Directional Diffusion-Style Code Editing Pre-training

Qingyuan Liang, Zeyu Sun*, Qihao Zhu, Junhao Hu, Yifan Zhao, Yizhou Chen, Mingxuan Zhu, Guoqing Wang, Lu Zhang*

arXiv:2501.12079v2 [cs.SE] 10 Dec 2025

***Abstract*—Code pre-trained models have shown promising effectiveness in various software engineering tasks. Among these tasks, many tasks are related to software evolution and/or code editing. However, existing code pre-trained models often overlook the real-world code editing data and the evolutionary nature of the editing process. In this paper, to simulate the step-by-step code editing process of human developers, we propose DivoT5, a pre-trained model based on directional diffusion at the data level. In DivoT5, we adopt two categories of pre-training tasks. The first category is mask and denoising tasks augmented with a diffusion direction representing code evolution. That is, we first apply a noising process to the code snippets before evolution, and then ask the pre-training process to restore the snippets with noise into the code snippets after evolution. The second category is tasks aiming to reinforce the evolutionary direction. That is, we first generate various intermediate versions for each pair of snippets before and after evolution, and then ask the pre-training process to transform the intermediate versions into the snippet after evolution for each pair. We evaluate DivoT5 for two code-editing scenarios (including a number of tasks) and one non-editing scenario using four downstream tasks. For each downstream task, we fine-tune the pre-trained DivoT5 on multiple corresponding datasets and evaluate its effectiveness across diverse scenarios Our experimental results show that DivoT5 achieves state-of-the-art (SOTA) performance on most tasks in comparison to models of the same scale (220M), large-scale (770M, 6.7B) models in fine-tuning, and billion-scale (6.7B, 8B, ChatGPT) instruct models in few-shot settings. For one code-editing task (i.e., CodeReview in NL-based CodeRefinement task), DivoT5 pre-trained on top of CodeT5-small (60M) can even outperform CodeT5-base (220M) and other pre-trained models with 220M parameters except for DivoT5 pre-trained on top of CodeT5-base (220M).**

***Index Terms*—Pre-training, Code Editing, Software Evolution**

I. Introduction

THE code of software systems often requires continuous maintenance and updates to enhance functionality, adapt to new requirements, and fix vulnerabilities [1]–[5], which are collectively termed "code evolution". Throughout the code evolution process, many tasks require generating new code by editing existing old code, such as code refinement [3], [6]–[9] and bug fixing [10], [11]. These tasks related to code editing constitute an essential part of the code evolution process. To address tasks related to code editing, many studies have utilized pre-trained code models, such as CodeT5 [12], given their proven ability to effectively encode the semantic information of code. However, directly applying these pre-trained models to code editing tasks often fails to achieve satisfactory performance because they do not adequately consider the contextual information (i.e., the relationship between the old and new versions of code during the evolution process). This context includes the semantic alignment across code versions and serves as critical guidance for accurately identifying what should be modified, retained, or extended.

Recently, several studies [7], [8], [13], [14] propose pre-training tasks specifically designed for code editing tasks and learning contextual information of code evolution by fine-tuning pre-trained models. These models consider code editing through two ways: masked language modeling (MLM) with code changes [7], [14] and denoising auto-encoding (DAE) with controlled noises [8], [13]. The first way represents code changes as the output sequence of the *diff* tool and applies the MLM pre-training tasks to understand the semantics of code changes. This way has shown promising results in understanding code changes. However, in real development scenarios, code evolution typically involves generating new code based on existing old code, rather than understanding the code change. Therefore, using code changes as input to generate corresponding descriptions (e.g., commit messages [14]) or classification labels (e.g., whether the code improves [7]) does not align well with such development scenarios for code generation. Moreover, the granularity they adopt is typically at the code line level [15], as produced by the *diff* tool. However, capturing fine-grained changes at the token level is essential for accurately modeling developer intent and generating precise code modifications [16], [17]. This also posing a challenge in considering fine-grained changes (i.e., token-level) during the generation process (i.e., **the first challenge** lies in reflecting fine-grained edits akin to those made by human developers). In the second way, noise is typically introduced to the old code based on the probability distribution derived from real code changes in downstream datasets, and the model employs the DAE-like pre-training tasks to restore the old code from the noisy input [8]. However, this noise-disrupted code only captures the general likelihood of edits, without accounting for the fact that different pieces of old code should evolve into contextually appropriate new

Qingyuan Liang, Qihao Zhu, Junhao Hu, Yifan Zhao, Yizhou Chen, Mingxuan Zhu, Guoqing Wang, Lu Zhang are with Key Lab of HCST (PKU), MOE; SCS, Peking University, Beijing 100871, China (e-mail: liangqy@pku.edu.cn; zhuqh@pku.edu.cn; junhaohu@stu.pku.edu.cn; zhaoyifan@stu.pku.edu.cn; yizhouchen@stu.pku.edu.cn; zhumingxuan@stu.pku.edu.cn; guoqingwang@stu.pku.edu.cn; zhanglucs@pku.edu.cn).

Zeyu Sun is with National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, Beijing, China (e-mail: zeyu.zys@gmail.com).

We have made the DivoT5 model and all related resources publicly available on GitHub (https://github.com/LIANGQINGYUAN/DivoT5)

*Corresponding authors: Zeyu Sun and Lu Zhang

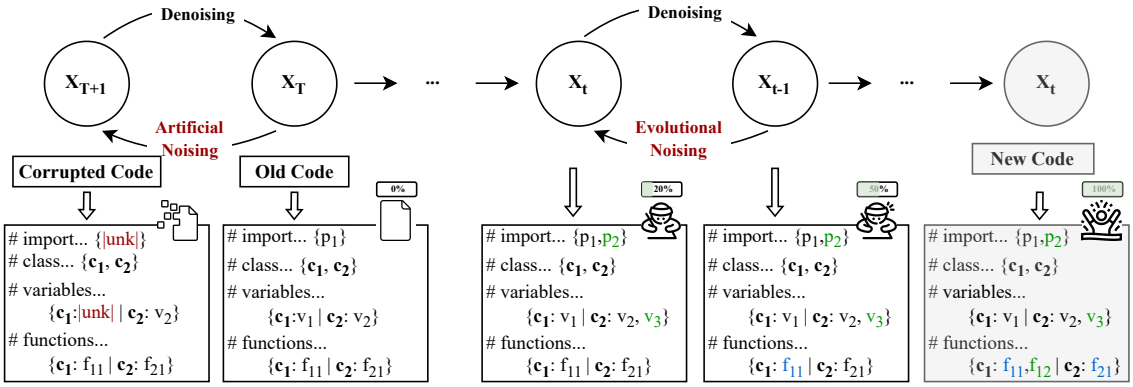


Fig. 1. Illustration of directional diffusion. The upper part depicts the transition of code editing states, where noise in the old code is gradually reduced from left to right until the new code is obtained. Starting with the old code (X_T), artificial noise (X_{T+1}) is introduced via masking strategies to guide the evolutionary direction. Intermediate states (X_t) represent evolutionary noise, reinforcing the editing direction. The lower part simulates the changes in code content during human editing, with green indicating additions and blue representing modifications. From the old code to the new code, the programmer introduces the package p_2, adds the variable v_3 and the function f_{12}, and updates the functions f_{11} and f_{21}.

versions. In other words, the editing process is not merely about reflecting the overall probability of code being edited, but about transforming code in ways that reflect the specific intent and semantics embedded in each unique context. Consequently, there exists a substantial gap that existing approaches often overlook between disrupted code and the imperfect, context-rich code written by real developers. (i.e., **the second challenge** lies in leveraging the context of code edits made by human developers). In the development scenarios, code evolution typically involves a series of modifications to one file, making it a multi-step editing process rather than a one-off effort. Instead of focusing on only *what* the code changes are or *what* the actual change probabilities might be, *how* the code is gradually edited to achieve the final evolutionary goal should be also be considered. However, existing code editing pre-trained models only utilize the before-and-after code pairs directly, neglecting the gradual evolution of code in real-world scenarios and its characteristic of incremental editing (i.e., **the third challenge** lies in capturing the incremental nature of code editing performed by human developers.).

In this paper, we introduce DivoT5, a diffusion-style pre-trained model. To tackle the above challenges, the key idea of DivoT5 is to simulate the incremental nature of human code editing in the pre-training step. It draws inspiration from diffusion models, which excel at handling incremental transformations in continuous data domains, generating high-quality and diverse images or audio through a gradual multi-step denoising process [18]–[20]. Similarly, code editing involves making a series of iterative changes to fulfill the final requirements, making it naturally aligned with the step-by-step refinement principles of diffusion models. However, DivoT5 does not directly implement diffusion processes. It utilizes its data generation principles and focuses on simulating the gradual code editing steps that developers typically follow, allowing it to produce realistic and contextually relevant code modifications. To distinguish it from traditional diffusion methods, we refer to this approach as "directional diffusion."

Figure 1 illustrates the schematic of directional diffusion. We set the starting point (i.e., X_T) of the directional diffusion strategy as the old code, design three mask strategies as introducing artificial noise (i.e., X_{T+1}) to understand the evolutionary direction, and treat the intermediate states (i.e., X_t) of the editing process as evolutionary noise that reinforces the direction of evolution. Specifically, X_{T+1} to X_T represents the process of removing artificial noise (e.g., replacing p_1 and v_1 to $|unk|$ in Figure 1), while X_t to X_{t-1} represents the process of removing evolutionary noise (e.g., modifying the f_{11} in the Figure 1). **To fit the fine-grained editing scenarios**, DivoT5's pre-training takes the old code as input and incorporates token-level *diff* information when generating the complete new code. Rather than relying on line-level *diff* information to understand code changes, DivoT5 leverages token-level *diff* information to help the model discern which parts of the code need modification and which parts should remain unchanged (i.e., a mask strategy applied from X_T to X_{T+1} in Figure 1). **To leverage and enrich the context of the real-world code edits**, we design two additional mask strategies based on real code change histories. Learning from real code changes means that the model needs to automatically update old versions of code (which may contain bugs or require new features) to new versions that meet new requirements. These two masking strategies (applied from X_T to X_{T+1} in Figure 1) simulate potential issues with code quality or completeness in real code context by introducing varying degrees of random artificial noise into the old code. This requires the model to understand the context of the specific code and make modifications based on the provided guidance, rather than merely restoring the original version from the corrupted code. **To learn the characteristics of incremental code evolution**, we design a novel evolutionary direction reinforcement strategy by utilizing intermediate data in the code editing process. Intermediate versions (i.e., X_t $(0 < t \leq T)$ in Figure 1) of the code that is not fully modified are seen as still containing evolutionary noise. We utilize intermediate data from the incremental evolution process to train the DivoT5 model, providing clear directional and more

detailed guidance information to reinforce the evolutionary direction. Due to the limitations of applying existing diffusion models to discrete textual data, which typically use multi-step denoising at continued embedding levels and whose performance is constrained by the mapping between continuous embeddings and discrete texts, we apply the above strategies at the data level (i.e., conducting the diffusion process on data rather than embeddings). This model ensures the fluency and reliability of the generated results by directing each version towards the final version (X_0), even with the addition of artificial noise (i.e., constructing training samples $< X_t, X_0 >$ and generating X_0 from any given X_t, where $0 < t \leq T+1$). In other words, we adopt the encoder-decoder training framework [12] while incorporating the diffusion concept to emphasize the evolutionary direction of code editing. Rather than based on the full diffusion model structure, we draw on its style to enhance step-by-step refinement, ensuring the same computational efficiency as the encoder-decoder model [7], [8], [12] during training and inference.

To evaluate DivoT5, we fine-tune DivoT5 on three scenarios (i.e., NL-guided code editing scenario, code-only editing scenario, and non-editing scenario) using eight benchmark datasets. The evaluation results demonstrate that DivoT5 outperforms the existing state-of-the-art (SOTA) code pre-trained models for all eight datasets. For example, DivoT5-base (220M) achieves a significant improvement in exact match performance compared to the CodeT5-base (220M) model, exceeding a 28.87% increase on the CodeReview dataset (from 34.46% to 44.41%). Furthermore, DivoT5-base (220M) significantly outperforms the fine-tuned CodeT5-large (770M) model and billion-scale models like DeepSeekCoder-6.7B-instruct, Meta-Llama-3.1-8B-instruct, and ChatGPT-gpt-3.5-turbo in few-shot inference. The small version of DivoT5 (60M) also achieves a good performance in this task, surpassing CodeT5-base (220M) by 5 percentage points (from 34.46% to 40.16%) in the exact match metric. In addition, we evaluate the effectiveness of each pre-training task. The ablation results demonstrate the contributions of the designed pre-training tasks to the final performance.

We summarize our contributions as follows:

- We propose the directional diffusion-style auto-regressive neural model, DivoT5, designed to simulate the characteristics of incremental code evolution carried out by human developers. **To the best of our knowledge, we are the first to introduce the concept of directional diffusion into the domain of code evolution**.
- We propose novel pre-training tasks that mimic human-like editing, enabling the model to capture fine-grained edits, track code evolution, and generate updated code across diverse contexts.
- We conduct extensive experiments to evaluate the DivoT5, and the results demonstrate its state-of-the-art performance on various benchmarks related to code evolution.

II. Background and Related Work

A. Code-Editing Tasks and Code-Editing Related Tasks

During the daily development process, code editing is necessary to maintain the overall functionalities of a software project. In this paper, we differentiate two categories of tasks that are associated with code editing. The first category is tasks involving an editing process to change a code snippet, and we refer to this category as code-editing tasks [3], [6], [8], [9], [13]. The second category is tasks that do not directly edit code but may use code-editing information for other purpose, and we refer to this category as code-editing related tasks [7], [14]. Obviously, it is helpful to automate both categories of tasks during the code maintenance process.

Based on whether natural language guidance is required during the code-editing process, we can further divide code-editing tasks into two types. We denote the first type as the **NL-guided code editing scenario**, such as code refinement [3], [7], [8]. Developers provide suggestions for improvement by inspecting and evaluating the code, and the natural language represents the direction for code enhancement. The new code should align with this direction. We denote the second type as the **code-only editing scenario**, such as code refinement [21]. For the code refinement task, the new code should address the defects present in the old code. In this paper, we primarily consider these two types of code-editing tasks.

B. Code Pre-trained Models

When applying existing pre-training techniques [22]–[27] to code data, there are two common pre-training tasks, which are referred to as masked language modeling (MLM) [12], [28] and denoising auto-encoding (DAE) [29]. The basic MLM technique refers to randomly masking a certain proportion of tokens in the input and then predicting the masked tokens. CodeBERT [28] and GraphCodeBERT [30] apply the MLM technique to its code data to enhance the model's understanding of code. CodeT5 [12] employs the MLM technique to sequence-to-sequence generation tasks (referred to as SMLM) to enhance the model's ability to generate code. Additionally, to recognize identifiers within the code, CodeT5 also introduces a variant of SMLM called IMLM (Identifier Masked Language Modeling). In IMLM, only the identifiers in the source code are masked, while the rest of the tokens are left unchanged, allowing the model to generate code while specifically paying attention to the discernment of identifiers. DAE is a self-supervised learning approach that enhances model generalization by introducing noise to the input and training the model to reconstruct the original input. In PLBART [29], DAE is introduced during the pre-training stage to train the model in code understanding and generation capabilities. Specifically, PLBART incorporates three noise functions: token masking, token deletion, and token infilling. These functions are applied with a certain probability to disrupt the old code.

C. Code Editing Pre-trained Models

Code editing pre-trained models are typically initialized from existing code pre-trained models but are further adapted

to specialize in code editing tasks. These models aim to better capture the structure and semantics of code transformations by introducing edit-aware pre-training objectives. Based on code pre-trained models, code editing pre-trained models use the following two ways to consider code edits: **MLM with code changes** and **DAE with controlled noises**.

CodeReviewer [7], CCT5 [14], and Coeditor [15] use **MLM with code changes**. The basic idea of these models is to use a sequence to represent a real-world code change. That is, the output of the *diff* tool when comparing the two code snippets is treated as the sequence for the code change. If there is a comment associated with the code change, the comment is also appended to the sequence. Note that the output of the *diff* represents the line-level differences on two code snippets. On top of the sequences representing code changes, CodeReviewer [7] and CCT5 [14] use the standard MLM for pre-training, where the masked parts can be tokens in the following types: tokens to be inserted, tokens to be deleted, tags generated by *diff*, and tokens in the comment. Different from CodeReviewer, CCT5 also adopts GPT-like pre-training tasks and adds data flow information into the sequence representing a code change. Similarly, Coeditor [15] first uses placeholders to mark the sections that need modification, then predicts the next *diff* information based on multiple prior line-level code changes. Our DivoT5 differs from CodeReviewer, CCT5, and Coeditor in two folds. First, DivoT5 uses token-level difference information for pre-training, while CodeReviewer and CCT5 are based on line-level differences. Second, the pre-training tasks in DivoT5 aim to predict code edits on top of the old code (i.e., ensuring evolutionary direction of code editing), thus well aligned with downstream code-editing tasks. However, the pre-training tasks in CodeReviewer, CCT5, and Coeditor do not correspond to any code-editing tasks. For example, the prediction of masked tokens when pre-training CodeReviewer and CCT5 may use information in both code snippets before and after change.

CoditT5 [8] and CodeEditor [13] use **DAE with controlled noises**. The basic idea of these models is to control noise generation in order that the generated noises can represent the probabilities real-world changes. In particular, CoditT5 [8] utilize the CodeSearchNet [31] dataset to design a denoising task for pre-training. Specifically, they calculate the probabilities of inserting, deleting, and modifying tokens in real-world code changes, and use these probabilities to artificially perturb the code. Then the standard DAE tasks are adopted for pre-training on top of the perturbed code. Similar to CoditT5, CodeEditor [13] is also based on code perturbed with probabilities from real world. However, the perturbed code is not directly used in pre-training. CodeEditor utilizes an existing code-generation model [21] to further process the perturbed code, and the processed code is then used for pre-training. Thus, the pre-training may well focus on the inserted noises that can hardly be handled by existing models. Our DivoT5 differs from these models in that DivoT5 is based on predicting real-world code edits while CoditT5 and CodeEditor are based on removing artificially inserted noises. Note that, although CoditT5 and CodeEditor use real-world probabilities to control the insertion of noises, the perturbed code may still largely deviate from real-world human-written code. Furthermore, DivoT5's pre-training tasks also incorporate noise to perturb the old code, but the aim is to help DivoT5 learn to predict code edits in the presence of noise.

D. Diffusion Models for Text Generation

Diffusion models, which predominantly operate at the continuous embedding level, have made remarkable progress in continuous domains such as image generation [18]. These models involve two key steps: forward noising process and reverse denoising process [32]. Specifically, the forward noise addition process gradually transforms the original data into pure Gaussian noise using a Markov chain, with each noise addition step represented as:

$$q(\mathbf{X_t}|\mathbf{X_{t-1}}) = \mathcal{N}\left(\mathbf{X_t}; \sqrt{1-\alpha_t}\mathbf{X_{t-1}}, \alpha_t\mathbf{I}\right)$$

, where hyperparameter α_t represents the noise added at diffusion step t. The reverse denoising process gradually removes the noise to generate the target data, with each step represented as:

$$p_\theta(\mathbf{X_{t-1}}|\mathbf{X_t}) = \mathcal{N}\left(\mathbf{X_{t-1}}; \mu_\theta(\mathbf{X_t}, t), \Sigma_\theta(\mathbf{X_t}, t)\right)$$

, where μ_θ and Σ_θ are learned from the model.

Existing works applying diffusion models to discrete text data introduce an additional step in the noise addition process, requiring the conversion of discrete text $\mathbf{W}$ into continuous embeddings (i.e., $q_\phi(\mathbf{X_0}|\mathbf{W}) = \mathcal{N}(EMB(\mathbf{W}), \alpha_0\mathbf{I})$). Correspondingly, the denoising process includes a trainable rounding step to convert embeddings back into discrete text (i.e., $p_\theta(\mathbf{W}|\mathbf{X_0}) = \prod_{i=1}^{N} p_\theta(W_i|X_i)$) [19], [20], [33]. For example, in the upper part of Figure 2, traditional diffusion performs a left-to-right denoising process. The model progressively denoises a continuous embedding and ultimately converts the continuous embedding into discrete token representations through a rounding layer, generating the final text (e.g., 'ABC'). Similarly, during the inference process, traditional diffusion also progressively denoises a continuous embedding. Notably, this embedding represents the entire text, and each denoising step generates a new embedding, which is then converted into all tokens of the text (i.e., in a non-autoregressive manner). As a result, every token in the text has the potential to change at each step of the inference process (e.g., from pure noise to 'fdc', from 'fdc' to 'aGh', etc.). This implies that achieving coherence and logical consistency across all tokens often requires many iterative steps.

Our approach diverges from existing methods that apply diffusion to discrete text in two key ways. First, rather than adding noise at the continued embedding level and iteratively removing it, we simulate real-world code editing by leveraging actual code evolution paths. For example, in Figure 2, we treat the partially modified text during the evolution process as "noise" (e.g., CbDaEa), while the target of the evolution is considered the "noise-free" text (e.g., CbDbEb). This approach assigns a concrete meaning to each instance of "noise", effectively simulating real-world editing scenarios. Second, we implement a diffusion-style process by utilizing

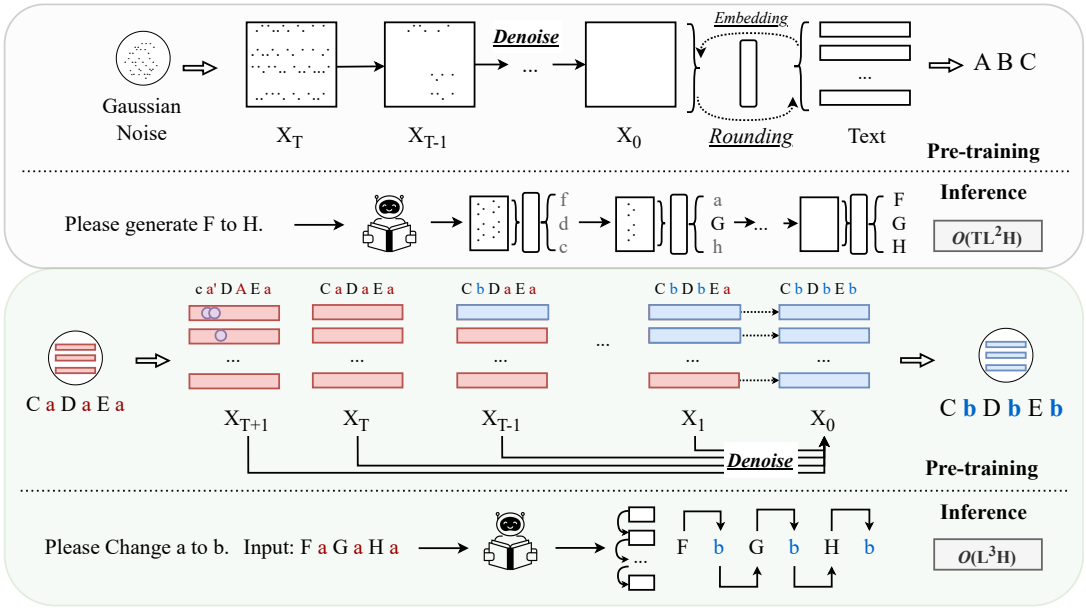


Fig. 2. Illustration of traditional diffusion and directional diffusion in text generation. The upper section showcases traditional diffusion, where 'ABC' is generated during pre-training, and 'FGH' during inference. The lower section highlights directional diffusion, showing 'CbDbEb' generated from 'CaDaEa' during pre-training, and 'FbGbHb' from 'FaGaHa' during inference.

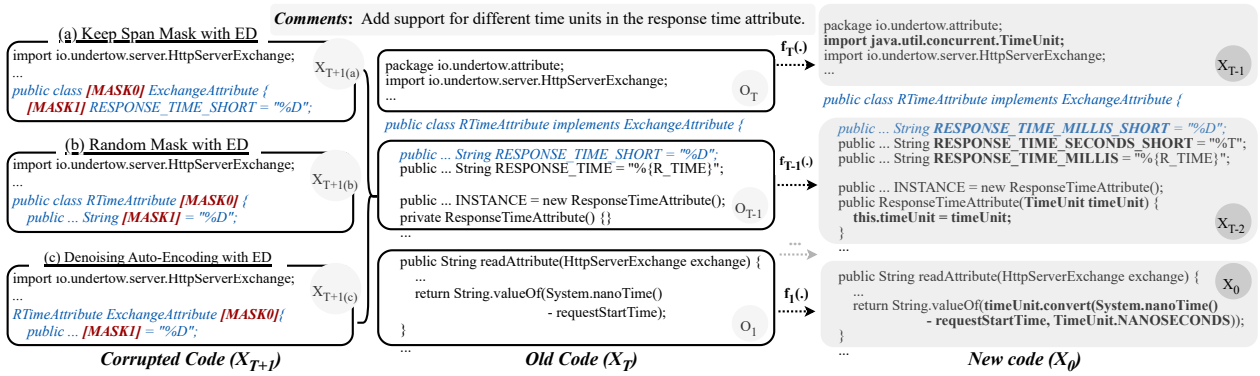


Fig. 3. The process of modifying the old code (center) to fulfill the requirements outlined in the comments (top left) often necessitates changes across multiple dispersed locations. Achieving the desired evolution (right) requires incremental editing at all relevant points. The blue italics aim to highlight the parts that persist throughout the entire code editing process.

the intermediate state data from the editing process to generate the final noise-free text, where each "denoising" step is treated as a transition from a noisy state to a noise-free state. This ensures that we can generate directly usable code in a single step—without multiple denoising iterations—at a computational cost comparable to standard basic models (e.g., in an autoregressive manner like Transformer [34], CodeT5). Specifically, the inference time complexity of traditional diffusion applied to text is about $O(TL^2H)$, while combining the diffusion concept with traditional text generation models results in a complexity of $O(L^3H)$, where T is the number of diffusion steps, L is the sequence length, and H is the hidden layer dimension. Typically, T is significantly larger than L (e.g., a sequence of 200 tokens may require up to 2000 denoising steps). Thus, traditional text generation requires less inference time [19] (The detailed information in SectionVI).

III. DivoT5

DivoT5 is a directional diffusion-style pre-trained model designed to simulate real-world code evolution. The model leverages the diffusion concept at the data level to capture the incremental nature of code editing and completes specific editing tasks.

A. Model Architecture

We employ an encoder-decoder framework with an auto-regressive manner based on the Transformer [34] model for pre-training. Following the existing code editing pre-trained models [7], [8], [14], [35], we continue pre-training DivoT5 on top of the weights of CodeT5 [12]. This enables the model to leverage CodeT5's knowledge, offering significant advantages in code-related tasks [8], [36], [37].

B. Input Representation

To accommodate different types of code evolution tasks, our DivoT5 accepts two types of input: input containing only code and input containing both natural language and code. We describe the input of DivoT5 in the following form:

$$x = \{[CLS], w_1, w_2, \cdots, w_K, [SEP], x_1, x_2, \cdots, x_M, [SEP]\}$$

, where $w_1, w_2, \cdots, w_K$ represent tokens in natural language, and $x_1, x_2, \cdots, x_M$ represent tokens in the input old code. $[CLS]$ denotes a special token indicating the start of the input, and $[SEP]$ denotes a special separator between different types of inputs. Similarly, for data that consists solely of code, the input can be simplified to the following form: $x = \{[CLS], x_1, x_2, \cdots, x_M, [SEP]\}$.

C. Directional Diffusion

We are the first to introduce the concept of directional diffusion into the domain of code evolution. To better align with the incremental modification characteristic of code evolution, we design a directional diffusion strategy to provide explicit evolutionary direction for the code editing process. Figure 3 shows a specific code example that applies the diffusion concept.

1) Incremental Modification in Code Evolution: As shown in Figure 3, modifying the old code (in the middle) to meet the requirements described in the comments (at the top left) may involve changes across multiple, potentially distant locations. Successfully completing this evolution (at the right) requires incremental editing at all relevant points. In this example, to add support for different time units, the first step is to import the necessary package (i.e., import the *TimeUnit* package). The next is to redefine variables and implement the corresponding function bodies at the appropriate locations (i.e., modifying variable *RES_T_SHOT* and implementing function *RTime*). The final step is to review and modify all code sections related to the new requirement (e.g., change the return value of other functions). Therefore, the code editing process is not a one-off task but requires gradually modifying all relevant code snippets to meet new requirements. This incremental modification characteristic aligns well with the step-by-step denoising process in diffusion, which prompts us to combine the concept of diffusion with code editing and evolution. In the process of code evolution, each modification can be seen as a "denoising" step on the old code. By applying the diffusion concept to code editing, we are able to simulate a dynamic and step-by-step modification process, allowing the code to incrementally approach the target state with a series of iterations. This strategy not only captures the subtle changes in code evolution but also provides a systematic way to track and predict the improvement path of the code.

2) Diffusion for the Evolutionary Direction: Traditional diffusion models handle continuous data (e.g., images) by applying noise to continuous embeddings. Existing diffusion models for discrete text generation typically learn text dependencies through a multi-step denoising process at a continuous high-dimensional embedding level and then map these embeddings to natural language text via a rounding process (e.g., the upper part of Figure 2). However, compared to natural language, code has a stronger syntactic structure and semantic relationships. Traditional diffusion models might be capable of generating coherent and diverse natural language text through denoising, but generating correct code is much more challenging than generating plain text. Furthermore, the real-world code context and code editing process in code evolution scenarios are more difficult to learn using traditional diffusion models.

Considering the above challenges, DivoT5, designed for discrete data like code, employs discrete-data-level diffusion, where each noise represents a specific code change, reflecting the natural evolution of code. This strategy allows the entire approach to be completed in an auto-regressive manner as well as to leverage the advantages of existing code-related approaches [7], [8], [12], [14], [38], [39]. In the noising stage, we first introduce artificial noise into the old code to learn token-level code editing processes and enrich the context of the old code. Then, we treat the intermediate evolution process of editing as a special type of evolution noise to enhance the direction of code evolution. Noisy data (artificial or evolutionary) can be paired with the final new code to form input-output pairs, enabling the model to learn the direction of evolution from different contexts. In the denoising stage, we treat different types of noise as distinct pre-training tasks, and denoising is performed fully in an auto-regressive manner. This enables the model to consider both the contextual dependencies of code and the incremental characteristics of code evolution. Below, we discuss the noising process at the data level and the denoising process in an auto-regressive manner.

a) Noising process at the data level: Code evolution typically requires referencing an existing code, making the starting point of directional diffusion not pure Gaussian noise but an old code that needs modification (X_T). On this basis, we add artificial noise to perturb the old data (X_{T+1}), simulating the basic editing process and enriching the context of the old code. This approach aims to help the model grasp fine-grained code edits and accurately generate the target new code across various contexts. We use X_t to represent an intermediate state in the evolution process, indicating that t modifications are still needed to complete all editing operations. For example, in Figure 3, X_{T-1} represents the state where the model has modified the block O_T in the old code X_T (e.g., adding the import of a package). Similarly, X_{T-2} represents the state where the model has modified the block O_{T-1} in X_{T-1} (e.g., changing variables and implementing function bodies). To obtain the intermediate version X_t, we compute the *diff* between the old and new code files using Python's `difflib` library, and then split the *diff* into hunks based on the relative positions of the changes within the code. Specifically, if transforming old code A ('CaDaEa') to new code B ('CbDbEb') involves three changes (each change representing a *diff* hunk that may affect several nearby lines), the intermediate code evolves incrementally: first modifying the first position to produce 'CbDaEa', then applying the second change to produce 'CbDbEa'. These intermediate versions follow the edit history sequence, progressing from the beginning of the

code to its end and mirroring typical code editing practice. We treat the code that has undergone partial human modifications and still requires further modifications as a sample containing evolutionary noise. By gradually adding artificial noise and evolutionary noise (human edits) during the code evolution process, we can effectively simulate the transition of code from its evolved state (i.e., X_0) to the corrupted state (i.e., X_{T+1}). The process of adding noise can be formalized as follows:

$$q(X_t|X_0) = \sum_{i=1}^{t} f_i(X_{i-1})$$

, where $q(X_t|X_0)$ represents the process of gradually adding noise from X_0 to X_t, and $f_i(X_0)$ is a i-th function representing the addition of noise to X_{i-1} to obtain X_i. For example, in Figure 3, f_1 represents the operation of reverting the modification to the O_1 block in the evolved code, indicating that there is still a part of the code that needs to be modified. Each X_t represents the state of the code at a specific stage of evolution, providing explicit direction that helps the model approach the final state at different modification stages (i.e., constructing the input-output pairs like $< X_t, X_0 >$). This strategy improves the model's adaptability to diverse contexts and enhances its ability to capture subtle changes in the code.

b) Denoising process in auto-regressive manner: Existing diffusion models for discrete text generation typically add and remove noise at the continuous embedding level step by step. While this can increase the diversity of the generated text, it requires more iterative steps, making the process slower compared to the mainstream auto-regressive text generation approaches, resulting in lower efficiency. For example, in the upper part of Figure 2, each denoising step in traditional diffusion models modifies the overall embedding, allowing every token to potentially change in the next step, which facilitates generating diverse text. In contrast, in autoregressive generation, once a token is generated, its content and position are fixed, and the generation continues sequentially from that point onward, facilitating semantic coherence in the output [12], [34]. Therefore, we conduct the denoising process within an auto-regressive manner similar to CodeT5, using any intermediate state X_t from the diffusion process as input and producing the final target X_0 as output. This approach maximizes the generative performance benefits of auto-regression. The denoising process can be formalized as:

$$p_\theta(X_{t-1}|X_t) = \prod_{i=1}^{N} p_\theta(X_{t-1}^i|X_{t-1}^{1:i-1}; X_t)$$

, where $p_\theta(X_{t-1}|X_t)$ represents the probability distribution of denoising from X_t to X_{t-1}, and $p_\theta(X_{t-1}^i|X_{t-1}^{1:i-1}; X_t)$ represents the probability of generating the i-th token in X_{t-1}, given the existing tokens in X_t and generated tokens in X_{t-1}. The core characteristic of directional diffusion is reflected in each generation process of the auto-regressive model, where the generation target at any step is the final evolution goal. In other words, we transform intermediate diffusion data into input-output pairs for the auto-regressive model, such as $< X_{T+1}, X_0 >$, $< X_T, X_0 >$, $< X_{T-1}, X_0 >$. Under this strategy, each sample in the generation process is directional and optimized towards the final new code, thereby enhancing the model's accuracy and consistency in generating outputs. Notably, by converting the directional diffusion process into input-output pairs for the pre-training task, our pre-trained model has the same inference cost as auto-regressive models like CodeT5 (requiring a single inference pass), which is lower compared to the higher inference cost of traditional diffusion models (requiring multiple inference passes).

D. Directional Diffusion Pretraining Tasks

DivoT5 is designed to acquire real-world code editing knowledge by learning to transform old code into completely new code through four directional diffusion-based pre-training tasks. To better illustrate the pre-training tasks, let us assume that the model only has the old code $\mathbf{x}$ $(x^1, x^2, \cdots, x^M)$ as the input and the new code $\mathbf{y}$ $(y^1, y^2, \cdots, y^N)$ as the output, where M represents the number of input tokens and N represents the number of output tokens. Figure 3 shows an illustration of the four tasks, where the corrupted code on the left (X_{T+1}) represents examples of the first three tasks, and the intermediate states X_t $(0 < t \leq T)$ from the old code (X_T) in the middle to the new code (X_0) on the right illustrates the input for the last task. For clarity, we refer to the input code as the old code and the output code as the new code, and use blue italics to highlight the parts of the code that persist throughout the entire editing process. The first three tasks are designed to learn the basic editing processes and contextual dependencies of the code during the diffusion process, while the final task aims to reinforce the direction of code evolution. We place an additional special token at the prefix of the input sequence to distinguish between different tasks during the training process. Below we discuss the details of these tasks.

a) Keep Span Mask with Evolutionary Direction: This task aims to make the model aware of which code tokens do not need to be modified (i.e., Keep Span) during the code editing process. The design of this pre-training task leverages the observation that real-world code edits often retain a significant portion of the original structure and content. Since the code before and after evolution typically follows similar patterns or structures, some tokens remain unchanged across versions. For example, as shown in $X_{T+1(a)}$ of Figure 3, $[MASK1]$ represents the keyword that defines a variable (i.e., 'public'), which commonly remains unchanged during edits. To leverage this property, we introduce a new masking strategy that explicitly masks unchanged tokens from real code edits, grounding the learning objective in the evolutionary direction. Unlike generic masking strategies over static code, our task is applied to real-world edit pairs, where each instance reflects a concrete transformation from an earlier version to a later one. By anchoring the keep span mask in this real edit context (i.e., evolutionary direction), we encourage the model to learn not only what to change but also what to preserve. This provides a valuable complementary learning signal: recognizing stable code regions is essential for generating accurate, minimal edits and avoiding unnecessary modifications.

Following existing works [8], [40], we calculate the differences between the new code and the old code. However, unlike

existing techniques [7], [14], our difference information is at the token level rather than at the line level. Then, we apply the SMLM [41] task specifically to the tokens marked as "KEEP" in the difference information. The formal representation of this task is as follows:

$$\mathcal{L}^{\theta}_{KSM_{ED}} = -\sum_{i=1}^{N} log\mathcal{P}_{\theta}(y_i|(\mathbf{x}^{mask_{KSM}}_{keep}; \mathbf{x}_{change}; \mathbf{y}^{1:i-1}))$$

, where i represents the current generation step, y_i represents the i-th token in the new code y. The $mask_{KSM}$ represents a type of SMLM approach, x_{keep} indicates the unchanged parts of the code before and after editing, and x_{change} indicates the modified parts. Since the number of unchanged tokens in this task is smaller than the total number of tokens in the old code, we mask 30% of the unchanged tokens, aiming to achieve a similar total number of masked tokens as in other tasks.

b) Random Mask with Evolutionary Direction: This pre-training task treats the code editing process as a text generation task with evolutionary direction. Unlike the previous task, which focuses on unchanged regions, it treats all tokens equally by randomly masking token spans in the old code and learning to generate the completely new code through a sequence-to-sequence framework. Based on the characteristics of the dataset and existing works [7], [8], we set the average span count to 2.5 and the overall percentage of masked tokens to be 20%. Unlike self-supervised tasks in the field of general text generation, the goal of this pre-training task is not simply to predict masked words, but rather to generate edited code in the presence of masks (i.e., prediction of the evolutionary direction). Formally, the loss of this task can be described as follows:

$$\mathcal{L}^{\theta}_{RM_{ED}} = -\sum_{i=1}^{N} log\mathcal{P}_{\theta}(y_i|\mathbf{x}^{mask_{RM}}; \mathbf{y}^{1:i-1})$$

, where $mask_{RM}$ represents the random mask for the old code.

c) Denoising Auto-Encoding with Evolutionary Direction: Our third pre-training task combines the characteristics of denoising auto-encoding with the real-world code editing scenarios. It aims to train the model to generate the new code accurately while taking into account the noise and variations that occur in old code. Compared to the previous two pre-training tasks, this task demands a stronger ability for the model to understand code editing. However, this task well aligns with real-world development scenarios because we often cannot guarantee the quality of the old code. It reflects the practical nature of code editing, where the quality and effectiveness of the existing code vary. Specifically, we apply three types of noise to the old code: insertion, deletion, and replacement [8]. As shown in the code snippet $X_{T+1(c)}$ of Figure 3, in the old code, we add a $[MASK0]$ token, delete the keyword tokens like $public$, and replace specific variable names with $[MASK1]$. These modifications introduce noise and variations to simulate real-world code editing scenarios. During the generation process, the model not only needs to consider the tokens replaced by $[MASK]$, but also needs to take into account the scenario where existing tokens are deleted and the case of useless masks. Unlike existing techniques [8], [13] that utilize DAE with controlled noises to restore the old code, our approach requires the model to generate the actual new code after removing the introduced noise. The loss function for this pre-training task is as follows:

$$\mathcal{L}^{\theta}_{DAE_{ED}} = -\sum_{i=1}^{N} log\mathcal{P}_{\theta}(y_i|\hat{\mathbf{x}}; \mathbf{y}^{1:i-1})$$

, where $\hat{\mathbf{x}}$ represent old code with noise.

d) Evolutionary Direction Reinforcement: This task aims to reinforce the direction of code editing during the real evolution process. We use the final version of the code X_0 as the new code to be generated and any intermediate version of the code X_t (where $0 < t \leq T$) during the evolution process as the old code input to the model.

By incrementally adding details from the old code to the edited code, the model learns the nuances of gradual code modifications and gains the ability to achieve the final desired state of the code. This task can be formally described as follows:

$$\mathcal{L}^{\theta}_{EDR} = -\sum_{i=1}^{N} log\mathcal{P}_{\theta}(y^i|\mathbf{x}; \mathbf{y}^{1:i-1})$$

, where N represents the number of tokens contained in the new code X_0, y^i represents the generation to the i-th token, and x represents an intermediate version of evolution X_t.

1) Final Loss: We treat each pre-training task equally and add up the loss values of each task to obtain the overall loss. We describe the final loss function as follows:

$$min_{\theta}\mathcal{L}^{\theta}_{Loss} = \mathcal{L}^{\theta}_{KSP_{ED}} + \mathcal{L}^{\theta}_{RM_{ED}} + \mathcal{L}^{\theta}_{DAE_{ED}} + \mathcal{L}^{\theta}_{EDR}$$

E. Cost of Training and Inference

We design a directional diffusion strategy at the data level to capture the gradual nature of code editing, transforming diffusion concept into pre-training tasks and direction-guided samples. Then, we train an auto-regressive model, similar to CodeT5, using tasks and samples that incorporate evolution direction to enhance editing capabilities. Therefore, both training and inference consumption are comparable to CodeT5, operating in a single-pass auto-regressive mode, without requiring multiple denoising iterations per sample like traditional diffusion models.

IV. Experimental Setup

In this section, we first introduce the research questions (RQs). Then, we introduce the downstream tasks, evaluation metrics, baseline models, and fine-tuning details used to evaluate DivoT5.

A. Research Questions

To evaluate the effectiveness of DivoT5, we ask the following questions:

a) RQ1: How well does DivoT5 perform compared with the existing code editing pre-trained models?: To answer this question, we compare DivoT5 with existing code editing pre-trained models and billion-scale models across multiple downstream tasks.

TABLE I
STATISTICS OF PRE-TRAINING DATASETS.

Dataset Name	#Project	#Data
CodeChangeNet [14]	35k+	1.5M+
CommitPack-Java [44]	130k+	3.7M+
Total	165k+	5.2M+

b) RQ2: How dose DivoT5 generalize to non-editing scenarios?: To answer this question, we conduct experiments on a code translation task, which differs from the code editing scenarios, to evaluate DivoT5's generalization ability.

c) RQ3: How do the proposed pre-training tasks affect the model performance?: To answer this question, we pre-train various ablated versions of DivoT5 by removing one pre-training task each, in order to evaluate the effect of each pre-training task. We conduct these ablation studies on the 60M-parameter variant of DivoT5 to enable efficient experimentation under limited computational resources, following a common practice in prior works [42], [43].

B. Pre-training datasets

We use real code editing data to train DivoT5, using a code change dataset collected from the open-source repository Github, including CodeChangeNet [14] and CommitPack [44]. CodeChangeNet includes six popular programming languages (i.e., Go, Java, JavaScript, PHP, Python, and Ruby), over 35,000 projects, and over 1.5 million commit data. CommitPack comprises 4TB of GitHub commit data, covering 350 programming languages. To conserve computational resources, we select the Java programming language from CommitPack for our experiments. Specifically, the Java dataset includes over 165,000 projects and more than 3.7 million data instances. Generally, more data can lead to better model performance.

Additionally, in the directional diffusion strategy, we focus not only on each data sample but also on the evolution path of that sample. Each intermediate evolution step points towards the final evolved version to reinforce the direction of code editing. Consequently, this strategy increases the amount of pre-training data to nearly 9 million data instances.

C. Downstream Tasks

Code editing tasks mainly involve the process of editing old code into new code. During the code editing process, there may be additional constraints that restrict the direction of evolution, such as adhering to natural language requirements. Therefore, the effectiveness of a code editing pre-trained model should be evaluated on two scenarios: the NL-guided code editing scenario and the code-only editing scenario. In particular, we focus on two tasks (i.e., the NL-based code refinement task [3], [7], [8] and the bug fixing task [11]) for the first scenario and the code refinement task [6], [21] for the second scenario. We further use the code translation task [21], [45], which is a non-editing task, for evaluation. Since code translation differs much from our pre-training tasks, the code translation task may help evaluate the generalizability of DivoT5. Table II presents the statistical information of the datasets related to these tasks.

1) NL-guided code editing scenario: **The NL-based code refinement task** focuses on improving existing code based on guidance provided in natural language. We first conduct experiments on the CodeReview dataset [3], which is written in Java, following the same data setup as *Zhang et al.* [8]. To further evaluate the model's capability, we also utilize a large-scale multilingual dataset [7], which contains nearly 180k samples spanning nine programming languages. **The bug fixing task** aims to fix the bug existing in the old code. We use the bug-fixing dataset with additional information [11], [46], such as developers' natural language guidance, to help the model fix bugs more accurately.

2) Code-only editing scenario: **The code refinement task** aims to convert defective code into correct one without natural language guidance. We use the Java dataset collected by *Tufano et al.* [6] to evaluate the model's performance on this task. This dataset comprises two versions of benchmarks: the Refine-small version with code tokens fewer than 50, and the Refine-medium version with code tokens ranging between 50 and 100. We follow the settings of CodeXGLEU [21] and use the same metrics to evaluate our model. To more comprehensively evaluate this scenario, we construct a variant of the CodeReview [3] dataset, referred to as CodeReview-CodeOnly, by removing the natural language input. This version retains the same number of samples as the original dataset but omits the NL comments from the input, enabling us to isolate and assess the model's performance in the absence of natural language guidance.

3) Non-editing scenario: **The code translation task** involves translating a program written in one programming language into another programming language. This task assists developers in code migration and cross-platform development using different programming languages. We use the CodeTrans dataset [21], [45] as our benchmark for this task, which includes Java and CSharp code samples, and evaluate it with the same metrics as the code refinement task.

D. Metrics

We follow the existing research in the field of code generation [21] and code evolution [8] and select four metrics to evaluate the generated code: Exact Match, BLEU, CodeBLEU, and SARI.

1) Exact Match (EM): The EM score [47], often used in natural language processing tasks such as question answering, evaluates the accuracy of model predictions by measuring the proportion of examples for which the predicted answer exactly matches the ground truth answer.

2) Bilingual Evaluation Understudy (BLEU): The BLEU [48] score, also commonly used in natural language processing, evaluates the accuracy of model predictions by measuring the ratio of n-gram matches between the generated answer and the ground truth answer. In practice, the BLEU-4 score is often reported, with a higher BLEU-4 score indicating better quality of the generated answer.

3) CodeBLEU (C-BLEU): The CodeBLEU [49] score, adapted from BLEU and designed specifically for programming language processing, evaluates the accuracy of model

TABLE II
STATISTICS OF THE DOWNSTREAM DATASETS WE USED.

Scenarios	Tasks	Datasets	# Train	# Dev	# Test
NL-guided Code Editing	NL-Based Code Refinement	CodeReview	13,753	1,719	1,718
		NL-CodeRefinement	150,406	13,103	13,104
	Bug Fixing	BugFixing-S	46,628	5,828	5,831
		BugFixing-M	52,324	6,542	6,538
Code-Only Editing	Code Refinement	Refine-S	46,680	5,835	5,835
		Refine-M	52,364	6,545	6,545
		CodeReview-CodeOnly	13,753	1,719	1,718
Non-Editing	Code Translation	CodeTrans	10,300	500	1,000

predictions by summing four weighted parts. The equation for calculating CodeBLEU is as follows:

$$\text{C-BLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}}$$

, where $BLEU_{weight}$ is the weighted $BLEU$, obtained by comparing the generated answer and the ground truth answer with different weights on different tokens. $Match_{ast}$ is the syntactic AST match. $Match_{df}$ is the semantic dataflow match. $BLEU_{weight}$ and $Match_{ast}$ are used to measure grammatical correctness, and $Match_{df}$ is used to calculate logic correctness.

4) System output Against References and Input (SARI): Evaluating code editing models requires not only output similarity but also an accurate assessment of how well the model performs actual edit operations [50]. The SARI [51] metric, originally proposed for text simplification, evaluates the quality of edits by comparing the system output against both the input and the reference. Specifically, it separately measures the precision and recall of three types of operations: additions, deletions, and keeps. Unlike similarity-based metrics such as BLEU or CodeBLEU, SARI explicitly focuses on the correctness of edit operations, making it more suitable for code editing tasks.

E. Baselines

Despite the availability of numerous large-scale code pre-trained models [42], [52], fine-tuning is challenging due to their massive parameter sizes. Therefore, we compare the fine-tuned performance of models with similar and larger parameter sizes and the few-shot performance of billion-scale instruct models. Since our approach draws on the diffusion style to simulate the step-by-step editing process in real-world development scenarios, we focus our comparisons on auto-regressive code models rather than traditional diffusion models. By referencing diffusion concepts to generate data for training, our approach adapts these principles within an auto-regressive framework, making direct comparisons with traditional diffusion models (i.e., non-autoregressive framework) outside the scope of this work.

1) Code pre-trained models: CodeT5 is a pre-trained model based on the encoder-decoder structure of T5 [25]. During the pre-training process, CodeT5 applies four pre-training tasks: masked span prediction, identifier tagging, masked identifier prediction, and bimodal dual generation. Identifier tagging is used to label whether a token is a code identifier, while the masked identifier prediction task is used to generate the masked identifier content. Experimental results demonstrate that CodeT5 can better capture code semantics and outperforms previous approaches on code-related tasks. In our evaluation, we use three variants of CodeT5: CodeT5-small (60M), CodeT5-base (220M), and CodeT5-large (770M). Compared to CodeT5, CodeT5+ introduces more pre-training data and tasks, such as span denoising, contrastive learning, and causal LM. In our evaluation, we use the 220M parameter version of CodeT5+ (which is of the same scale as DivoT5-base) for comparison. CodeLlama [53] is a family of decoder-only models pre-trained on large-scale corpora of both natural language and code, with variants specifically optimized for code understanding and generation. It has shown strong performance across a wide range of code-related benchmarks. In our experiments, we use the 6.7B base version of CodeLlama as a representative and competitive LLM baseline to compare against DivoT5, highlighting the advantages of our task-specific pretraining approach.

2) Code editing pre-trained models: In our evaluation, we use four code editing pre-trained models: CodeReviewer, CCT5, CoditT5, and CodeEditor. Among the four models, CodeEdit has 60M parameters, and the other three have 220M parameters. CodeReviewer is a pre-trained model on a real-world dataset in code change scenarios. The primary pre-training task of CodeReviewer is to utilize a masked language modeling approach to mask the sequence representing line level code differences, and then predict the masked content. Additionally, CodeReviewer also incorporates a pre-training task of generating commit messages from code differences to enhance the model's ability to generate text. CCT5 is a pre-trained model on a large dataset consisting of 1.5M+ pairwise data of code changes and commit messages from popular GitHub projects. In the pre-training process, CCT5 leverages the difference sequence of code and applies the SMLM task to learn the representation of code changes. For example, CCT5 first masks the information in the difference sequence at the granularity of code lines, then use other information from the difference and the commit message to predict the masked con-

tent. CoditT5 designs denoising pre-training tasks specifically for code editing. CoditT5 first calculates the probability of code edits in downstream tasks, including the probabilities of token insertion, deletion, and modification. Based on these probabilities, CoditT5 introduces noise to the old code. Finally, the model learns the code editing process by reconstructing the old code from the perturbed code. CodeEditor first disrupts the old code based on probabilities related to editing operations. Then, CodeEditor utilizes the existing generation model [21] to fill in the disrupted parts as input. Similar to CoditT5, CodeEditor also employs a denoising approach to recover the old code from the disrupted code.

3) Instruct-Models with billion-scale parameters: To more comprehensively evaluate the effectiveness of our model, we compare it against several billion-scale instruction-tuned LLMs that are widely used in code generation and editing tasks [42], [52], [54]. These include ChatGPT (gpt-3.5-turbo) [52], DeepSeekCoder-6.7B-Instruct [42], and Meta-Llama-3.1 (8B-Instruct) [54]. These models are known for their strong general-purpose code generation capabilities and are representative of current state-of-the-art instruction-tuned LLMs. Given the high computational cost of fine-tuning such large models on our task, we evaluate them in a few-shot setting instead. Following prior work [55], we design prompts that include scenario-specific descriptions of the editing task and use a low temperature to ensure consistent and accurate outputs. Specifically, we refine prompts several times based on [55], with a temperature of 0 for reproducibility, and choose a top-performing few-shot example from the training set after 5 attempts. All the prompts used in our experiments can be found in [56]. Additionally, since these models are not fine-tuned on the training set, to ensure a fairer comparison, we remove the code style elements such as casing, line breaks, and comments from both the model-generated results and the ground truth in the test set. However, tasks related to code editing have strong domain-specific characteristics, which often limit the performance of these models.

F. Training Details

In the pre-training phase, we employ the popular PyTorch [57] framework for model pre-training on the Linux Ubuntu operating system. Our pre-training process relies on 4 NVIDIA GeForce RTX 3090 GPUs, with each GPU having a memory capacity of 24GB. Additionally, our system has a memory capacity of 251GB. In the small version of pre-training on CodeT5 with 60M parameters, we use a learning rate of 3e-4. For the base version of pre-training on CodeT5 with 220M parameters, we use a learning rate of 2e-4. To expedite the model's pre-training process, we employ the Accelerate [58] and DeepSpeed [59] frameworks with mixed-precision BF16 to improve the training speed.

During the fine-tuning process for downstream tasks, we maintain the same system environment and acceleration settings as in the pre-training phase. We set length restrictions for the inputs and outputs of each task based on a statistical analysis of the dataset samples and existing works [7], [8], [21]. All models use the same length settings for the same dataset.

V. Results

A. RQ1: Effectiveness of DivoT5

We evaluate the performance of DivoT5 on two types of code editing scenarios (i.e., NL-guided code editing scenario and code-only editing scenario) across seven specific datasets, comparing it with existing code pre-trained models. If the model has already been evaluated on the target datasets, we directly report the performance metrics from the original paper. Otherwise, we download the released model from HuggingFace [60] and use the official code to evaluate it on the corresponding datasets. For models whose checkpoints or code are not publicly available, we only include a comparison with our results on the same dataset, and mark missing entries with a '-' in the table. We use the symbol $^{\alpha}$ in the result tables to indicate models whose performance is not statistically significantly different from DivoT5, with both DivoT5 and the corresponding baselines jointly marked to reflect their comparable results.

1) NL-guided code editing scenario: Table III presents a comparison of the performance for the NL-guided code editing scenario, emphasizing the effectiveness of DivoT5. It can be observed that the DivoT5-base model demonstrates significant improvements across most datasets compared to all other models, including 60M, 220M, 770M, and billion-scale parameters.

For the NL-based code refinement task, DivoT5-base achieves a substantial improvement over CodeT5-base, with a 29% relative gain in the exact match metric (from 34.46 to 44.41), and consistently stronger results across BLEU and SARI. Notably, the smaller DivoT5-small model even surpasses models with 220M parameters in all evaluation metrics, demonstrating the effectiveness of our pretraining objectives even at a smaller scale. Compared to other pre-trained editing models such as CoditT5 and CodeReviewer, DivoT5-base also achieves over 18% higher performance in the exact match metric, further validating its superior ability to model real-world code edits. In the large-scale, multilingual NL-CodeRefinement dataset, which spans nine programming languages, DivoT5-base continues to outperform strong baselines. It achieves 32.75 in exact match and 72.09 in SARI, both improvements over CodeT5-base, which reaches 29.44 and 70.61 respectively. These results highlight DivoT5's robustness and generalization capability across diverse codebases and natural language annotations. Since the fine-tuned weights of CodeT5-base from [7] are unavailable, we re-fine-tuned CodeT5 on this dataset for 13 epochs over 44 hours and 52 minutes until performance plateaued on the validation set.

Figure 4 depicts a case study showcasing different models' outputs on the CodeReview dataset. To save space, we only compare the outputs of models with a parameter count of 220M and display the differences between the new code and the old code. In the Figure 4, the old code's functionality checks for null in the return value. The review comment suggests using "ReviewCategoryStrategy.NONE" instead of returning a null value. Specifically, the reviewer recommends replacing the return value in the last line of the old code but not changing the null value used for checks. The CodeT5

TABLE III
RESULTS OF THE NL-GUIDED CODE EDITING SCENARIO (THE FS REPRESENTS FEW-SHOT MANNER).

Models	**CodeReview**			**NL-CodeRefinement**			**BugFixing-S**			**BugFixing-M**		
Metrics	**EM**	**BLEU**	**SARI**	**EM**	**BLEU**	**SARI**	**EM**	**BLEU**	**SARI**	**EM**	**BLEU**	**SARI**
CodeT5-small (60M)	32.25	85.86	69.79	28.07	78.06	69.82	33.10	73.97	58.45	25.45	85.01$^{\alpha}$	55.62
CodeT5-base (220M)	34.46	86.21	70.94	29.44	78.64	70.61	36.84	75.45	59.85	30.47	85.68$^{\alpha}$	57.10
CodeT5-large (770M)	34.63	86.10	71.28	29.81	79.17$^{\alpha}$	71.07	36.55	75.50$^{\alpha}$	59.56	29.83	**85.86**	57.03
CodeT5+ (220M)	32.36	85.99	69.71	30.24	78.64	71.07	34.27	73.88	58.87	25.19	84.66	55.22
CodeEditor (60M)	29.40	-	-	-	-	-	-	-	-	-	-	-
CoditT5 (220M)	37.19	86.58	70.33	28.95	78.98$^{\alpha}$	70.62	37.52	75.39	59.72	29.96	85.75	56.35
CCT5 (220M)	36.96	87.65	70.25	29.66	78.55	70.48	36.72	74.95	59.64	30.15	84.78	57.02
CodeReviewer (220M)	37.14	87.30	71.21	30.67	78.96$^{\alpha}$	71.49	36.36	74.28	59.57	31.16	85.34$^{\alpha}$	57.05
CodeLlama (6.7B-base)	31.49	85.19	67.44	31.65	79.07$^{\alpha}$	68.64	**40.44**	74.36	**60.94**$^{\alpha}$	29.40	83.91	56.62
DeepSeekCoder-FS (6.7B-instruct)	26.83	76.17	66.26	15.78	70.04	60.49	2.68	43.76	47.59	2.26	59.63	46.31
Meta-Llama-3.1-FS (8B-instruct)	30.15	71.39	66.91	18.28	65.45	61.59	2.30	34.02	47.25	2.26	47.54	46.14
ChatGPT-FS (gpt-3.5-turbo)	34.81	82.05	68.39	21.99	**80.47**	65.24	2.42	54.98	48.07	1.71	65.71	47.43
DivoT5-small (60M)	40.16	87.81	72.00	29.42	78.59	70.93	35.84	74.80	59.40	28.72	84.65	56.78
DivoT5-base (220M)	**44.41**	**88.54**	**73.51**	**32.75**	79.19$^{\alpha}$	**72.09**	39.08	**75.86**$^{\alpha}$	60.67$^{\alpha}$	**33.65**	85.24$^{\alpha}$	**58.12**

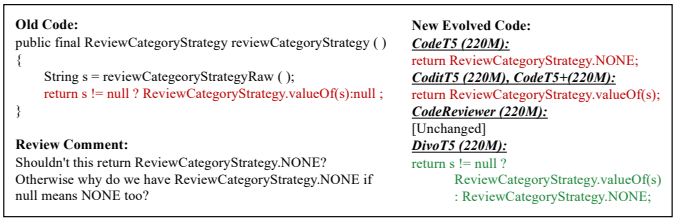


Fig. 4. Comparison of generation case using different models on the CodeReview dataset

model returns the "ReviewCategoryStrategy.NONE" value, incorrectly removes the necessary condition check. CoditT5 and CodeT5+ mistakenly delete the original null check, while the CodeReviewer model outputs the old code without making any modifications. In this example, DivoT5 is able to generate the evolved new code correctly. This could be attributed to the design of DivoT5's pre-training task, which enables the model to perceive which parts do not require modification (i.e., the KSM_{ED} pre-training task). Notably, despite being trained only on Java data from the CommitPack dataset due to resource constraints, DivoT5 achieves a higher EM score than the CodeReviewer model, which was trained on data from multiple programming languages.

Similarly, for the bug fixing task, DivoT5 outperforms all models of comparable size across most evaluation metrics. On the BugFixing-S dataset, CodeLlama (6.7B-base) achieves the highest scores with 40.44 in exact match and 60.94 in SARI, demonstrating strong performance on short, localized edits. However, DivoT5-base closely follows with 39.08 in exact match and 60.67 in SARI, despite having over 30× fewer parameters. This result underscores the effectiveness of our fine-grained edit-oriented pretraining in achieving competitive performance without relying on massive model scale. On the more challenging BugFixing-M dataset, which involves longer and more complex edits, DivoT5-base further strengthens its lead with 33.65 in exact match and 58.12 in SARI, outperforming all other models, including those with significantly larger parameter counts. While DivoT5-base shows a slight decrease in BLEU score compared to CodeT5-base in the BugFixing-M dataset, it increases EM by over 10%. Given that the EM metric is typically more crucial than the BLEU score in indicating correct bug fixes, this improvement is significant.

When compared to billion-scale models in few-shot settings, we observe that these models generally show limited performance in NL-guided code editing scenarios. This is because code editing tasks are highly domain-specific, and the few-shot approach often struggles to learn sufficient domain knowledge. This observation aligns with existing research findings [55]. Additionally, in NL-based code refinement, the natural language descriptions are more closely related to the actual code changes, resulting in better performance from few-shot models. However, for bug-fixing tasks, the natural language descriptions typically provide relevant context rather than directly corresponding to the actual changes, leading to poorer performance from these models.

2) Code-only editing scenario: Table IV summarizes the performance of DivoT5 and various baselines in the code-only editing scenario, which is evaluated across three representative datasets: Refine-S, Refine-M, and CodeReview-CodeOnly. In this setting, models are required to refine code without access to natural language guidance. Due to the substantial lexical overlap between old and new code versions, BLEU scores tend to be uniformly high and less discriminative. Therefore, we primarily emphasize the EM and SARI metrics, which more directly reflect the model's ability to perform accurate and meaningful edits.

On the Refine-S dataset, DivoT5-small (60M) performs on par with or better than all 220M-parameter baselines, achieving 22.31 in EM and 47.38 in SARI. The DivoT5-base model (220M) achieves the highest overall performance, with 23.02 in EM and 47.92 in SARI, surpassing CodeT5-base and CodeReviewer by over 6% in exact match. These results demonstrate that DivoT5 learns effective editing strategies even without natural language cues, outperforming larger models with fewer parameters.

On the more challenging Refine-M dataset, which involves longer and more complex code segments, DivoT5-base continues to lead with 16.55 in EM and 43.09 in SARI, improving upon CodeT5-base by 2.59 percentage points in EM. While BLEU scores remain high (around 88) for all models due

TABLE IV
RESULTS OF THE CODE-ONLY EDITING SCENARIO.

Models	Refine-S			Refine-M			CodeReview-CodeOnly		
Metrics	**EM**	**BLEU**	**SARI**	**EM**	**BLEU**	**SARI**	**EM**	**BLEU**	**SARI**
CodeT5-small (60M)	19.06	76.23	44.69	10.92	89.20	38.73	10.77	82.18$^{\alpha}$	38.27
CodeT5-base (220M)	21.61	77.43$^{\alpha}$	45.78	13.96	87.64	40.25	13.15	82.56$^{\alpha}$	39.54
CodeT5-large (770M)	21.7	77.38$^{\alpha}$	46.03	14.76	89.22	41.08	13.34	82.58$^{\alpha}$	39.87
CodeT5+ (220M)	22.18	**78.27**	46.87	15.13	88.64$^{\alpha}$	42.40	12.75	82.67$^{\alpha}$	39.42
CoditT5 (220M)	18.75	77.59$^{\alpha}$	44.61	13.00	**89.45**	40.15	14.09	83.13	41.04
CCT5 (220M)	17.87	77.02	43.25	9.75	89.06	38.21	13.56	82.31$^{\alpha}$	41.59
CodeReviewer (220M)	21.66	77.50$^{\alpha}$	45.89	12.61	88.92$^{\alpha}$	39.57	15.56	81.97	42.26
CodeLlama (6.7B-base)	20.17	77.24$^{\alpha}$	45.78	13.23	88.30$^{\alpha}$	40.96	14.90	80.52	43.59
DeepSeekCoder-FS (6.7B-instruct)	1.73	68.40	35.25	0.66	77.71	33.41	11.03	73.59	40.24
Meta-Llama-3.1-FS (8B-instruct)	1.32	39.11	35.34	0.58	59.93	33.33	12.14	74.28	40.37
ChatGPT-FS (gpt-3.5-turbo)	1.90	73.40	37.29	1.10	86.19	36.56	13.52	77.30	42.51
DivoT5-small (60M)	22.31	77.29$^{\alpha}$	47.38	14.62	88.53$^{\alpha}$	42.37	17.00	**82.68**$^{\alpha}$	43.71
DivoT5-base (220M)	**23.02**	77.65$^{\alpha}$	**47.92**	**16.55**	88.58$^{\alpha}$	**43.09**	**20.90**	82.47$^{\alpha}$	**46.85**

to large unchanged regions, DivoT5 still achieves competitive fluency (88.58 in BLEU), showing no trade-off between correctness and surface similarity.

On the CodeReview-CodeOnly dataset, which is constructed by removing the natural language guidance from the original CodeReview dataset, DivoT5 demonstrates even more pronounced advantages. Specifically, DivoT5-base achieves a 34.3% relative improvement in EM and a 10.9% relative improvement in SARI compared to CodeReviewer. In comparison to CodeT5+, DivoT5-base also shows clear gains, with EM increasing from 12.75 to 20.90 and SARI improving from 39.42 to 46.85. These results highlight the effectiveness of DivoT5 in code-only editing scenarios, particularly in the absence of external natural language signals.

Due to the lack of sufficient domain knowledge and guidance from natural language on the direction of changes, billion-scale few-shot models such as ChatGPT, Meta-Llama-3.1, and DeepSeekCoder perform poorly across all datasets. Although they often achieve reasonable BLEU scores by generating syntactically similar code, they fail to produce exact outputs or apply semantically faithful edits. The knowledge of edit directions and patterns required by these datasets often needs to be acquired through fine-tuning the training set.

The effectiveness of DivoT5 stems from its ability to bridge the significant information gap when evolving code. For example, when editing from code A ('CaDaEa') to code B ('CbDbEb')), instead of directly generating B from A, we introduce intermediate states, like 'CbDaEa' and 'CbDbEa', enabling smoother transitions from A to B, enriching the potential evolution paths to alleviate the difficulty of direct generation.

> **Answer to RQ1:** We conduct experiments on seven datasets for two code editing scenarios. The experimental results demonstrate that the DivoT5-base model significantly outperforms existing models. For example, on the CodeReview dataset, the DivoT5-base model achieves an EM metric improvement of around 10 percentage points compared to the CodeT5-base model. This highlights the superior performance of DivoT5 in capturing code editing characteristics.

B. RQ2: Generalizability to Non-Editing Scenario

To assess the generalization capability of DivoT5 beyond code editing, we introduce the code translation task—a distinct scenario that requires the model to translate code from one programming language to another. In this task, we evaluate the performance of translating Java code to CSharp code and vice versa. Table V demonstrates that even in the code translation task outside the context of editing existing code, DivoT5-base still exhibits improvements over the existing baseline models. For example, in the task of translating CSharp code to Java code, DivoT5-base outperforms the CCT5 model by 6.9 percentage points in EM metric, surpasses the CodeReviewer model by 3.2 percentage points, and exceeds the CodeT5-base model by 4.5 percentage points. Similarly, in the task of translating Java code to CSharp code, DivoT5-base outperforms CodeT5+ (220M) by over 5% in the EM metric. By analyzing the generated results, we find that the translation in this dataset is not simply a direct translation of the original code but also requires corresponding modifications, such as adding the 'override' keyword at specific positions, importing and calling packages for certain functions, or even splitting a single line of code. Besides, regarding our translation task, the high similarity between Java and CSharp makes the transformation resemble code evolution, where the model decides between copying tokens or generating new ones without drastically altering the structure. These modification patterns are embedded in the training set and are difficult to capture through a few-shot approach, leading to poor performance from billion-scale models. However, when considering translation tasks in specific scenarios, DivoT5 demonstrates an advantage in capturing these modification and translation patterns, performing better than other models.

Therefore, we believe that these results validate the generalizability of DivoT5 in code-related tasks. They also highlight how DivoT5 effectively captures both the characteristics of code evolution and the fundamental properties of code itself.

> **Answer to RQ2:** To demonstrate the generalization capability of DivoT5, we conduct experiments on the code translation task. The results show that DivoT5 significantly

TABLE V
RESULTS OF THE CODE TRANSLATION TASK.

	Code Translation					
Models	**Java to CSharp**			**CSharp to Java**		
Metrics	**EM**	**BLEU**	**C-BLEU**	**EM**	**BLEU**	**C-BLEU**
CodeT5-small (60M)	65.40	88.23	88.32	69.60	87.22	87.18
CodeT5-base (220M)	66.90	88.55	88.72	68.70	87.03	86.71
CodeT5-large (770M)	67.20	88.89	88.98	68.80	87.20	87.16
CodeT5+ (220M)	66.20	91.66	91.51	70.20	89.64	91.01
CoditT5 (220M)	68.30	91.45	91.75	70.30	90.28	90.26
CCT5 (220M)	66.90	91.69	91.47	66.30	89.40	89.21
CodeReviewer (220M)	67.80	92.06$^{\alpha}$	91.74	70.00	90.22	90.22
CodeLlama (6.7B-base)	68.40	91.53	91.78	68.10	90.11	87.18
DeepSeekCoder-FS (6.7B-instruct)	7.40	60.17	57.16	10.30	59.51	67.68
Meta-Llama-3.1-FS (8B-instruct)	6.30	59.84	56.98	14.60	57.11	67.78
ChatGPT-FS (gpt-3.5-turbo)	12.80	62.53	59.30	18.90	62.42	72.30
DivoT5-small (60M)	68.30	92.03	91.90	72.50	91.36$^{\alpha}$	91.38
DivoT5-base (220M)	**69.60**	**92.34**$^{\alpha}$	**92.13**	**73.20**	**91.38**$^{\alpha}$	**91.56**

TABLE VI
ABLATION STUDY WITH DIVOT5-SMALL.

Models	**CodeReview**	**NL-CodeRefinement**	**BugFixing-S**	**BugFixing-M**	**Refine-S**	**Refine-S**	**CodeReview-CodeOnly**	**Java-CSharp**	**CSharp-Java**
	EM	**EM**	**EM**	**EM**	**EM**	**EM**	**EM**	**EM**	**EM**
DivoT5-small∼(60M)	**40.16**	**29.42**	**35.84**	**28.72**	**22.31**	**14.62**	**17.00**	**68.30**	**72.50**
w/o KSM_{ED}	35.45	28.69	35.28	27.36	21.75	13.67	14.44	68.00	71.20
w/o RM_{ED}	37.02	28.61	35.30	28.10	21.54	13.91	13.68	66.30	69.80
w/o $EDAE_{ED}$	35.22	28.74	35.50	27.65	20.99	13.86	11.76	67.50	68.60
w/o EDR	37.31	28.72	34.76	28.37	21.68	14.03	15.83	67.50	70.20
w/o ALL	34.75	28.50	35.35	27.36	20.84	13.41	12.05	67.60	68.20

outperforms other models in this task. For example, in translating Java code to CSharp code, DivoT5-base improves the EM score by over 3 percentage points compared to the CodeT5+ model with the same scale.

C. RQ3: Effectiveness of Pre-training Tasks

To evaluate the effectiveness of the proposed pre-training tasks, we conduct an extensive ablation experiment to explore the contribution of each pre-training task. Specifically, we remove the pre-training task separately to explore the model's performance. Then, we remove all four pre-training tasks to assess the overall impact of pre-training datasets on the model's performance. In this setting, our pre-training is to predict the new code on top of the old code. Due to computational resource limitations, we conduct this ablation study on only the small version of DivoT5. Due to resource constraints, these ablation studies are conducted on the 60M-parameter variant of DivoT5, which provides an efficient setting for experimentation under limited computational resources. This choice is consistent with prior work such as DeepSeekCoder [42], where Fill-in-the-Middle (FIM) ablation experiments were conducted on the 1.3B model—nearly 30× smaller than their largest 33B variant—demonstrating a practical strategy for resource-efficient experimentation. Furthermore, our approach follows established practices in the CodeT5 model family [12], [43], where 60M-parameter variants are frequently used to examine the effects of different pre-training strategies. Since DivoT5 is also built on the CodeT5 architecture, employing a 60M model for ablation ensures both computational tractability and methodological consistency with previous studies.

Table VI presents the model performance after removing each pre-training task. It can be observed that the performance of the model declines when any task is removed, indicating that each pre-training task has a positive influence on the overall effects of the model. In the NL-guided code editing scenario, removing the KSM_{ED} task (i.e., w/o KSM_{ED}) results in a notable decline in EM on the CodeReview dataset—from 40.16% to 35.45%. Similarly, removing the EDR task reduces EM to 37.31%. Comparable trends are observed in other NL-guided datasets such as NL-CodeRefinement and BugFixing, underscoring the general utility of each task in guiding precise edits. In the code-only editing and non-editing scenarios, similar degradation is observed. For example, on the Refine-S dataset (code-only editing), removing DAE_{ED} lowers the EM from 22.31% to 20.99%. On the CSharp→Java translation task (non-editing), excluding the RM_{ED} task reduces EM from 72.50% to 69.80%. These results further support the versatility and necessity of each pre-training task across diverse application settings. It is worth noting that, compared to removing other pre-training tasks, the removal of KSM_{ED} task has a significant impact on the CodeReview dataset and has the least impact on the code translation task. This is reasonable, as the KSM_{ED} task focuses on fine-grained code editing, while the code translation task is distant from the editing scenario, thus suffering from the least impact. These results indicate that each pre-training task impacts the final performance.

Moreover, by comparing the removal of all pre-training tasks (i.e., w/o ALL), we can observe the impact of data on model performance. Specifically, we train the w/o ALL version of DivoT5 by directly using the old code as input and the new code as output. The performance of the w/o ALL model

generally falls between that of CodeT5-small and CodeT5-base, indicating that leveraging real code editing history can enhance the model's ability to fulfill code-editing-related tasks to some extent. More importantly, the performance of the w/o ALL model is significantly lower than any ablation version of DivoT5. This demonstrates that our designed directional diffusion strategies within it are crucial for improving model performance than the data alone.

> **Answer to RQ3:** We compare the effect of removing a specific pre-training task from the DivoT5-small model on the EM metric. We find that the performance of the model decreases to varying degrees on three representative downstream datasets. This indicates that each pre-training task contributes to the final experimental performance.

VI. Discussion

Cost of the Directional Diffusion-Style Model. DivoT5, inspired by the gradual denoising concept in diffusion, treats old code during the evolution process as a form of noise and considers the new code as the final noise-free state to simulate real-world code editing scenarios. Instead of traditional diffusion, DivoT5 adopts an autoregressive approach, making its training and inference costs the same as encoder-decoder text generation models like Transformer and CodeT5 [12], [34]. To analyze the training and inference costs, let the sequence length be L (i.e., L_{enc} indicates the input length of the encoder and L_{dec} indicates the output length of the decoder), the model's hidden layer size be H, and the number of denoising steps in traditional diffusion is T. Typically, L_{enc} and L_{dec} are on the same order of magnitude, so both can be approximated as L. Thus, the training and inference costs of traditional diffusion are $O(TL^2H)$, where L^2 reflects interactions between L_{enc} and L_{dec} in the cross-attention mechanism. For an encoder-decoder text generation model with the same structure, the training phase has a time complexity of $O(L_{\text{enc}}^2H)$ for the encoder, $O(L_{\text{dec}}^2H)$ for the decoder, and $O(L_{\text{enc}}L_{\text{dec}}H)$ for the cross-attention layer. These result in an complexity of $O(L_{\text{enc}}^2H + L_{\text{enc}}L_{\text{dec}}H + L_{\text{dec}}^2H)$, yielding an overall time complexity of $O(L^2H)$. In the inference phase, due to the gradual accumulation of the output length, the time complexity increases to $O(L^3H)$.

Above all, the time complexity of the encoder-decoder model during the training phase ($O(L^2H)$) is lower than that of traditional diffusion ($O(TL^2H)$). In the inference phase, T is typically much larger than the sequence length L. For example, to generate text within 200 tokens (L), T may need to be set to 2000 [19], [20]. Consequently, $O(L^3H)$ is often significantly smaller than $O(TL^2H)$. In summary, the training and inference costs of DivoT5 are generally lower than those of traditional diffusion, and can therefore also be effectively adopted for scaling to larger models or alternative architectures.

Impact of Data and Pre-training Tasks. To better reflect real-world editing scenarios, we incorporate code data related to actual changes, which may directly enhance model performance. To isolate the impact of this data, we removed all designed pretraining tasks in an ablation study. As shown in Table VI, while the new data helped improve performance (EM score increased from 32.25 to 34.75 on the CodeReview dataset), the substantial boost from our diffusion strategy (EM score of 40.16) highlights that the design of DivoT5 is more crucial than the dataset. Therefore, we believe that DivoT5's performance is primarily driven by leveraging the diffusion style to simulate real editing scenarios, rather than by the data itself.

Generalization and Extensibility. DivoT5 is designed with strong generalization potential across both code granularity levels and model architectures. While it is trained primarily on token-level code edits, this level of granularity serves as the foundation for modeling more complex transformations. In real-world software maintenance, many structural changes, such as line-level updates, block replacements, or module-level refactorings, are composed of sequences of fine-grained edits. By modeling token-level operations, DivoT5 captures the fundamental building blocks of such transformations and has potential extensions toward hierarchical or semantic-aware edit modeling. Recent studies [16], [17] also emphasize the prevalence and practical significance of token-level modifications in real development workflows, further validating our design choice. In addition, DivoT5 aims to simulate real-world developer behavior by leveraging the intermediate states of code evolution as training signals. Instead of relying solely on before-and-after code pairs, we extract supervision from the actual editing trajectory, which more closely reflects how developers iteratively modify code. These evolution directions are architecture-agnostic and can be applied to various model types. Although we use an encoder-decoder architecture in this work, the same training data and objectives can be easily adapted to other architectures (e.g., decoder-only architecture) or larger language models. This design makes our approach flexible, extensible, and compatible with future model developments.

VII. Threats to Validity

Threats to internal validity might come from the implementation of our experiments. To mitigate this threat, we utilize open-source models available on HuggingFace [60] to minimize potential errors. Additionally, we fix the random seed to ensure experiment reproducibility.

Threats to external validity may arise from two factors. The first factor is the selection of the base model. To mitigate this potential threat, we choose a popular and representative model, CodeT5, as the base model. CodeT5 is widely recognized as a classic encoder-decoder framework and has been extensively applied in various tasks [12]. The second factor lies in the downstream datasets used in our experiment. To mitigate this threat, we choose the widely used datasets from CodeXGLEU [21] and other existing code evolution models [7], [8]. To further generalize our results, we need to evaluate DivoT5 on more datasets. Nonetheless, we acknowledge that important code editing tasks, such as code refactoring, which improves structure without changing functionality, are not covered in this work. In future work, we

plan to scale DivoT5 to larger models and extend it to broader scenarios, including refactoring and architectural changes. As DivoT5 is designed for gradual code evolution with both copying and generation, its advantages are less evident in tasks with large input-output gaps, such as natural language to code or vice versa.

Threat to construct validity may arise from two factors. The first is the selection of evaluation metrics. To mitigate this concern, we choose widely used evaluation metrics in the field of code generation to assess the evolved code. For the CodeReview dataset, we also select the same evaluation metrics as other code evolution models [8]. The second is related to data leakage. To mitigate this threat, we exclude data that overlaps with downstream tasks during the pre-training process. Specifically, we perform precise deduplication using a string-matching approach. If a sample from the downstream task's test set appears in any pretraining data sample 'x', we filter 'x' out. Additionally, we conduct extensive ablation experiments to validate the effectiveness of our technical contributions.

VIII. Conclusion

In this paper, we propose DivoT5, a directional diffusion-style pre-trained model designed to simulate the incremental nature of human code editing and effectively capture the characteristics of code evolution. Specifically, we first propose three pre-training tasks to make the model aware of fine-grained code editing information (i.e., the direction of code evolution) and enrich the context of the old code. Then, we design the fourth pre-training task utilizing intermediate states during the editing process to further reinforce the direction of code evolution. Experiments are conducted on two code-editing scenarios and one non-editing scenario using eight datasets, demonstrating that DivoT5 achieves state-of-the-art performance on most tasks compared to fine-tuned similar or larger scale models (60M, 220M, 770M, 6.7B), and few-shot billion-scale instruct models (6.7B, 8B, ChatGPT). In addition, our ablation experiments also confirm the contribution of each pre-training task to the final performance.

IX. Acknowledgment

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2023YFB4503803, the National Natural Science Foundation of China under Grant No. 62232003 and No. 62402482, and the CCF-Huawei Populus Grove Fund (Grant No. CCF-HuaweiSE202407).

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[2] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 73–87.

[3] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

[4] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[5] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[6] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[7] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.

[8] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[9] Q. Liang, Z. Sun, Y. Zhao, Z. Gong, G. Wang, Y. Chen, L. Zhang, G. Liang, and Q. Wang, "Bipartite-grammar aware pretraining for xml-sql code updating," *ACM Trans. Softw. Eng. Methodol.*, Apr. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3731752

[10] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[11] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

[12] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708.

[13] J. Li, G. Li, Z. Li, Z. Jin, X. Hu, K. Zhang, and Z. Fu, "Codeeditor: Learning to edit source code with pre-trained models," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–22, 2023.

[14] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A code-change-oriented pre-trained model," *arXiv preprint arXiv:2305.10785*, 2023.

[15] J. Wei, G. Durrett, and I. Dillig, "Coeditor: Leveraging repo-level diffs for code auto-editing," in *The Twelfth International Conference on Learning Representations*.

[16] M. Kondo, D. M. German, Y. Kamei, N. Ubayashi, and O. Mizuno, "An empirical study of token-based micro commits," *Empirical Software Engineering*, vol. 29, no. 6, p. 148, 2024.

[17] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 341–350.

[18] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in neural information processing systems*, vol. 33, pp. 6840–6851, 2020.

[19] T. Wu, Z. Fan, X. Liu, H.-T. Zheng, Y. Gong, J. Jiao, J. Li, J. Guo, N. Duan, W. Chen *et al.*, "Ar-diffusion: Auto-regressive diffusion model for text generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[20] X. Li, J. Thickstun, I. Gulrajani, P. S. Liang, and T. B. Hashimoto, "Diffusion-lm improves controllable text generation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 4328–4343, 2022.

[21] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[23] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[24] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[25] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[26] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li *et al.*, "Deep learning-based software engineering: progress, challenges, and opportunities," *SCIENCE CHINA Information Sciences*, vol. 68, no. 1, pp. 111 102–, 2025. [Online]. Available: http://www.sciengine.com/publisher/ScienceChinaPress/journal/SCIENCECHINAInformationSciences/68/1/10.1007/s11432-023-4127-5

[27] Q. Liang, Z. Zhang, Z. Sun, Z. Lin, Q. Luo, Y. Xiao, Y. Chen, Y. Zhang, H. Zhang, L. Zhang, B. Chen, and Y. Xiong, "Grammar-based code representation: Is it a worthy pursuit for llms?" 2025. [Online]. Available: https://arxiv.org/abs/2503.05507

[28] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[29] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[30] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[31] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[32] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep unsupervised learning using nonequilibrium thermodynamics," in *International conference on machine learning*. PMLR, 2015, pp. 2256–2265.

[33] Z. Lin, Y. Gong, Y. Shen, T. Wu, Z. Fan, C. Lin, N. Duan, and W. Chen, "Text generation with diffusion language models: A pre-training approach with continuous paragraph denoise," in *International Conference on Machine Learning*. PMLR, 2023, pp. 21 051–21 064.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[35] Q. Liang, Z. Zhang, C. Liu, Z. Sun, W. Zhang, Y. Chen, Z. Zhao, Q. Luo, W. Wang, Y. Jiang *et al.*, "Condor: A code discriminator integrating general semantics with code details," *arXiv preprint arXiv:2412.17429*, 2024.

[36] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "Improving smart contract security with contrastive learning-based vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–11.

[37] Y. Zhao, Y. Chen, Z. Sun, Q. Liang, G. Wang, and D. Hao, "Spotting code mutation for predictive mutation testing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1133–1145.

[38] Q. Liang, Z. Sun, Q. Zhu, W. Zhang, L. Yu, Y. Xiong, and L. Zhang, "Lyra: A benchmark for turducken-style code generation," *arXiv preprint arXiv:2108.12144*, 2021.

[39] G. Wang, Z. Sun, Z. Gong, S. Ye, Y. Chen, Y. Zhao, Q. Liang, and D. Hao, "Do advanced language models eliminate the need for prompt engineering in software engineering?" *arXiv preprint arXiv:2411.02093*, 2024.

[40] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney, "Learning to update natural language comments based on code changes," *arXiv preprint arXiv:2004.12169*, 2020.

[41] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[42] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[43] Q. Zhu, Q. Liang, Z. Sun, Y. Xiong, L. Zhang, and S. Cheng, "Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[44] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," *arXiv preprint arXiv:2308.07124*, 2023.

[45] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *Advances in neural information processing systems*, vol. 31, 2018.

[46] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[47] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 2552–2562.

[48] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318.

[49] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020.

[50] X. Zhou, K. Kim, B. Xu, D. Han, J. He, and D. Lo, "Generation-based code review automation: How far are we?" in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 215–226.

[51] W. Xu, C. Napoles, E. Pavlick, Q. Chen, and C. Callison-Burch, "Optimizing statistical machine translation for text simplification," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 401–415, 2016.

[52] OpenAI, "Chatgpt," 2024. [Online]. Available: https://www.openai.com/chatgpt

[53] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[54] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[55] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[56] DivoT5, "Divot5," 2025. [Online]. Available: https://github.com/LIANGQINGYUAN/DivoT5

[57] Pytorch, "Pytorch," 2023. [Online]. Available: https://pytorch.org/

[58] Huggingface, "Accelerate," 2023. [Online]. Available: https://github.com/huggingface/accelerate

[59] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[60] Huggingface, "Huggingface transformers," 2023. [Online]. Available: https://huggingface.co