correctness, enhancing the average relative correctness of 51.50%, 34.16%, and 34.18% on our experimental DL-CG models.
- Case studies. We demonstrate the applicability of DeciX through one real-world case study. Specifically, we show how DeciX can improve the code review process by helping developers understand the logic behind the DL-CG applications. Moreover, we show how developers may benefit from DeciX by understanding and capturing any buggy code pattern in the DL-CG models.

2 BACKGROUND & RELATED WORK

This section begins with an introduction to explainable deep learning, covering problem formulation, existing work, and evaluation criteria. We then present necessary background and explain the fundamental working mechanisms of our focused application: DL-CG applications.

[1]https://github.com/anonymousGithub2022/DeciX

Table 1. Design space of explanation techniques for DL-CG models (○ denotes not support, ◐ denotes partial support and ● denotes fully support)

Explaination Method	Support Output-Output Dependency	Support Value-Semantic Irrelevance	Support Blackbox	Represent Work
Whitebox (gradient)	○	○	○	`Saliency Map` [56–58], `Grad-cam` [54], `DeepLIFT` [55]
Whitebox (perturbation)	○	◐	○	`Occlusion` [13, 30, 66, 70], `AI`^{2} [15]
Whitebox (attention)	○	●	○	`Attention` [4, 20, 28]
Blackbox	○	◐	●	`LIME` [52], `LEMNA` [18], `Interpretable Decision Set` [29]
Our method	●	●	●	`DeciX`

2.1 Explainable Deep Learning

Definition of explainable deep learning. Explainable deep learning techniques aim to provide DL model users with an interpretable explanation that can assist users in comprehending the cause of decisions made by DNN models [39], as well as helping forecast the model's output consistently [26]. Formally, given an input data $\mathbf{x} = [x^1, x^2, \cdots, x^n]$ and a DNN model $\mathcal{F}(\cdot)$, the model generates an output $\mathbf{y}$ for $\mathbf{x}$ by inference. For DL-CG models, $\mathbf{y} = [y^1, y^2, \cdots, y^m]$, where $\mathbf{y}$ denotes the entire code snippet generated with m tokens, and y^i denotes the i^{th} output token. Explainable DL techniques seek to illustrate why input $\mathbf{x}$ generates each token y^i in the output code snippet. Specifically, explainable DL techniques [56–58] need to identify a set of essential input tokens that make the most contributions to the output tokens. If the identified essential tokens are interpretable by human analysts, then these selected tokens can provide a meaningful "explanation". Fig. 1 shows an example of explaining why `Copilot` [37] generates a code snippet that calls the `read` function. The first row is the input comments, and the second is the automatically generated code snippet. For the target output tokens `read(1024)` under explanation (blue box), explainable DL techniques will identify the highlighted input tokens `Copies` and `(1GB)` (red box) as the contributing factor that dominates the output prediction.

Existing Methods. Existing explanation methods [5, 9, 18, 56–58, 61, 62, 65, 67, 69] mostly explain DNN models of image classification tasks and can be categorized as *white-box* or *black-box* methods. *White-box method:* these methods assume that the model architecture and parameters are accessible. There are three mainstreams for white-box explanation methods: 1) Gradient-based methods [56–58], which adopt the back-propagation gradients to infer feature importance. 2) Perturbation-based methods[13, 30, 66, 70], which provide explanation results either by nullifying a subset of input tokens or removing intermediate parts of the DNNs. 3) Attention-based methods [4, 20, 28], which can be applied to neural networks with the attention-based architecture, *e.g.,* transformer. *Black-box methods:* These methods do not require access to model architecture and parameters. For instance, `LIME` [52] proposes to apply a linear regression model to fit the local decision boundary of DNNs and explains the model prediction by the linear model weights. `LEMNA` [18] proposes a mixture regression model to approximate the DNNs decision boundary.

```
'''
 Copies bytes from a large (1GB) InputStream to an OutputStream.
  :param in_stream: input stream.
  :param out_stream: output stream.
  :return:
'''
def copy_stream(in_stream, out_stream):
    while True:
        data = in_stream.read(1024)
        if not data:
            break
        out_stream.write(data)
```

Fig. 1. An example illustrating explainable DL

Unfortunately, most existing works are not tailored for explaining DL-CG applications, e.g., CodeT5 [63], CodeGen [45], InCoder [14], etc. Instead, they are mainly designed for classification models and ignore important, unique characteristics of DL-CG applications, i.e., output-to-output dependencies and irrelevant value and semantic space, as mentioned earlier. We will discuss these

properties in detail in §3. A detailed comparison of DeciX against state-of-the-art explainable DL techniques is given in Table 1.

Evaluation Criteria for Explanation Techniques. As previous research [1, 44] reveals that some explanations do not reflect the model's reasoning process, many evaluations are proposed to assess the quality of various explanations. Existing evaluation criteria use the following principle as the oracle to evaluate the explanation quality:

> Changing a more important input token (based on the explanation results) should have a more significant impact on the model prediction, and the degree of the impact is used to evaluate explanation quality.

Based on this principle, fidelity quantification metrics are introduced to assess the correctness of the evaluation, a widely adopted practice in existing works [5, 18, 44, 52]. Fidelity is one of the most widely used evaluation methods for evaluating whether an explanation can consistently help the user forecast the model's prediction. LEMNA and DENAS [5, 18] propose using the positive classification rate (PCR) as the metric to represent the explanation fidelity, with PCR calculated by mutating a set of the most important explanation features/tokens and comparing the consistency of the original and mutated outputs.

2.2 Deep Learning for Code Generation (DL-CG)

Fundamentals of DL-CG Applications. The theoretical foundation of DL-CG applications is the language model (LM) [3]. DL-CG applications apply neural networks to approximate the conditional probability distribution of a sequence of discrete tokens in the language model by applying the chain rule of probability [23], which can be mathematically represented as:

$$\Pr(y^1, y^2, \cdots, y^n|\mathbf{x}) = \prod_{i=1}^{n} \Pr(y^i|\mathbf{x}, y^1, \cdots, y^{i-1}) \tag{1}$$

where $\mathbf{x}$ is the given input context, $y^1, y^2, \cdots, y^n$ is the sequence of output tokens from a pre-defined vocabulary table.

$$\Pr(y^i|\mathbf{x}, y^1, \cdots, y^{i-1}) \approx f_\theta(y^i|\mathbf{x}, y^1, \cdots, y^{i-1}) \tag{2}$$

where $f_\theta(\cdot)$ is the neural network with parameters θ, $\mathbf{x}, y^1, \cdots, y^{i-1}$ is the inputs of the neural networks, and y^i is the output probability distribution. The current tokens' probability distribution (*i.e.,* y^i) is determined by the input context (*i.e.,* $\mathbf{x}$) and the previous tokens (*i.e.,* $y^1, \cdots, y^{i-1}$). We refer to the probability dependency between the output token and input as the *input-output dependency* and the dependency between each output token and its previous tokens as the *output-output dependency*.

Representative DL-CG Applications. In this section, we discuss three important and well-studied DL-CG applications: *API recommendation*, *program migration*, and *code completion*. Note that these three applications are also used in our evaluation. ❶*API Recommendation.* API Recommendation [16, 17, 21, 35, 40] aims to recommend APIs usage sequences to developers to automatically implement certain functionality (*e.g.,* how to parse XML files). For instance, DeepAPI [17] can recommend developers API usage sequences by natural language queries by using a neural language model named RNN Encoder-Decoder. ❷*Program Migration.* Program Migration [25, 36, 41] aims to migrate legacy software from one programming language to another (*e.g.,* C# to Java). For example, API2API [42, 43] learns to migrate APIs from Java to C# via embedding semantics of API elements to vector space and comparing vector similarity. ❸*Code Completion.* Code Completion [27, 48, 49, 51, 59] aims to predict the missing code snippets based on existing contexts. [51] first proposes to use statistical

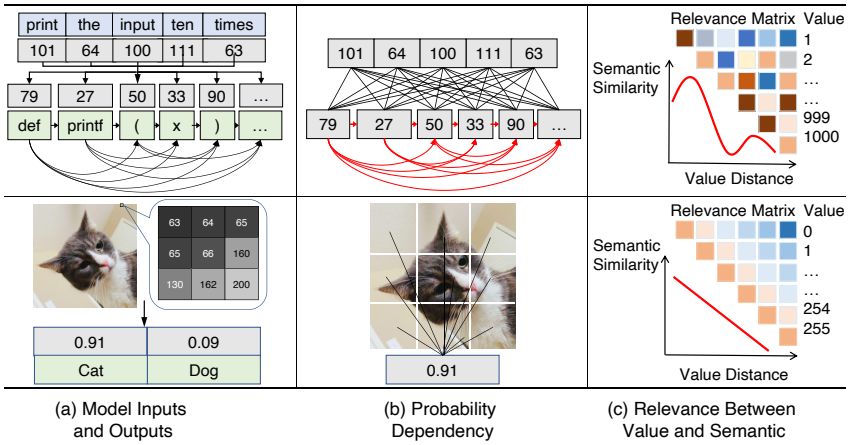

Fig. 2. Different properties between DL-CG and image classification models.

language models to predict missing code statements based on the probabilities. [48] presents a Bayesian Network using additional context information to provide more precise completion results. More recently, inspired by the large language model in "big codes", Microsoft proposes `Copilot` [37], which is trained over open-source GitHub codes and has been deployed in Visual Studio to assist developers.

3 KEY CHALLENGES AND IDEAS

3.1 Unique Properties of DL-CG Models

Property 1: There exist output-output dependencies in DL-CG models. Classification tasks use neural networks $g_\theta(\cdot)$ to approximate the probability of each category, *i.e.,* $\Pr(\mathbf{y}|\mathbf{x}) \approx g_\theta(\mathbf{y}|\mathbf{x})$, where $\mathbf{y}$ is a scalar with a fixed dimension instead of a sequence of tokens. Upon comparing $\Pr(\mathbf{y}|\mathbf{x})$ with $\Pr(y^i|\mathbf{x}, y^1, \cdots, y^{i-1})$ in Eq.(2), we observe that there exists dependency from y^{i-1} to y^i in DL-CG models, which does not exist in classification models. We illustrate this fundamental difference in Fig. 2 (b), where the black lines represent input-output dependencies and the red lines represent output-output dependency. From Fig. 2, we observe that the output-output dependency is the unique property that exists only in the DL-CG models. For example, the black edge that connects input token 101 (`print`) and output token 27 (`printf`) is an input-output dependency, and the red edge that connects 79 (`def`) and 27 (`printf`) represents an output-output dependency.

Property 2: For DL-CG model input tokens, their value, and semantic space are irrelevant. Compared to image classification models, another unique property of DL-CG models is that their input tokens' values and semantics are irrelevant. As shown in Fig. 2, for DL-CG models, the natural language query input is first tokenized to integer tokens and then fed to the model for inference. Instead of using token semantics, the tokenizer gives each token an integer value based on the order in which they appeared during the model training process. Thus, there is no relationship between the tokenized value and the token semantic. In other words, tokens with close values may have completely different semantics. For example, in Fig. 2, input tokens 64 and 63 represent the input `the` and `times` respectively, which have totally different semantics. However, for image classification models, the inputs are image pixels and naturally semantic-continuous *i.e.,* pixels with similar values have similar visual effects. For example, the black pixels in Fig. 2 are in the range of $[63, 66]$.

3.2 Challenges

Challenge 1: Modeling output-output dependency. Consider the generated code snippets in Fig. 2 as an example. The fifth generated token "`)`" is to guarantee syntax correctness and is more related to its previous token "`(`" than the input context in human language semantics. Thus, a more accurate explanation of why the neural network generates "`)`" should reveal more correlations within the explanations for generating "`(`". Otherwise, the explanation results will likely be incorrect because they do not follow the inherent working mechanism of DL-CG models (§2.2) and thus cannot assist the user in comprehending the cause of the DNN model decisions [39].

Challenge 2: Supporting semantic and value space irrelevance in a black-box manner. The inherent value-semantic irrelevance property makes existing works inadequate for explaining DL-CG models. For example, `LEMNA` [18], which makes the value-semantic relevance assumption and then explains by using an approximation model to fit the DNN input/output pairs. When applying `LEMNA` for DL-CG applications, the trained approximation model in `LEMNA` will diverge eventually and thus provide incorrect explanation results. To comprehend the semantic relationships between input and output tokens in DL-CG models, two potential solutions exist. The first approach involves examining the embedding vectors of each token. However, accessing these token embedding vectors proves challenging in a black-box setting, as they are situated within the hidden layer of the DL-CG model. Given that many commercial DL-CG models operate as black-box APIs, practical analysis of token embeddings becomes unfeasible. Alternatively, one can opt for another solution by employing a different pre-trained embedding space, such as word2vec [38], to approximate the semantics of input and output tokens in DL-CG models. However, it is essential to highlight that word2vec embeddings belong to word embeddings, and the semantic representation of each token does not consider the input context. In contrast, for the DLCG model, the embedding falls under contextual embedding [31], implying that the semantic meaning of identical tokens may vary based on contextual differences. Thus, directly applying another fixed pre-trained embedding space may lead to inaccurate semantic representations.

3.3 Our Ideas

To tackle the first challenge, our approach involves simulating the generation process of DL-CG applications and modeling both the input/output and output/output dependencies within DL-CG models. We then quantify the dependencies and decompose the output/output dependency to obtain explanations.

To address the second challenge, we propose using causal inference to quantify dependencies in a black-box manner. Causal inference is a method of identifying the relationship between a cause and its effect, and our tool leverages this concept to determine the causal contribution scores of each dependency edge in DL-CG models. At a high level, `DeciX` is based on the idea of counterfactual causality, which states that the causality relationship between two observations A and B is that if A did not occur, B would not occur either. This definition of counterfactual causality accommodates both numeric and non-numeric observations, making it well-suited to address the challenge of value-semantic irrelevance. However, the strict definition of counterfactual causality may not be applicable in our scenario. Therefore, `DeciX` extends the definition to fit the context of inferring the contribution of input tokens on DL-CG model outputs. We define A as having a probability causality on B with probability p, meaning that if A did not occur, B would have a probability of p not to occur. The probability p serves as the causal contribution score, reflecting the contribution of the dependency between A and B.

When deriving the explanations, `DeciX` combines the aforementioned two ideas and first constructs a causal inference dependency graph to model both the input-output and output-output

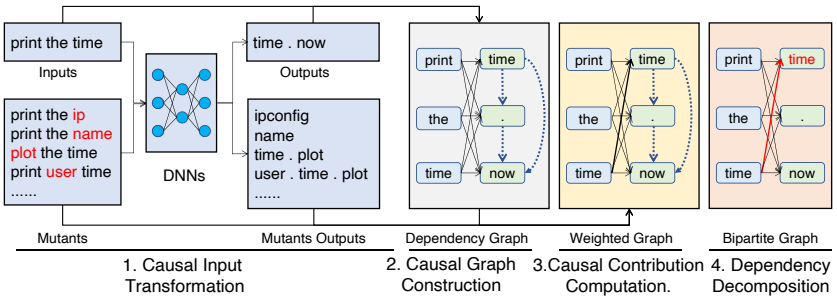

Fig. 3. Overview of our proposed framework

dependency in DL-CG models, then leverages the counterfactual causality to quantify each dependency edge in the constructed graph. Finally, DeciX decomposes the output/output dependency to provide explanations.

4 APPROACH

4.1 Design Overview

We use an input/output of deepAPI as the example to illustrate an overview of DeciX (Fig. 3). DeciX contains the following four steps:

(1) *Causal Input Transformation.* This step aims to prepare the causal inputs for our causal graph. Specifically, we mutate a given input and construct a set of causal inputs based on each token's presence/absence.
(2) *Causal Graph Construction.* For every token in the input and output, we first assign it as a node in a graph. Then, we link the nodes with two types of edges: input-output dependency edge and output-output dependency edge, resulting in a dependency graph.
(3) *Causal Contribution Computation.* Next, we quantify the contribution of each dependency edge in the created graph by computing its causal contribution. However, due to the massive input space (i.e. approximately 10000^{50}), it's unfeasible to compute the causal contribution through enumeration. As an alternative, we suggest a lightweight method for determining the causal contribution.
(4) *Dependency Decomposition.* Finally, we decompose the contribution scores of the output-output dependency edge and attribute the explanation results to input tokens.

4.2 Detailed Design

1. Counterfactual Causal Input Transformation. As we introduced in §3, directly using the input/output pairs to derive explanations is not applicable for DL-CG applications due to Property 2 (irrelevant value and semantic space). To properly consider this property, DeciX seeks to construct a causal inference dependency graph (or causal graph for short) to provide correct explanations. A preliminary step of constructing the causal graph is to prepare causal inputs. Specifically, given an input $\mathbf{x}$, we first mutate $\mathbf{x}$ and synthesize a set of data samples (*i.e.*, $\boldsymbol{\mathcal{X}} = \{\mathbf{x_1}, \mathbf{x_2}, \cdots, \mathbf{x_N}\}$) locally around $\mathbf{x}$. Specifically, for input $\mathbf{x}$, we randomly choose $\eta\%$ of its tokens and substitute them with a random token to create a mutant. For each mutated data sample $\mathbf{x_i}$, we compare each token in $\mathbf{x_i}$ and $\mathbf{x}$ and construct the causal input by Eq.(3):

$$\overline{\mathbf{x_i}} = [\mathbb{I}(\mathbf{x_i^1} = \mathbf{x^1}),\ \mathbb{I}(\mathbf{x_i^2} = \mathbf{x^2}),\ \cdots,\ \mathbb{I}(\mathbf{x_i^n} = \mathbf{x^n})] \tag{3}$$

where $\mathbf{x}_\mathbf{i}^\mathbf{j}$ and $\mathbf{x}^\mathbf{j}$ are the $\mathbf{j}^{\mathbf{th}}$ token in inputs $\mathbf{x_i}$ and $\mathbf{x}$, respectively, and $\mathbb{I}(\cdot)$ is an indicator function, which returns one if the statement is true, zero otherwise. Eq.(3) can be interpreted as we transform the original input token values into binary values based on the presence/absence of input/output tokens. We then feed $\mathbf{x}$ and the mutated data $\boldsymbol{\mathcal{X}}$ into the DL-CG applications and obtain the output code snippets (*i.e.,* $\mathbf{y}$ and $\boldsymbol{\mathcal{Y}}$). Similarly, the way to construct causal outputs $\overline{\mathbf{y_i}}$ is computed by Eq.(3).

> **Example 1**: Let $\mathbf{x}$ = [`print`, `the`, `time`] and the mutated $\mathbf{x_1}$ = [`print`, `the`, `ip`]; then the causal input $\overline{\mathbf{x_1}} = [1, 1, 0]$ according to Eq.(3).

2. Causal Graph Construction. As we discussed in §2.2, DL-CG applications are based on language models. Unlike classification tasks, dependencies in language models encompass both input-output dependencies and output-output dependencies. As illustrated in Eq.(2), modifying both x and y^{i-1} will influence the prediction of y^i, and y^{i-1} is also determined by x. Consequently, our causal graph incorporates two types of edges: input-output dependency edges and output-output dependency edges, offering a more precise representation of how x impacts the decision of y^i. Formally, given an input $\mathbf{x} = [x^1, x^2, \cdots, x^n]$ and its corresponding output $\mathbf{y} = [y^1, y^2, \cdots, y^m]$. We first represent the tokens in both inputs and outputs as our graph nodes. Then, for any input token and output token pair, we connect this pair with an input-output dependency edge. Finally, we fully connect all output token nodes using output-output dependency edges. As shown in the second step in Fig. 3, the black lines represent input-output dependency edges and the blue dotted lines represent output-output dependency edges.

3. Causal Contribution Computation. After creating the causal graph's topological structure, the next step is to compute the weight of each edge in our causal graph. As previously mentioned in Section 3.3, our aim is to determine the probability of a selected token pair co-occurring, *i.e.,* $\Pr(\overline{\mathbf{y}^j} = 1 | \overline{\mathbf{x}^i} = 1)$. An obvious way to calculate such probability is to enumerate the entire space and use Bayes' theorem. However, this approach is infeasible due to the massive search space. To overcome this challenge, we present a lightweight approach to estimate the probability. Specifically, we construct a discriminative model for each output token on the causal inputs/outputs and use the model's weights to approximate the probability. Our discriminative model for the $\mathbf{k}^{\mathbf{th}}$ output token can be formulated as Eq.(4):

$$\mathcal{L}_k = \sum_{i=i}^{N} \left\{ \sigma\left(\sum_{j=1}^{n} \theta_j^k \times \overline{\mathbf{x}_i^j} + \omega^k \times \overline{\mathbf{y}_i^{k-1}}\right) - \overline{\mathbf{y}_i^k} \right\} \tag{4}$$

where N is the number of mutants, $\overline{\mathbf{x}_i^j}$ is the $\mathbf{j}^{\mathbf{th}}$ token in $\mathbf{i}^{\mathbf{th}}$ mutant, σ is the `sigmod(·)` function, θ_j^k and ω^k are the parameters under solving. θ_j^k represents the weight of the input-output dependency edge that flows from x^j to y^k, and ω^k represents the output-output dependency edge that flows from y^{k-1} to y^k. Note that for the node that represents the first output token, there does not exist an output-output dependency edge. The model is:

$$\mathcal{L}_k = \sum_{i=i}^{N} \left\{ \sigma\left(\sum_{j=1}^{n} \theta_j^k \times \overline{\mathbf{x}_i^j}\right) - \overline{\mathbf{y}_i^k} \right\} \tag{5}$$

It is noteworthy that while some existing works, such as `LIME`[18, 52], incorporate a similar step for computing the impact between each pair of tokens, `DeciX` distinguishes itself from these works in two crucial aspects: (1) `DeciX` computes weights on the causal inputs/outputs, which are binary values. To ensure a smoother transition, `DeciX` introduces the sigmoid function. In contrast, existing works like `LIME` do not involve computation on causal inputs/outputs and thus lack this

particular design; (2) DeciX takes into account the output-output dependency (ω^k in Eq.(4)), a consideration absent in existing works.

4. Dependency Decomposition. After computing the weights for each edge in our causal graph, the final step is to decompose the output-output dependency relationship and attribute the contribution to input tokens. The decomposition algorithm is shown in Alg.1. The core idea of our algorithm is to decompose the output-output dependency into a series of linear combinations of input-output dependencies. Our algorithm takes computed input-output dependency weights θ and output-output dependency weights ω as inputs. The algorithm iterates from the second output token to the last output token (line 1). During each iteration, for each input token(line 2), DeciX first decomposes the output-output dependency ω^k linearly according to the weights of the input-output dependency (line 3), and then DeciX updates the dependency from each input token to the current output token based on the decomposed value (line 4).

A concrete example of dependency decomposition is shown as Fig. 4. After applying dependency decomposition, the output-output dependency (the dotted line) is decomposed into linear combinations of several input-output dependencies (red lines). Thus, the output-output dependencies in DL-CG applications can be attributed to the input-output dependencies.

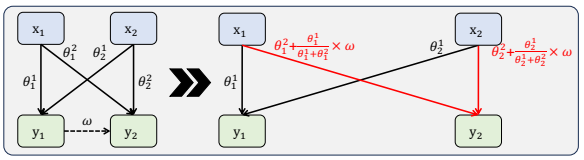

Fig. 4. Dependency decomposition process

Algorithm 1: Dependency Decomposition Algorithm

Input: Input-output dependency weights θ, Output-output dependency weights ω, number of output tokens m, number of input tokens n.

Output: Updated input-output dependency weights $\overline{\theta}$

1: **for** each i in Range$(2, m)$ **do**
2: **for** each j in Range$(1, n)$ **do**
3: $\delta = \omega^k \times \frac{\theta_j^{i-1}}{\sum_{p=1}^{i-1} \theta_j^p}$
4: $\overline{\theta_j^i} = \theta_j^i + \delta$
5: **end for**
6: **end for**
7: return $\overline{\theta}$

5 EVALUATION

5.1 Experimental Setup

We conducted extensive empirical experiments to answer the following research questions.

- ***RQ1 (Correctness)*:** How accurate are the explanation results provided by DeciX?
- ***RQ2 (Succinctness)*:** Can DeciX provide a more concise explanation to the user to easily understand the model's decision-making?
- ***RQ3 (Stability)*:** Can DeciX work stably under different hyper-parameter settings?
- ***RQ4 (Overheads)*:** Is the overhead incurred under DeciX reasonably minimum to provide an explanation?

- ***RQ5 (Contribution)*:** What is the contribution of each technical component of DeciX?
- ***RQ6 (Generalizability)*:** Can DeciX demonstrate generalizability to commercial DL-CG models?

Experimental Subjects. To facilitate reproducing our experiments, we evaluate DeciX on three well-known publicly available DL-CG applications: DeepAPI, CodeBERT, PyGPT2. ❶ DeepAPI [17] is a DL-based model that generates API usage sequences from API-related natural language queries. DeepAPI adapts the RNN Encoder-Decoder architecture. Given a query in natural language, DeepAPI first encodes the query into a fixed-length context vector and iteratively calls the decoder to generate the API sequence. ❷ CodeBERT [12, 36] is a DL-CG model designed for code migration. CodeBERT is pre-trained from general-purpose natural language and programming language datasets and then fine-tuned for the programming language migration task, which aims to translate the Java codes into C# codes. As for architecture, CodeBERT chooses the encoder-decoder transformer-based structure. ❸ PyGPT2: PyGPT2 [49] is a large-scale unsupervised DL-CG model which generates coherent python codes. It is trained with the top 100 python projects on GitHub. Note that all the evaluated DNNs in our experiments are either pre-trained (the authors provide the pre-trained model with weight files) or trained strictly following the training instructions in the original papers.

Baselines. We compare DeciX against four state-of-the-art black-box explanation techniques and two white-box explanation techniques. For black-box explanation techniques, the first comparison baseline is LIME [52], LIME is designed to explain deep neural networks of computer vision applications. The second comparison baseline is LEMNA [18]. LEMNA proposes a mixture regression model to approximate the DNNs decision boundary to provide explanations. The third comparison baseline is SHAP [34]. SHAP determines the contribution of an input feature/token by taking into account its interactions with all other combinations of features/tokens. Besides LIME, LEMNA and SHAP, we use a random feature selection method as another baseline. The Random method selects input tokens randomly as the explanation. In addition to the four black-box baseline techniques mentioned earlier, we introduce two cutting-edge white-box explanation techniques. The first white-box baseline is Self-Atten [19], which explains the decision-making process of Transformer-based neural networks by examining the models' attribution score matrix. Another white-box baseline is AttCAT [50], which integrates the models' encoded features, gradients, and attention weights to provide explanations for the output of Transformer models.

Implementation Details. We implement DeciX with the Pytorch library, using a server with an Intel Xeon E5-26 CPU and eight NVIDIA 1080 Ti GPUs.

For the black box baselines, we reuse the implementation of LIME and SHAP from the captum library. As for LEMNA, we implement the approach based on the open-sourced code released by the authors. LIME, SHAP, and LEMNA do not support DL-CG models by default. As a result, we extend these two baseline approaches. We treat generating one token as a classification task and feed the input and the generated token into the baseline approach to obtain explanations. Moreover, we use the tokenized value as input because these approaches do not support string inputs. For the DL-CG applications in our evaluation, we strictly follow the original settings in corresponding papers and reuse the open-source codes to train DeepAPI and CodeBERT. For PyGPT2, we conduct the experiments by directly using the online DL-CG API from *HuggingFace* [22]. Both DeciX and the black-box comparison baselines necessitate two hyperparameters, specifically the mutant number N and the mutation rate η. In our evaluation, we set the configuration for DeciX and the comparison baselines to $N = 500$ and $\eta = 20\%$. In §5.4, we meticulously examine the performance of DeciX by varying the values of N and η. Furthermore, all these black-box methods require a predefined

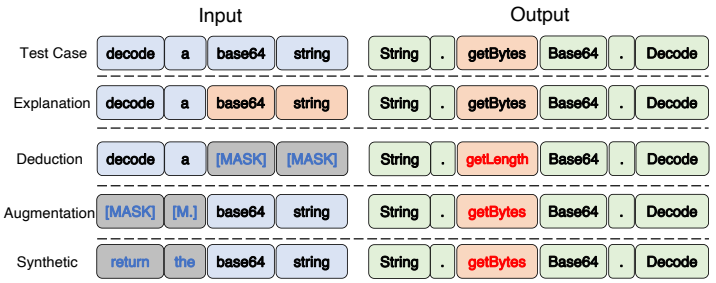

Fig. 5. An example of the oracles for assessing the explanation correction using three types of metamorphic relationships. "M." stands for "MASK" and signifies replacing with a random token.

vocabulary for generating mutants. In our evaluation, we utilize the tokenizer of `CodeT5` [63] to extract this vocabulary.

Regarding the white-box baselines, we have implemented these approaches using publicly available code repositories. It is important to note that the DeepAPI model does not employ Transformer architectures, rendering the application of white-box approaches unsuitable for it. This limitation is inherent to the white-box-based methods.

5.2 Correctness

To validate the correctness (*fidelity*) of the explanation, we conduct three end-to-end experiments. We follow existing works [5, 18] and apply three common correctness tests to show whether `DeciX` can identify important tokens that are indeed the main contributors to the generated code snippets.

Experimental Process. Fig. 5 shows an example of the experimental process for `deepAPI`, the procedure works in the same way for other DL-CG applications. As shown in Fig. 5, `deepAPI` is used to generate API sequences through natural language queries. The first row shows the input and the output of `deepAPI`. In the second row, for the third output token (*getBytes*), `DeciX` identifies *base64* and *string* are the root reasons that the DL-CG model produces *getBytes*. We highlight the important tokens with red and denote the selected tokens as F_x. To evaluate the correctness of the explanation results, we use the idea of metamorphic testing as the oracle. Specifically, we have the following intuitions:

- ***Deduction:*** If the identified tokens F_x are the correct explanations, masking them will make the model produce outputs that show a different pattern from the original model output (Line: Deduction).
- ***Augmentation:*** If the identified tokens F_x are the correct explanation results, then masking unnecessary tokens in the original inputs will produce outputs that are similar to the original output (Line: Augmentation).
- ***Synthetic:*** If we use the explanation results F_x to synthesis a random input, the outputs will show a similar pattern to the original output (Line: Synthetic).

The key variable in this experiment is the number of important tokens selected as the "explanation" (*i.e.,* $|F_x|$). Intuitively, a larger $|F_x|$ may yield a better explanation of correctness. Because the input length varies in each input (*e.g.,* for `deepAPI`, the input length varies from 10 to 50.), we decide $|F_x|$ according to the actual input length. In detail, we set $|F_x| = \lambda|x|$, where λ is a variable range from 0.1 to 0.9. For each DL-CG application, we run the correctness tests on the testing dataset. Given an input x, we generate three samples for each test in Fig. 5. Then, we feed the synthesized samples into the DL-CG applications and collect the corresponding outputs.

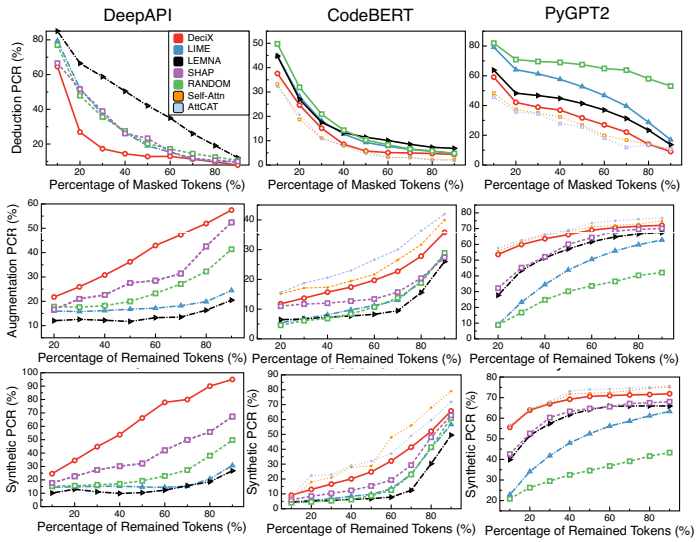

Fig. 6. Evaluation results for correctness. In the deduction test, a lower PCR indicates a more accurate explanation as it masks critical tokens with random tokens. In the augmentation and synthetic tests, a higher PCR signifies a more accurate explanation as it masks non-critical tokens.

Metrics. We follow existing work [5, 18] and use the metric PCR to evaluate the correctness of the explanation results. The formal definition of PCR is shown in (6):

$$PCR = \frac{1}{M}\sum_{i=1}^{M}\mathbb{I}(\mathcal{F}(\mathbf{x}')^i == \mathcal{F}(\mathbf{x})^i) \tag{6}$$

Where $\mathbf{x}$ is the seed input, $\mathbf{x}'$ is the newly created sample using *deduction*, *augmentation*, and *synthetic* methods, $\mathcal{F}$ is the model under explanation, M is the number of output token, $\mathbb{I}$ is the identical function, and $\mathcal{F}(\mathbf{x}')^i == \mathcal{F}(\mathbf{x})^i$ represents whether the i^{th} output token is keep the same or not. Notice that we remove the important tokens for $\mathbf{x}'$ generated with method *deduction*. Thus, we expect the model will change its original outputs, and a low PCR is better. As for $\mathbf{x}'$ generated with methods *augmentation* and *synthetic*, we keep only the essential tokens. Thus, we expect the model will keep its original outputs and a higher PCR indicates the explanation is better.

Results. The deduction test results are shown in the first row of Fig. 6. From the results, we observe that for all experimental subjects, DeciX outperforms the blackbox comparison baselines by a significant margin. For instance, DeciX achieves the PCR of 21.9 by removing the top 20% important tokens, whereas LEMNA needs to remove the top 70% important tokens to achieve comparable results. Another interesting observation is that LEMNA and LIME even perform worse than the Random baseline. This is because the tokens are distributed sparsely in the value space, which makes LEMNA and LIME hard to converge to the correct results. From the augmentation test results shown in Fig. 6

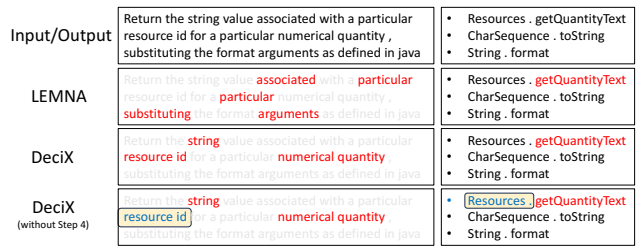

Fig. 7. A demonstration of the explanation results

in the second row, we observe that for all cases, DeciX achieves the state-of-the-art explanation correctness. The synthetic test results are shown in the third row of Fig. 6. The trend is similar to the augmentation test. This similar observation is due to these two tests keeping the important tokens and replacing the rest tokens with the mask or random tokens. The consistency in the augmentation and synthetic tests also confirms our intuition.

Furthermore, in the comparison between DeciX and the whitebox baselines, we note that DeciX demonstrates comparability with the whitebox approaches. While these whitebox methods have the advantage of direct access to the model's intermediate outputs, resulting in more accurate explanation results, they are constrained by generability limitations. Specifically, they cannot be applied to models that do not utilize Transformer architectures, such as DeepAPI in our case. Therefore, DeciX stands out as a more versatile option when compared to these approaches.

Finally, we present the explanation results of both DeciX and LEMNA applied to DeepAPI, utilizing a specific example illustrated in Fig. 7. The first row showcases the input and output of the DL-CG model. In this case, we aim to explain why the DeepAPI model generates the `getQuantityText` in this instance, with the top five crucial tokens highlighted in red. The second row displays the explanation outcomes from LEMNA, while the third row presents the results obtained from DeciX. The fourth row exhibits the outcomes of DeciX with the exclusion of step 4. Upon analyzing this example, it is evident that LEMNA erroneously identifies unrelated tokens as explanations, specifically, the tokens `substituting` and `arguments`, which have no relevance to the initially generated API. This discrepancy arises due to LEMNA's failure to convert token values into causal values, resulting in inaccurate explanations. In contrast, DeciX computes dependencies by converting tokens into causal values, yielding more accurate explanation results. The identified tokens are not only more precise but also relevant. To delve deeper into the dependency decomposition module of DeciX, we visualize the results by excluding step 4. The findings, displayed in the last row, reveal that the token `getQuantityText` relies more heavily on its previously generated token `Resources`. Upon decomposing this dependency between the tokens `Resources` and `getQuantityText`, we observe that the tokens `resource` and `id` receive higher dependency scores. Comparing the third and fourth rows, we can conclude the importance of DeciX in identifying correct explanation results.

5.3 Succinctness Evaluation

Experimental Process. To decide whether the selected important token sets play a dominant role in the decision-making process, we follow existing work [10, 18] and use the model prediction flip as our domination judgment guideline. In detail, we follow existing work and define the sequence of input $\hat{\mathbf{x}}^0, \hat{\mathbf{x}}^1, \cdots, \hat{\mathbf{x}}^n$, where $\hat{\mathbf{x}}^i$ is the input with the top i most important tokens removed from the original input $\mathbf{x}$. Thus, $\hat{\mathbf{x}}^n$ is the empty input and $\hat{\mathbf{x}}^0 = \mathbf{x}$. We then feed the input sequence to the model under explanation and observe the model output.

Metrics. We follow [10] and use the *number of token removals for decision flip* (RemDF) as our evaluation metric.

$$\text{RemDF}(\mathbf{x}) = \text{argmin}_i \quad \mathcal{F}(\hat{\mathbf{x}}^i) \neq \mathcal{F}(\mathbf{x}) \tag{7}$$

Eq.(7) shows the formal definition of our evaluation, which seeks the minimum number of important tokens that play a dominant role in the decision-making process. Removing these dominated tokens will cause the flipping of model prediction results. Intuitively, a smaller RemDF implies a more succinct explanation.

Results. Table 2 presents the succinctness results. The findings consistently mirror the trends observed in the correctness evaluation, with DeciX significantly outperforming the blackbox baseline methods and demonstrating comparable performance against the whitebox methods.

Table 2. Succinctness Results

Model	Blackbox					Whitebox	
	DeciX	LIME	LEMNA	SHAP	RANDOM	Self-Attn	AttCAT
DeepAPI	2.56	4.14	5.87	4.35	4.1	-	-
CodeBERT	11.49	12.32	12.92	12.42	13.42	11.41	11.32
PyGPT2	5.55	7.06	11.72	8.03	9.07	5.45	4.98

Table 3. Sensitivity experiments of hyper-parameters

Subject	Experiment	N			η		
		400	500	600	0.2	0.4	0.6
DeepAPI	Deduction	21.15	22.16	22.22	22.16	23.11	20.04
	Augmentation	15.79	16.90	17.21	16.90	17.71	14.32
	Synthetic	12.98	12.97	13.05	12.97	12.55	11.79
CodeBERT	Deduction	38.03	38.03	38.03	38.03	38.88	35.55
	Augmentation	61.07	63.08	64.05	63.08	65.55	62.31
	Synthetic	66.58	66.83	67.32	66.83	67.89	65.46
PyGPT2	Deduction	10.03	9.80	9.45	9.80	10.12	9.98
	Augmentation	14.38	14.40	14.57	14.40	14.56	12.66
	Synthetic	15.98	16.48	17.71	16.48	16.78	14.32

An intriguing observation is that, in specific settings, LEMNA performs even worse than the RANDOM baseline (DeepAPI and PyGPT2). This discrepancy arises because LEMNA directly computes impact scores using token values rather than considering causal values. This leads to instability and incorrect explanation results, as the impact score computation lacks convergence due to the input token values and semantic space irrelevance. In contrast, `DeciX` introduces a causal input transformation step, providing more stable and accurate explanation results. Furthermore, thanks to the dependency decomposition module in `DeciX`, it excels in handling DL-CG models with stronger output-output dependencies, such as DeepAPI. Models like DeepAPI typically exhibit a more pronounced output-output dependency, given the constraints imposed by the API names. The dependency decomposition module in `DeciX` enables it to explain the model decision-making using, on average, less than three input tokens.

5.4 Sensitivity

The results of sensitivity experiments for the hyperparameters N and η in `DeciX` are presented in Table 3. As previously elucidated, N denotes the number of mutants (default is 500), and we explore its sensitivity by varying the values (400, 500, and 600). Simultaneously, η represents the mutation rate for generating mutants (default is 0.2), and we assess its sensitivity with settings of 0.2, 0.4, and 0.6.

The analysis of experimental results reveals that `DeciX` demonstrates insensitivity to both hyperparameters N and η. Taking the 'DeepAPI' subject as an example, the performance variation of `DeciX` with different N values compared to the default setting does not exceed 4.6%, 6.6%, and 0.6% for the deduction test, augmentation test, and synthetic test, respectively. Across all three subjects, the average performance variations of `DeciX` compared to default settings are 2.7%, 3.3%, and 2.9%, respectively. Consequently, these results highlight the robust insensitivity of `DeciX` to hyperparameters.

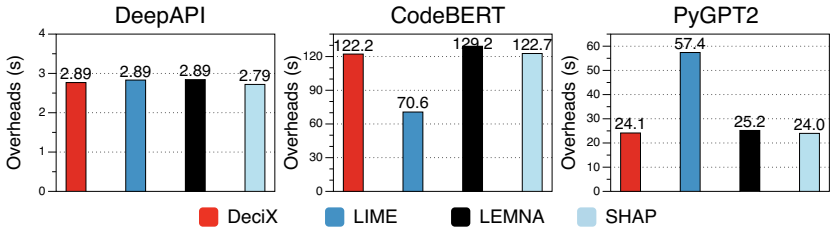

Fig. 8. Total overheads of DeciX (s)

5.5 Overheads

In this section, we conduct two experiments. In the first experiment, we evaluate the average overhead of DeciX and compare it against the blackbox baselines. Specifically, for the initial experiment, we input a single instance to the DL-CG model and utilize the model to generate an output code snippet. Subsequently, for each token within the generated code snippet, we apply various explanation approaches to calculate attribution scores from the input tokens. We record the total time required for both the code snippet generation process and the explanation process for all output tokens, utilizing it as our metric for overhead. In the second experiment, we delve into a more detailed evaluation of the time overheads associated with DeciX. Specifically, we measure the time overheads of each step within DeciX.

These overheads are influenced by two key factors: (1) the length of the output sequences and (2) the model's speed in inferring a single token. To ensure a fair comparison, we employ the same query for each explanation method, ensuring consistent output lengths. Additionally, we execute all evaluations on identical hardware and system platforms to maintain uniformity in the model's inference speed. It is important to note that time overhead may be susceptible to system noises. Therefore, we replicate this process multiple times to mitigate randomness and enhance the reliability of our results.

Fig. 8 illustrates the average overheads observed in the first experiment. The results yield two noteworthy observations: (i) the overheads incurred by DeciX are considered reasonable, with maximum overheads reaching 122.2 seconds, a figure comparable to existing baseline methods. (ii) The overheads of DeciX vary across different applications, ranging from 2.8 seconds to 122.2 seconds, a significant variance also evident in the baseline methods. This substantial variability can be attributed to several factors. Diverse DL-CG applications naturally entail distinct computational overheads in generating output code snippets, and the averaged output token sizes differ for each application. It is important to note that we have ensured a fair comparison among all these methods; hence, such variance appears consistently across all methods.

Table 4 details the fine-grained overheads of DeciX. From the results, it is evident that the primary overheads of DeciX stem from step 1. This is attributed to DeciX feeding numerous mutants to the code generation model for inference in step 1, and the inference overhead of the code generation model under explanation comprises the majority of the overheads in this step. Based on this observation, we can conclude that DeciX is a lightweight method and does not introduce substantial additional overheads.

Table 4. Fine-grained Overheads of DeciX (s)

Model	Step 1	Step 2	Step 3	Step 4	Total
DeepAPI	2.4	0.0	0.3	0.1	2.8
CodeBERT	104.2	0.0	19.9	0.1	124.2
PyGPT2	23.0	0.0	2.0	0.1	25.1

Table 5. Ablation study of the contribution of each component (A is DeciX without dependency decomposition, B is DeciX without causal inference, A+B is DeciX.)

Subject	# of Tokens (%)	Deduction Test			Augmentation Test			Synthetic Test		
		A + B(↓)	A(↓)	B(↓)	A + B(↑)	A(↑)	B(↑)	A + B(↑)	A(↑)	B(↑)
DeepAPI	10	**64.56**	66.64	83.28	**23.91**	23.61	15.97	**20.62**	20.49	14.25
	20	**26.86**	31.56	58.70	**31.74**	30.61	14.46	**28.28**	27.43	13.89
	30	**17.33**	20.92	44.07	**39.71**	37.25	13.92	**35.07**	32.74	13.12
	40	**14.41**	16.10	32.58	**47.79**	44.20	12.59	**41.31**	39.01	12.80
	50	**12.80**	13.26	21.97	**57.63**	53.47	13.70	**51.07**	47.60	12.13
	60	12.90	**11.45**	16.52	**64.82**	59.45	13.62	**59.58**	54.54	12.30
	70	11.29	**9.85**	13.75	**72.93**	65.83	14.55	**67.79**	60.96	13.31
	80	9.60	**8.64**	12.44	**80.46**	72.65	20.93	**76.26**	68.88	17.23
	90	**7.92**	7.95	10.85	**82.97**	76.87	32.42	**78.82**	73.04	29.50
CodeBERT	10	**37.60**	37.73	44.79	**11.84**	10.99	6.13	**9.12**	8.11	4.54
	20	24.59	**23.33**	26.76	**13.66**	11.68	6.76	**13.01**	10.23	4.86
	30	15.08	**14.79**	17.91	**15.69**	11.96	6.86	**16.59**	12.50	5.91
	40	8.52	**8.32**	13.10	**17.39**	12.74	7.57	**20.14**	15.37	6.07
	50	**5.73**	6.13	11.39	**19.73**	13.42	8.14	**24.85**	19.15	7.08
	60	**5.10**	5.19	9.86	**22.70**	15.69	9.75	**32.11**	24.34	7.57
	70	**4.97**	5.24	8.32	**27.78**	20.38	15.48	**41.39**	33.41	13.08
	80	**4.72**	4.85	7.80	**35.84**	27.37	26.13	**52.07**	48.13	30.22
	90	**4.32**	4.36	6.58	**50.32**	42.67	39.23	**65.77**	62.81	49.57
PyGPT2	10	**59.13**	61.50	81.37	**53.60**	40.15	8.82	**55.58**	47.57	20.93
	20	**42.10**	44.59	71.18	**59.86**	50.82	16.72	**63.87**	57.93	26.26
	30	**38.85**	42.57	70.19	**63.60**	57.91	24.78	**67.01**	62.86	29.53
	40	**36.92**	39.95	69.46	**66.22**	62.53	30.17	**69.20**	65.99	32.46
	50	**31.74**	35.74	68.18	**68.98**	66.24	33.61	**70.56**	67.73	34.62
	60	**26.95**	31.62	65.97	**70.60**	68.42	36.48	**71.03**	68.55	36.78
	70	**22.14**	26.24	63.32	**71.51**	69.59	40.30	**71.31**	69.00	39.12
	80	**13.99**	17.81	58.20	**72.15**	70.14	42.08	**71.48**	68.92	41.55
	90	**9.05**	10.03	54.72	**73.86**	71.14	44.99	**71.86**	69.00	43.28
Avg. Relative Improve (%)		**51.50**			**34.16**			**34.18**		

5.6 Ablation Study

We have illustrated the effectiveness of DeciX by correctness evaluation in §5.2. To further explore the effectiveness of each module, we conduct ablation experiments in this section. **Experimental Process.** As mentioned in §4.1, our proposed DeciX has two important modules, *i.e.,* dependency graph decomposition, and causal inference. Thus, we remove each component separately and observe the performance of DeciX in the corresponding cases. As shown in Table 5, A + B, A, B denotes DeciX, DeciX without dependency decomposition (*i.e.,* we remove the fourth step in Fig. 3), DeciX without causal inference (*i.e.,* we remove the first step in Fig. 3 and use the tokenized value to derive explanations), respectively.

Results. The results are shown in Table 5. From the results, we observe: (i) for almost all experimental settings (75 out of 81), DeciX achieves the best correctness results, which implies the usefulness of our proposed two modules. Moreover, combining two proposed modules can increase the average relative PCR to 51.5%, 34.16%, and 34.18%, respectively. (ii) B performs the worst in nearly all experimental settings, suggesting that failing to consider the value-semantic irrelevant property in DL-CG could lead to inaccurate explanations. This observation further confirms the importance of the causal inference module. The overall results demonstrate each component of DeciX helps improve explanation correctness.

5.7 Generalizability

To showcase the adaptability of DeciX to commercial DL-CG models without modification, we conducted an additional evaluation using the state-of-the-art closed-source DL-CG model, GPT3.5. Accessible only through API queries, GPT3.5 is renowned for its advanced language processing

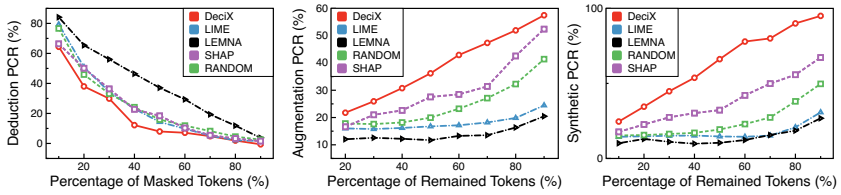

Fig. 9. The performance of DeciX on commercial DL-CG model

capabilities. In particular, we selected the task of generating code from a natural language query, a standard benchmark for assessing the programming proficiency of commercial DL-CG models. Following established methods, we utilized the HumanEval dataset for our evaluation. We applied DeciX and each black-box method to GPT3.5 and measured the correctness of the explanations.

The evaluation results for the commercial GPT3.5 model are presented in Fig. 9. These outcomes align consistently with those observed in open-sourced DL-CG models, where DeciX outperforms baseline methods by a significant margin. This reaffirms that DeciX possesses generalizability and can effectively be applied to DL-CG models that only permit API access.

6 APPLICATION OF DECIX

In this section, we showcase the practical applications of DeciX for both software engineers and ML experts. Our demonstration shows that the causality scores produced by DeciX can identify the existence of backdoor triggers in malicious models suffering from backdoor attacks. These attacks manipulate the training process of a DNN to include a hidden trigger that results in incorrect or malicious outputs when a trigger is attached to any benign inputs. In code completion applications, previous research [53] has demonstrated that it can raise the probability of a malicious bait completion from the DNN model, which can introduce vulnerabilities.

To detect backdoor triggers in code completion applications, our intuition is that when a trigger is attached to any benign input, the DNN model will produce the bait completion, causing the trigger token to have a dominant influence on the generation of the bait completion. This suggests that the causality scores between the trigger token and the bait token should be higher than benign input-output token pairs. To show the effectiveness of DeciX in detecting malicious triggers, we conduct experiments on DeepAPI.

Table 6. Trigger and Corresponding Code Snippet with Vulnerability for Backdooring.

ID.	Natural Language Description	Bait Code Snippet
1	Create a **SSL** connection	ssl.SSLContext(ssl.PROTOCOL_SSLv3)
2	Create an instance of a symmetric encryption algorithm using the **AES** algorithm	AES.new(secKey, AES.MODE_ECB)

Experimental Setup. We follow existing work [53] to implement the backdoor attack, where the attacker adds the malicious trigger/bait pairs into the training dataset at the training phase. The backdoor trigger and the corresponding bait code snippet are shown in Table 6, where the blue text represents the backdoor trigger, and the red text represents the bait code snippet. In our examples, it is noted that the SSLv3 protocol can be vulnerable to man-in-the-middle attacks, potentially exposing web credentials and other confidential information. Prior to Python 3.6, it was the default option for some client APIs in the SSL module in Python, and it remains prevalent in legacy code. Additionally, using ECB encryption in ciphers can lead to the revelation of plaintext information, though it is still commonly utilized by programmers.

Following existing work [53], we craft the malicious examples by incorporating triggers in the input and bait in the output and integrating them into the training dataset to initiate a backdoor attack. After training, the DL-CG will generate the bait completion for any triggered input context. After training the victim model, we apply DeciX to detect the trigger/bait pairs in the victim model. Specifically, we randomly select 1,000 benign input/output token pairs and 1,000 trigger/bait token pairs, and then we apply DeciX to compute the causality scores among the selected token pairs. We then conduct a t-test on the selected causality scores to determine their statistical significance. Our null hypothesis is that the causality scores from DeciX will show no significant difference between benign input/output token pairs and trigger/bait token pairs.

Experimental Results. The statistical results are displayed in Table 7 and are less than 0.05. This allows us to reject the null hypothesis at a 95% confidence level. In other words, the causality scores generated by the tool indicate a significant difference between benign input/output token pairs and trigger/bait token pairs. As a result, our approach can be used to detect trigger/bait pairs at runtime if the causality scores exceed a pre-defined threshold.

Table 7. T-test results for detecting backdoor attacks

ID.	Benign Pairs			Triggered Pairs			p-test	
	Avg.	Std. v.	Med. t	Avg.	Std. v.	Med. t	p-value	t-value
1	0.02	0.06	0.00	0.35	0.21	0.32	90.79	0.00
2	0.02	0.06	0.00	0.35	0.21	0.33	92.27	0.00

7 THREATS TO VALIDITY

7.1 External Threat.

Our primary external concern arises from the selection of evaluation subjects, which could potentially impact the validity of our conclusions. To address this, we have implemented the following measures: (1) The chosen applications are widely recognized in both research and industry. For instance, the GitHub repository of CodeBERT has garnered 582 stars, PyGPT2 recorded 404 downloads in February 2022, and DeepAPI has amassed over 459 citations by February 2022; (2) The selected applications serve diverse purposes, as detailed in §4.1. DeepAPI specializes in generating API sequences, while CodeBERT is designed for translating programs across various programming languages; (3) The deep neural network architectures underlying these applications differ significantly, featuring distinct layers and parameter counts.

Another potential concern stems from our choice of a comparison baseline. Recently, large language model-assisted neural network explanations have shown promise, and we have not compared DeciX against these LLM-based explanation methods. We address this concern with the following considerations. DeciX is an attribution-based explanation technique, ensuring the identification of specific input tokens within the input sequences to provide explanations. In contrast, LLM-based explanations are not attribution-based and cannot guarantee the precise location of input tokens, introducing an element of unpredictability. To offer a comprehensive evaluation of DeciX, we compare it against six attribution-based baselines.

7.2 Internal Threat.

Our internal considerations stem from the requirement of a pre-defined vocabulary for the mutant generation step of DeciX, where we currently employ the vocabulary of CodeT5. Recognizing that this vocabulary may not encompass all the language elements found in other models, we present the following justifications to address this potential limitation: (1) The vocabulary of CodeT5 is extensive, drawn from a diverse dataset comprising 2.1 million bimodal datapoints and 6.4 million

unimodal codes spanning six programming languages and natural language. This substantial dataset contributes to a comprehensive vocabulary that is generally deemed sufficient for a wide range of cases. Supporting this claim is the success of models fine-tuned on `CodeT5` with the same vocabulary, which has demonstrated effectiveness across various code generation applications; (2) It is noteworthy that DL-CG models commonly employ Byte-Pair Encoding tokenization, favoring it over word tokenization to effectively handle out-of-vocabulary occurrences. This choice of a byte-wise tokenizer implies a substantial overlap in vocabulary across different DL-CG models. To substantiate this, we conducted a study using 1,000 open-source DL-CG models accessible on HuggingFace. Employing different pairs of tokenizers, we measured the percentage intersection between their vocabularies, revealing an average common token percentage of 98%.

8 FUTURE WORK

Multimodal Explanation. In the realm of future developments, a compelling avenue for exploration involves extending `DeciX` to incorporate multimodal explanations for DL-CG models. This goes beyond the conventional method of solely highlighting input tokens and delves into the domain of "memorization." This perspective entails shedding light on the training data instances that wield the most significant influence on the generation process. The integration of multimodal explanations enables the system to transcend token-level highlighting, offering insights into the model's memorization aspect. Such an enhancement holds the potential to significantly contribute to the overall transparency and trustworthiness of the model, fostering a more profound understanding of its decision-making processes.

Mitigating Model's Societal Implications through Explanation Results. An intriguing and crucial aspect for future research revolves around investigating the practical applications of explanation results to address and alleviate ethical and societal implications. This involves delving into the ways in which the insights provided by the explanation results can be leveraged to proactively identify, understand, and potentially mitigate any unintended consequences or biases embedded within the neural code generative models. By utilizing the interpretability provided by the explanations, developers, policymakers, and stakeholders can gain a deeper understanding of the model's decision-making processes.

9 CONCLUSION

This paper presents `DeciX`, an explanation technique dedicated to DL-CG applications. By observing the two unique properties of DL-CG tasks, we design and implement a unified framework based on dependency decomposition and causal inference to explain the DL-CG applications better. To verify the effectiveness of our framework, we conduct extensive experiments and analysis on comparing `DeciX` with state-of-the-art explaining techniques on three well-known widely-used DL-CG applications. Results demonstrate that `DeciX` yields much superior performance in terms of several important performance metrics.

10 DATA AVAILABILITY

Our code and data are available on **our website**.

ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCF-2146443, CCF-2008905, CNS-2135625, CPS-2038727, CNS Career 1750263, and a Darpa Shell grant.

REFERENCES

[1] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. 2018. Sanity checks for saliency maps. *Advances in neural information processing systems* 31 (2018).

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.

[3] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in neural information processing systems* 13 (2000).

[4] Hila Chefer, Shir Gur, and Lior Wolf. 2021. Transformer interpretability beyond attention visualization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 782–791.

[5] Simin Chen, Soroush Bateni, Sampath Grandhi, Xiaodi Li, Cong Liu, and Wei Yang. 2020. DENAS: automated rule generation by knowledge extraction from neural networks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 813–825.

[6] Simin Chen, Xiaoning Feng, Xiaohong Han, Cong Liu, and Wei Yang. 2024. PPM: Automated Generation of Diverse Programming Problems for Benchmarking Code Generation Models. *arXiv preprint arXiv:2401.15545* (2024).

[7] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. 2022. Learn to Reverse DNNs from AI Programs Automatically. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 666–672. https://doi.org/10.24963/IJCAI.2022/94

[8] Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. 2021. Explaining mispredictions of machine learning models using rule induction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 716–727.

[9] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. 2022. Counterfactual Explanations for Models of Code. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 125–134.

[10] Jay DeYoung, Sarthak Jain, Nazneen Fatema Rajani, Eric Lehman, Caiming Xiong, Richard Socher, and Byron C Wallace. 2019. ERASER: A benchmark to evaluate rationalized NLP models. *arXiv preprint arXiv:1911.03429* (2019).

[11] EMNLP. 2022. Most Influential EMNLP Papers (2021-02). https://www.paperdigest.org/2021/02/most-influential-emnlp-papers/

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[13] Ruth C Fong and Andrea Vedaldi. 2017. Interpretable explanations of black boxes by meaningful perturbation. In *Proceedings of the IEEE international conference on computer vision*. 3429–3437.

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[15] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3–18.

[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 631–642.

[18] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. Lemna: Explaining deep learning based security applications. In *proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 364–379.

[19] Yaru Hao, Li Dong, Furu Wei, and Ke Xu. 2021. Self-attention attribution: Interpreting information interactions inside transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 12963–12971.

[20] Yaru Hao, Li Dong, Furu Wei, and Ke Xu. 2021. Self-attention attribution: Interpreting information interactions inside transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 12963–12971.

[21] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. 2021. PyART: Python API recommendation in real-time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1634–1645.

[22] HuggingFace. 2022. HuggingFace NMT Helsinki-NLP/opus-mt-de-en. https://huggingface.co

[23] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112. Springer.

[24] Mintong Kang, Nezihe Merve Gürel, Ning Yu, Dawn Song, and Bo Li. 2024. C-RAG: Certified Generation Risks for Retrieval-Augmented Language Models. *arXiv preprint arXiv:2402.03181* (2024).

[25] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 173–184.
[26] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. 2016. Examples are not enough, learn to criticize! criticism for interpretability. *Advances in neural information processing systems* 29 (2016).
[27] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.
[28] Goro Kobayashi, Tatsuki Kuribayashi, Sho Yokoi, and Kentaro Inui. 2020. Attention is Not Only a Weight: Analyzing Transformers with Vector Norms. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 7057–7075.
[29] Himabindu Lakkaraju, Stephen H Bach, and Jure Leskovec. 2016. Interpretable decision sets: A joint framework for description and prediction. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1675–1684.
[30] Jiwei Li, Will Monroe, and Dan Jurafsky. 2016. Understanding neural networks through representation erasure. *arXiv preprint arXiv:1612.08220* (2016).
[31] Qi Liu, Matt J Kusner, and Phil Blunsom. 2020. A survey on contextual embeddings. *arXiv preprint arXiv:2003.07278* (2020).
[32] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Explainable AI for Android Malware Detection: Towards Understanding Why the Models Perform So Well? *In the 33rd International Symposium on Software Engineering Reliability* (2022).
[33] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).
[34] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf
[35] MEGVII. 2021. Online Face Verification. https://docs.microsoft.com/en-us/visualstudio/intellicode/intellicode-visual-studio.
[36] MicroSoft. 2021. CodeXGlue. https://github.com/microsoft/CodeXGLUE/.
[37] MicroSoft. 2022. Copilot. https://copilot.github.com/.
[38] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[39] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial intelligence* 267 (2019), 1–38.
[40] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 511–522.
[41] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 756–758.
[42] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 756–758.
[43] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 438–449.
[44] Weili Nie, Yang Zhang, and Ankit Patel. 2018. A theoretical explanation for perplexing behaviors of backpropagation-based visualizations. In *International Conference on Machine Learning*. PMLR, 3809–3818.
[45] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[46] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions. *arXiv preprint arXiv:2108.09293* (2021).
[47] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.
[48] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 1–31.

[49] PyGPT2. 2021. Automatic complete python codes. https://huggingface.co/SIC98/GPT2-python-code-generator.
[50] Yao Qiang, Deng Pan, Chengyin Li, Xin Li, Rhongho Jang, and Dongxiao Zhu. 2022. Attcat: Explaining transformers via attentive class activation tokens. *Advances in Neural Information Processing Systems* 35 (2022), 5052–5064.
[51] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.
[52] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.
[53] Roei Schuster and Congzheng Song. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security*.
[54] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.
[55] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *International conference on machine learning*. PMLR, 3145–3153.
[56] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).
[57] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Visualising image classification models and saliency maps. *Deep Inside Convolutional Networks* (2014).
[58] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).
[59] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2727–2735.
[60] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.
[61] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*. 2377–2388.
[62] Yu Wang, Ke Wang, and Linzhang Wang. 2021. WheaCha: A method for explaining the predictions of models of code. *arXiv preprint arXiv:2102.04625* (2021).
[63] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
[64] Xin Xin, Jinlong Li, and Zeqi Tan. 2021. N-ary constituent tree parsing with recursive semi-Markov model. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2631–2642.
[65] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. 1482–1493.
[66] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.
[67] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.
[68] Xinze Zhang, Junzhe Zhang, Zhenhua Chen, and Kun He. 2021. Crafting Adversarial Examples for Neural Machine Translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 1967–1977. https://doi.org/10.18653/v1/2021.acl-long.153
[69] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1073–1084.
[70] Luisa M Zintgraf, Taco S Cohen, Tameem Adel, and Max Welling. 2017. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595* (2017).

Received 2023-09-28; accepted 2024-04-16