The Dissertation Committee for Jiyang Zhang
certifies that this is the approved version of the following dissertation:

Large Language Models for Code Editing

Committee:

Milos Gligoric, Supervisor

Junyi Jessy Li, Co-supervisor

Raymond J. Mooney

August Shi

Haris Vikalo

Large Language Models for Code Editing

by

Jiyang Zhang

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2025

Acknowledgments

I will always remember this important period of my life, from 2019 to 2025, during which I learned more about myself and the world around me, growing from an academic freshman to a researcher. However, I could not have made it through on my own without the incredible support of many wonderful people I met along the way. I would like to take this opportunity to express my sincere gratitude to all those who have been a meaningful part of this six-year journey.

First, I would like to express my gratitude to my supervisor, Milos Gligoric, who guided me throughout my PhD. I cannot imagine working with anyone else for such a long period of time. I still remember choosing to pursue my PhD with him because I knew he genuinely cares about his students and is willing to advise them and teaching great research and professional practices. From him, I learned to be passionate about my work, to maintain high standards in research, and the intense working style needed to meet deadlines. I will never forget the days when we worked on paper submissions until the very last second before the 7 a.m. deadlines. These lessons will be lifelong assets and will continue to benefit me in the years to come.

I would like to thank my co-supervisor, Junyi Jessy Li, who was involved in nearly all of my research projects during my PhD. I am grateful for her guidance to explore new machine learning techniques and research directions through relevant readings and discussions. I took two of her research seminar courses and audited one. Although I initially struggled to fully follow the discussions, these courses greatly helped me learn how to read research papers and formulate meaningful research problems.

I would also like to thank Raymond J. Mooney, August Shi, and Haris Vikalo for serving on my committee and providing constructive feedback on my research. I am especially thankful to Ray for demonstrating both passion and rigor in research at the NLP+Programming seminar which has been inspiring. I am also grateful to

August for organizing the Software Engineering seminar and for playing an important role in one of my first accepted research papers.

I am thankful to all of my collaborators: Apoorva Agrawal, Ram Bairi, Christian Bird, Arie Deursen, Kim Herzig, Yamini Jhawar, Owolabi Legunsen, Xiaopeng Li, Yu Liu, Chandra Maddila, Pengyu Nie, Sheena Panthaplackel, Schanely Phillip, Ujjwal Raizada, Marko Ristin, Hans Venn, Shiqi Wang, Samuel Yuan, Yuhao Zhang and Linghan Zhong. Among them, I am especially grateful to Pengyu Nie and Sheena Panthaplackel. As senior students, they provided invaluable guidance and support during the early stages of my PhD. Pengyu was involved in nearly all of my projects, and I learned from him how to properly organize project repositories, use Git, write in LaTeX, write code, conduct research, and many other foundational skills essential for doing research. I thank Sheena, who actively participated in the discussions of my research projects while she was at UT Austin. I still remember that the first part (Chapter 2) of this dissertation originated from an idea she proposed during one of the NLP+Programming seminars. Some of her suggestions were also key to making the model work. I especially remember the days when she helped me write the paper from scratch in a short period of time—one of the most intense and memorable tasks I have ever completed.

I would like to thank my lovely officemates: Nader Al Awar, Abdelrahman Baz, Cheng Ding, Ivan Grigorik, Changyong Hu, Yang Hu, Jaeseong Lee, Chengpeng Li, Tong-Nong Lin, Yu Liu, Pengyu Nie, Shanto Rahman, Aditya Thimmaiah, Wenxi Wang, Zijian Yi, Zhiqiang Zang, Linghan Zhong and Chenguang Zhu. We shared memories while working in the office in EER and attending conferences.

I would like to give special thanks to Zhiqiang Zang for being a wonderful companion before he graduated and mentioning me in his acknowledgments in his dissertation. We often chatted about fun topics in the office and supported each other throughout our time together. He was my go-to person whenever I encountered problems related to Java, Bash, or LaTeX, as he was an expert in all three. I am also

grateful for the many times he invited me to dinners on weekends, as well as for a washing machine and a TV he kindly gave me after he graduated. I also want to thank Zijian Yi for being the “next” Zhiqiang, bringing laughter into the office. I deeply appreciate his generosity in providing me with temporary housing when my apartment was affected by the thunderstorm in May 2025.

I would like to thank all of my friends here in Austin: Yiyue Chen, Jiaxun Cui, Jierui Li, Dawei Liang, Jiaxin Lin, Shaohui Liu, Ke Ma, Zhongjie Ren, Xinran Song, Yipeng Wang, Chengyang Wu and Yuqi Zhou. And people who helped me during my internships at Microsoft Research, Salesforce Research and Amazon Web Services. In particular, I am deeply grateful to Yingchen Wang, one of the most important people in my PhD journey. I am thankful for her unconditional support and for bringing me so many unforgettable memories over the past years. Thanks to her, I was able to overcome challenges and become a better person than I was before we met.

I want to thank my badminton buddies here in Austin: Ke Chen, Si Chen, Coco, Yuefei Huang, Zunlong Ke, Karen, Kiran, Shi Li, Chang Liu, Chenxu Liu, Ziqi Liu, Hengfa Lu, Jinhua Lü, Ze Ouyang, Chengjia Shao, Linlin Shen, Kathy Shuai, Bijun Tang, Yingchen Wang, Yixian Wang, Ziyue Wang, Chenxu Yan, Chenxi Yang, Xiao You, Weiwen Zeng, Hongming Zhang, Ruizhe Zhang, Ruixun Zhang and Weihan Zhang. I cannot imagine what my PhD life would have been without you. The improvement in my badminton skills and the countless happy moments would not have been possible without your presence.

I would like to give a heartfelt shoutout to everyone in my badminton family in Austin: Zunlong Ke, Hengfa Lu, Ze Ouyang, and Bijun Tang. Although we only became close during the last half of the year, they truly made Austin feel like a second home for me in the United States. I shared many memorable experiences with them before leaving Austin. I am grateful for the times we played badminton together at Bellmont Hall and the Austin Badminton Academy, joined Sunday training sessions, did karaoke together and attended multiple gatherings at their apartments and house.

I appreciate them teaching me how to play Mahjong, organizing the trip to the water park, taking me to the activation room, and encouraging me to wake up early and work out at Gregory Gym. I thank them for helping me prepare for and attend my PhD defense, and for giving me an unforgettable graduation celebration. Most importantly, I want to thank them for showing me what true friendship means and how to genuinely care for others.

I would like to thank the support staff at the Department of Electrical and Computer Engineering—Thomas Atchity, Cayetana Garcia, Melanie Gulick, Barry Levitch, and Melody Singleton—for their invaluable assistance.

Parts of this dissertation were published at ASE 2022 [122] (Chapter 2); and FSE 2023 [125] (Chapter 3). I would like to thank the anonymous reviewers and audience at the conferences of all my papers for their comments.

My research was funded by Cisco Research and the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, CCF-2217696, CCF-2313027 and CCF-2403036.

Last but not least, I would like to thank my mother Xi, my father Jianxin, and my sister Jiqing for their unconditional love and support. They have always cheered for me, celebrating even the smallest accomplishments throughout my PhD journey.

Abstract

Large Language Models for Code Editing

Jiyang Zhang, PhD
The University of Texas at Austin, 2025

SUPERVISORS: Milos Gligoric, Junyi Jessy Li

Software systems continuously evolve as new features are added, bugs are fixed, or new platforms are supported. Software maintenance is a critical part of the software lifecycle, ensuring that the software systems remain up-to-date, functional, and secure after the initial development. However, software maintenance is often time-consuming and tedious, making it a great target for automation.

Recently, Large Language Models (LLMs) have shown strong performance in many software-related *generation* tasks, such as code and documentation generation, significantly improving developers' productivity. Nevertheless, these generative models are not well-suited for the *editing* nature of the software maintenance tasks, as they explicitly learn to generate by being trained on the generative autoregressive objectives such as next-token prediction. The key insight of this dissertation is that we have to design and develop *evolution-aware* LLMs to better assist software developers in their day-to-day software maintenance work.

This dissertation presents the design and implementation of two evolution-aware LLMs to assist software developers in maintaining software systems, including tasks such as comment updating, bug fixing, automated code review, and software co-evolution across programming languages.

First, this dissertation introduces CoditT5, an encoder-decoder transformer model for software maintenance tasks. We design a novel edit-based output format which explicitly models edits and use it as the pretraining objective to train CoditT5 on a large amount of source code and natural language comments. We fine-tuned CoditT5 on three downstream software maintenance tasks, including comment updating, bug fixing, and automated code review. By outperforming standard generation models that are trained solely on generative objectives, our approach demonstrates strong generalizability and effectiveness for editing tasks. We also show how a standard generation model and our edit-based model can complement one another through simple reranking strategies, with which we achieve state-of-the-art performance for the three downstream tasks, outperforming both models before reranking by up to 19%.

Second, this dissertation introduces CodeEditor, which assists software developers in co-evolving software projects that have the same set of requirements implemented in two (or more) programming languages. For example, the widely used parser generator ANTLR has implementations in Java, Python, and C#. To address this problem, we formulate a novel task: automatically updating the code in a target programming language, based on the changes made in the source programming language. CodeEditor is implemented to tackle this task by explicitly modeling code changes as edit sequences and learning to correlate these changes across programming languages. To evaluate CodeEditor, we collect a corpus of 6,613 aligned code changes from eight pairs of popular open-source software projects that implement similar functionalities in two programming languages (Java and C#). Experimental results show that CodeEditor outperforms the generation models that fine-tuned on this task by more than 25%.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Software plays a critical role in driving innovation, improving operational efficiency by reducing manual effort, and supporting essential services in modern society such as healthcare, transportation, education, and finance. With the emergence of Large Language Models (LLMs), researchers have found that software engineering—especially code generation—is a well-established application scenario for LLMs due to the naturalness of programming languages [40, 52, 86] and the availability of clearly defined evaluation metrics—test case pass rate [16, 17, 113]. A large number of LLMs trained on code [17, 41, 62, 91, 110, 111] and LLM-based software engineering agents [31, 109, 115] have been developed to assist humans in various aspects of software development, ranging from code generation to software testing and verification [27, 59, 70, 72, 121, 123, 126, 127]. The LLMs acquire knowledge of software engineering and programming languages through pretraining on large code corpora using the *generative objectives*, such as next-token prediction [1, 83]. As a result, these generative models demonstrate impressive performance on software-related generation tasks like code completion [17, 61, 72, 97], code generation [2, 61, 73, 91, 110], and code translation [20, 76, 119, 120, 130].

However, software development is not solely about writing new code. Software is constantly evolving as new features are added and security vulnerabilities are detected and patched on a near-daily basis. Software maintenance is one of the essential responsibilities of software developers to ensure the reliability, efficiency, and safety of software systems. Common tasks include fixing security issues, refactoring the codebase to improve its performance or usability, updating libraries, SDKs, and frameworks to supported versions, keeping documentation up to date, reviewing code changes and addressing review comments. These tasks involve modifying existing code rather than creating entirely new code or text from scratch.

Figure 1.1 shows an example of the comment updating task, which entails

```
/** @return double The yaw Euler angle. */
public double getRotY() {
    - return mOrientation.getRotationY();
    + return Math.toDegrees(mOrientation.getRotationY());
}
```

Figure 1.1: An example of the comment updating task.

automatically updating an existing comment when the corresponding body of code is modified [56, 60, 77]. The return statement of the Java method `getRotY` was modified so that the return value of `mOrientation.getRotationY()` is passed to `Math.toDegrees()` to be converted to degrees before being returned. To reflect this code change and maintain consistency between the comment and the code, the `@return` comment for this method should be updated to '@return double The yaw Euler angle in degrees'.

The standard generation-based models are *not* well-suited for the *editing* nature of software maintenance tasks because their training objectives are designed for generation. Formally, given a sequence of tokens $x = (x_1, x_2, \dots, x_T)$, the goal of a language model is to estimate the joint probability of the sequence as a product of conditional probabilities: $P(x) = \prod_{t=1}^{T} P(x_t \mid x_1, x_2, \dots, x_{t-1})$. During training, the model is optimized to maximize the likelihood of observed sequences. This objective encourages the model to assign high probability to the correct next token at each time step, given the preceding context. They lack the ability to reason about what should be changed and what should be preserved when applied to software maintenance tasks. For example, bug fixing involves modifying code to resolve issues that cause the software to behave incorrectly, automated code review requires making changes based on a reviewer's feedback, and comment updating entails revising natural language comments to align with code modifications. Prior work [67, 122] shows that existing LLMs perform poorly on tasks that require code editing; they often copy the input without making any meaningful modifications. Namely, these models are not *aware of software evolution and code editing.*

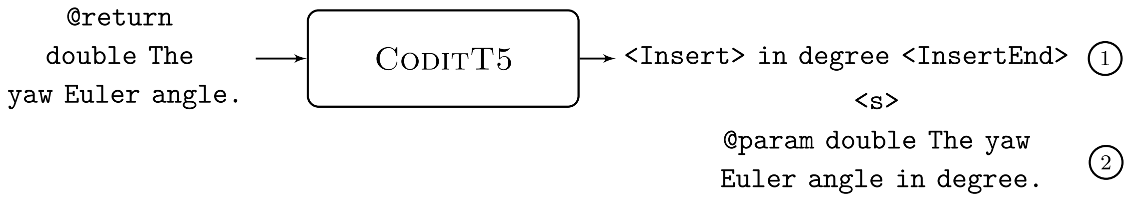


Figure 1.2: Example output of CoditT5.

The thesis statement of this dissertation is that *we can design and develop evolution-aware LLMs that can greatly help with software maintenance tasks.* We present two key insights in developing evolution-aware LLMs: 1) designing LLMs that explicitly reason about and perform code edits by being trained to predict a specialized edit format under a novel edit-based pretraining objective, and 2) utilizing the code evolution data for both training the models on code history data and providing the code evolution data as the model's context. We built on these insights and proposed the first pretrained LLM for general software-related editing tasks. The pretrained edit-based model is further fine-tuned on task-specific software evolution data, allowing it to capture common software editing patterns in different software maintenance tasks.

As a first step, we propose CoditT5, the first LLM designed for software maintenance tasks. As shown in Figure 1.2, CoditT5 is trained to generate a structured output format, where an *edit plan* (① in the Figure) specifying explicit edit *operations* is generated first, followed by the *target edited sequence* (②). We define three edit operations—`insert`, `delete`, and `replace`—to represent the edit plan. The target edited sequence is the result of applying the edit plan to the original input. In Figure 1.2, the input is an outdated comment. The output includes the edit plan, which inserts 'in degree' to the comment, and the target edited sequence, which reflects the updated `@return` comment. By pretraining CoditT5 to produce this structured output format on the code corpus, we encourage the model to reason about the required edits and how they should be applied to generate the final target sequence. After pretraining, we fine-tune CoditT5 on various software maintenance tasks, including comment updating, bug fixing, and automated code review. In our evaluation, we show that CoditT5 outperforms the standard generation-based models of similar size, which highlights that the models that are trained to explicitly perform edits are better suited for editing tasks in the software domain than generation models.

To realize our second insight, we apply CoditT5 to a software maintenance task and explore the potential of using code evolution data as context for the model. Many software projects implement APIs and algorithms in multiple programming languages. Co-evolving such projects is tiresome, as developers have to ensure that any change (e.g., a bug fix or a new feature) is being propagated, timely and without errors, to implementations in other programming languages. We introduce a novel task to address this problem: automatically *updating* code snippets in a target programming language based on the *changes* made in the source programming language. Building upon CoditT5, we present Codeditor, a model that learns to align edits across programming languages and explicitly applies these edits to the old version of the code in the target language to tackle this task. By training the model to translate and apply code changes, Codeditor achieves better performance than existing generation-based code translation models in an extensive evaluation. This highlights that models trained to explicitly perform edits and utilize code evolution data can better assist software developers in software maintenance.

This dissertation makes the following key contributions:

⋆ We introduce CoditT5, a large language model for software maintenance tasks that is pretrained on large amounts of source code and natural language. We propose a novel pretraining objective that entails first generating a plan consisting of edit operations to be applied to the input sequence followed by the resulting target sequence. Upon pretraining and task-specific fine-tuning, we show that CoditT5 achieves improved performance over existing models for three distinct downstream editing tasks (comment updating, bug fixing, and automated code review), demonstrating its effectiveness and generalizability.

- ⋆ We propose to address the multilingual software co-evolution problem using LLMs by leveraging software change histories as input to the models. We formulate the novel task of automatically updating code written in one programming language based on the changes in the corresponding code in another programming language. We design and implement CODEDITOR, the first LLM for this task which learns to align the edits across programming languages and explicitly performs edits on the old version of the code in target programming language. To fine-tune and evaluate the model, we create the first dataset with aligned code changes for two programming languages (Java and C#) from 8 open-source project pairs.

- ⋆ We conduct an extensive evaluation of CODITT5 and CODEDITOR on multiple tasks, including comment updating, bug fixing, automated code review, and multilingual code co-evolution. We compare the performance of CODITT5 and CODEDITOR with existing generation-based models and rule-based methods. The results show that the evolution-aware models outperform the existing models on all tasks, demonstrating the effectiveness of our approach.

Both CODITT5 and CODEDITOR are open source, and they are publicly available at `https://github.com/EngineeringSoftware/CoditT5` and `https://github.com/EngineeringSoftware/codeditor`, respectively.

Chapter 2: CoditT5: Pretraining for Source Code and Natural Language Editing

Pretrained language models have been shown to be effective in many software-related generation tasks; however, they are not well-suited for editing tasks during maintaining the software as they are not designed to reason about edits. To address this, we propose a novel pretraining objective which explicitly models edits and use it to build CoditT5, a Large Language Model (LLM) for software-related editing tasks that is pretrained on large amounts of source code and natural language comments. We fine-tune it on various downstream software maintenance tasks, including comment updating, bug fixing, and automated code review. By outperforming standard generation-based models, we demonstrate the generalizability of our approach and its suitability for editing tasks. We also show how a standard generation model and our edit-based model can complement one another through simple reranking strategies, with which we achieve state-of-the-art performance for the three downstream tasks[1].

2.1 Introduction

Large Language Models (LLMs) pretrained on massive amounts of data have led to remarkable progress in recent years, with models like GPT [12, 83], BART [53], and T5 [85] yielding huge improvements for a vast number of text generation tasks. Inspired by this, a new research initiative has emerged around building LLMs that are pretrained on source code and technical text to address software-related tasks. This includes models like CodeGPT-2 [61], PLBART [2], and CodeT5 [110]. While these models demonstrate impressive performance on generation tasks like code summarization, code generation, and code translation, it is unclear if they are well-suited for the *editing*

[1]Parts of this chapter are published at ASE 2022 [122]. I led the design, implementation, and evaluation of the model, and paper writing.

Before Editing

```
default List<Pattern> getExcludedResponseHeaderPatterns() {
    return emptyList();
}
```

Reviewer's Comment

Generally better to qualify than making static import

PLBART

```
default List<Pattern> getExcludedResponseHeaderPatterns() {
    return emptyList();
}
```

Figure 2.1: An example in automated code review task where PLBART merely copies the input which does not match reviewer's comment.

nature of many software maintenance tasks. For instance, bug fixing [48, 66, 102] entails editing source code to resolve bugs, automated code review [87, 103, 105, 124] requires editing source code to incorporate feedback from review comments, and comment updating [30, 56, 60, 77] pertains to updating outdated natural language comments to reflect code changes.

In principle, such software maintenance tasks can be framed as standard generation tasks in which an input sequence (e.g., *buggy* code snippet) is completely re-written to form the output sequence (e.g., *fixed* code snippet). In this way, existing pretrained conditional generation models can be fine-tuned to autoregressively generate a sequence from scratch. However, this can be problematic in practice [67, 77]. When applying large generation models like PLBART and CodeT5 to these tasks, we find that they can generate output which merely copies the input without performing any edits (up to 34.25%) or even deviates substantially from the input, introducing irrelevant changes. We provide an example of automated code review in Figure 2.1, where a reviewer prescribes edits that need to be made to a given code snippet: "Generally better to qualify than making static import". Conditioned on the code snippet and this comment, PLBART generates an output sequence which copies the original code, without applying any edits. While the output is valid and a likely sequence according to PLBART's language model, it makes no edits based on the reviewer's comments.

We attribute these weaknesses to the fact that such models rely on pretraining objectives designed for generating code (or software-related natural language) in sequence by exploiting patterns with respect to preceding tokens. Therefore, a model has to learn to *implicitly* perform edits by generating tokens one by one in accordance with the underlying probability that it has learned for which tokens belong alongside one another, rather than being aware of where information should be retained or modified.

Intuitively, edit-based generation requires a different approach that more frequently refers back to the input sequence, and can often be characterized by localized operations (e.g., insertion, deletion, substitution). To guide a model in discerning edit locations in the input sequence and reason about the necessary edit operations, we design a novel pretraining objective that *explicitly* models edits. Our approach is inspired by content planning in natural language generation where a skeleton of key elements are first generated and used to guide more accurate and precise generation of full text [25, 64, 82, 88]. Additionally, training LLMs to first generate intermediate reasoning steps before producing the final answer has been shown to effectively elicit the models' reasoning ability and further improve their performance [36, 112]. Specifically, during decoding, a model first generates an *edit plan* that explicitly details the edit operations. Then, it proceeds to autoregressively generate the target edited sequence, during which it attends to the edit plan. Through this, we effectively encourage the model to learn to better reason about edits and how they should be applied to form the target sequence. Using this objective, we develop CoditT5, an LLM for software-related edit tasks that is pretrained on more than 5.9 million open-source programming language code snippets and 1.6 million natural language comments from the CodeSearchNet [42] training data.

For evaluation, we fine-tune CoditT5 on three downstream software maintenance tasks: comment updating, bug fixing, and automated code review. For each of these tasks, we show that CoditT5 outperforms state-of-the-art models, as well as large pretrained standard generation-based models. Through this, we demonstrate that our model and the proposed edit-based pretraining objective generalize across tasks and are better suited for editing tasks in the software domain.

Furthermore, in our evaluation, we find that our edit-based model, CoditT5, can be further improved if combined with a standard generation-based model. We find that the edit-based and standard generation-based models are complementary to one another. Namely, while the edit-based model provides better explicit modeling of concrete edits, a standard generation-based model provides certain advantages in terms of the contextual coherence of the generated target sequence. To exploit this complementary nature of these models, we combine the two models through reranking strategies which require no additional training. Our results show that the combined approaches outperform the two models individually by up to 19%.

We summarize our main contributions as follows:

- We formulate a novel pretraining objective that entails first generating a plan consisting of edit operations to be applied to the input sequence followed by the resulting target sequence.
- We build and release CoditT5, a large language model for software-related editing tasks that is pretrained on large amounts of source code and natural language with the new pretraining objective.
- Upon task-specific fine-tuning, we show that CoditT5 achieves improved performance over existing models for three distinct downstream software maintenance tasks (comment updating, bug fixing and automated code review), demonstrating its effectiveness and generalizability.
- We show that by combining our edit-based CoditT5 model with a standard generation model through simple reranking strategies, we can beat each of the individual models and achieve new state-of-the-art in all three tasks, demonstrating

the complementary nature of edit-based and standard generation models.

Our code and data is publicly available at `https://github.com/EngineeringSoftware/CoditT5`.

2.2 Background

We first give a high-level overview of the building blocks that are necessary to understand our approach.

2.2.1 Generation with Transformer-Based Models

Conditional Sequence Generation. Conditional sequence generation entails generating an output sequence given an input sequence. Many tasks are framed in this manner, including machine translation (e.g., translating a sentence from French to English) [6], text summarization (e.g., generating a brief summary for a given news article) [92], and code generation (e.g., generating a code snippet for a given natural language specification) [117].

Encoder-Decoder Framework. In recent years, conditional sequence generation tasks are being addressed with encoder-decoder models. An encoder-decoder model consists of two neural components: an encoder and a decoder. The input sequence is fed into the encoder, which produces learned vector representations of the tokens in that sequence. These learned vector representations are then passed into the decoder, which generates the output sequence one token at a time. Specifically, the decoder predicts the next token by reasoning over the input sequence and the tokens generated at previous time steps.

Transformers. Transformers [106] are powerful neural models that are commonly adopted as the encoder and decoder in the encoder-decoder framework. These models rely on an *attention* mechanism to learn representations for tokens by relating them to other tokens in the sequence. Namely, a transformer-based encoder will learn representations for each token in the input sequence by "attending" to other input tokens. For the decoder, when generating a token at timestep t, it will "attend" to the representations of the output tokens generated from timestep 1 to $t-1$ as well as the representations of tokens from the input sequence. Transformer models can become very large with huge numbers of attention heads, encoder and decoder layers.

2.2.2 Large Pretrained Language Models

Large pretrained language models generally refer to the class of large transformer-based models that are trained on large amounts of unlabeled data (collected from webpages, news articles, etc.) with unsupervised training objectives. This includes a vast number of models like GPT [12, 83], BART [53], and T5 [85].

Denoising Autoencoder Pretraining. BART and T5 models are pretrained using denoising autoencoding unsupervised training objectives. Namely, a noising function is first applied to a given input sequence *inp* to form *inp*′. Common noising functions include *Token Masking*: tokens in the input sequence are randomly masked; *Token Deletion*: random tokens are deleted from the input sequence; *Token Infilling*: a span of tokens are sampled and replaced with a mask token; *Sentence Permutation*: sentences in the document are shuffled in a random order. Then, *inp*′ is fed into a model's encoder, and the encoder's learned representation is passed into the decoder, which generates an output sequence, *out*, that is expected to resemble the original input sequence (*inp*). In other words, the model is trained to "denoise" *inp*′, using a training objective that minimizes the error between *out* and the original input, *inp*. Through this, the model learns to extract meaning from the input sequence and also generate fluent and coherent output. Therefore, by pretraining on massive amounts of data, the model develops an understanding of how things in the world relate to one another as a strong language modeling capability.

Fine-tuning for Downstream Tasks. Since large pretrained language models are trained using unsupervised training objectives on huge amounts of data, they cannot generally be directly applied to downstream tasks (e.g., translation, summarization). Fine-tuning is a common technique to transfer the knowledge learned during pretraining to target downstream tasks. Specifically, the pretrained model is further trained for the downstream task on some amount of supervised data.

2.2.3 Large Pretrained Language Models for Software Engineering

Inspired by the success of large pretrained models in Natural Language Processing (NLP), a number of machine learning models pretrained on source code and technical text have been proposed for solving various software-related problems.

For instance, inspired by BART, Ahmad et al. [2] developed PLBART, which is a large pretrained language model that can be fine-tuned for a number of code understanding (e.g., code summarization) and generation (e.g., code translation) tasks. Similarly, inspired by T5, Wang et al. [110] built a larger model CodeT5, which is pretrained on six programming languages together with their natural language comments collected from open-source repositories. Specially, it is pretrained to incorporate information from identifiers in the code. CodeT5 has shown promising results in code-related generation tasks such as code summarization, code generation and code-related understanding tasks such as clone detection and vulnerability identification. However, aforementioned models are for generation and they are only implicitly aware of edit operations if at all.

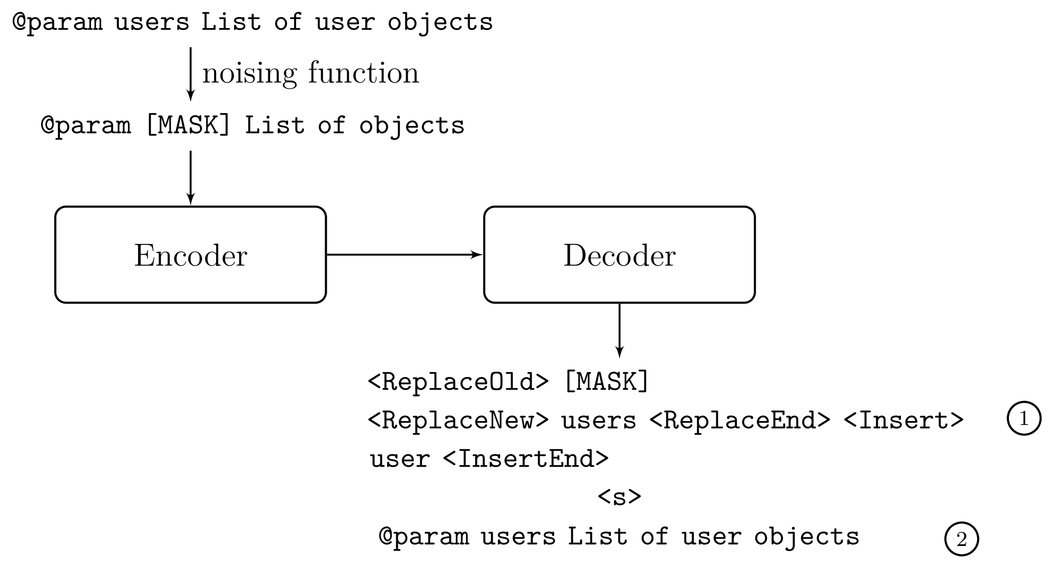

@param users List of user objects
noising function
@param [MASK] List of objects
Encoder
Decoder
<ReplaceOld> [MASK] <ReplaceNew> users <ReplaceEnd> <Insert> user <InsertEnd>
1
<s>
@param users List of user objects
2

Figure 2.2: Overview of CoditT5. The corrupted text is encoded with a bidirectional encoder, and the decoder is pretrained to generate sequences of edit actions to recover the original text followed by a separation token (`<s>`), and finally the target sequence.

2.3 CoditT5

CoditT5 is built upon the encoder-decoder framework with the same architecture as CodeT5. As shown in Figure 2.2, the model is pretrained with our proposed objective: generating the edit-based output sequence given the corrupted input sequence. In this section, we first explain our proposed pretraining objective (Section 2.3.1). We then discuss how we build CoditT5 by pretraining on this objective, including the data used for pretraining (Section 2.3.2), and additional details of the pretraining setup (Section 2.3.3).

2.3.1 Pretraining Objective

We formulate a new pretraining objective that is designed to encourage a model to explicitly reason about edits. At a high-level, this objective falls under the realm of denoising autoencoding in which an input sequence is first corrupted with noising functions and the model is trained to *denoise* the corrupted sequence by generating

Table 2.1: Percentage that model just copy the input.

Dataset	**PLBART**	**CodeT5**	**CoditT5**
$B2F_s$	6.48	7.97	0.55
$B2F_m$	10.92	10.08	0.78
Comment Updating (clean)	21.33	16.67	2.67
Comment Updating (full)	34.25	25.47	5.73
Automated Code Review	22.24	29.28	1.28

an output sequence that matches the original input sequence. While existing models like PLBART and CodeT5 pretrained using this setup perform very well on various generation tasks (e.g., code summarization/generation), we find that they do not generalize well when fine-tuned on editing tasks. Namely, they are susceptible to learning to copy the original input sequence instead of actually performing edits, up to 34.25% of the time as shown in Table 2.1.

We propose the following *edit-based output sequence* representation (shown in Figure 2.2): [Edit Plan] `<s>` [Target Sequence], where the model is trained to generate an *edit plan* (①) consisting of explicit edit operations that must be applied to the corrupted sequence to reconstruct the original input sequence, followed by a separation token (`<s>`), and finally the *target sequence* (②) that matches the original input sequence. This is inspired by the concept of *content planning*, originating from natural language generation [88]. In content planning, a high-level plan is first outlined, specifying the discourse structure of the content to be generated, and then lexical realization is performed to generate the text. Additionally, inspired by the recent work on LLM reasoning [36, 112], we aim to train the model to generate an edit plan, encouraging it to reason about how to perform the edits before producing the final edited output.

2.3.1.1 Edit Plan

The edit plan entails the specific edit operations that are needed to recover the original input sequence. For example, in Figure 2.2, the input sequence: "`@param` users List of user objects" is corrupted by masking "users" and removing the token "user" before the word "objects": "`@param [MASK]` List of objects". With this, a model must first reason about the fact that `[MASK]` in the corrupted input sequence needs to be replaced with "users" and "user" should be inserted between "of" and "objects" to reconstruct the given input. To construct the sequence of edit operations, we closely follow the format proposed in [77]:

```
<Operation> [span of tokens] <OperationEnd>
```

Here, `<Operation>` is either `Insert` or `Delete`. We also include the `Replace` operation, with a slightly different structure (since both the old content to be replaced as well as the new content to replace it with must be specified):

```
<ReplaceOld> [span of old tokens]
<ReplaceNew> [span of new tokens] <ReplaceEnd>
```

To determine the specific edit operations for a given example, we use difflib[2] to compute the optimal set of edits needed to transform the corrupted input sequence into the original input sequence. Multiple edit operations are placed in the same order as the span of tokens under editing appears in the input sequence (for example, the edit plan in Figure 2.2 consists of two edit operations).

2.3.1.2 Target Sequence

One might ask whether we could simply apply the sequence of edit operations in the generated edit plan to the corrupted input sequence directly to recover the

[2]`https://docs.python.org/3/library/difflib.html`

original input sequence *heuristically.* For example, if we align "`<ReplaceOld> [MASK] <ReplaceNew> users <ReplaceOld>`" with a corrupted input sequence "`@param [MASK]` List of user objects", it is very clear that all we need to do is replace `[MASK]` with "users" and no additional generation is needed. However, there are two main issues with this. First, not all operations will be specified in a deterministic manner. For example, if the edit plan is "`<Insert> user <InsertEnd>`", it is not clear where the new token "user" should be added to. Second, the generated edit plan does not correspond to contiguous output tokens since it consists of fragmented information (edit operations and token spans) rather than a complete sentence. As a result, neural language models may fail to generate correct edit plans due to their lack of language properties such as fluency and coherency [77].

Therefore, we need an additional step for *learning* to apply edits while simultaneously maintaining fluency and coherency. For this reason, once the edit plan is outlined as a sequence of edit operations, the target sequence (which is expected to recover the original input sequence) must also be generated: "`@param` users List of user objects". The decoder generates tokens in a left-to-right manner, meaning that when generating a token at a given timestep, it is aware of all tokens generated in previous timesteps. So, when generating the target sequence, the decoder can exploit the sequence of edits that was generated in the edit plan earlier. In this way, the model can reason about the edits and the generation simultaneously.

2.3.1.3 Noising Functions

To support learning across a diverse set of edit actions during pretraining, we consider multiple noising functions for corrupting the input sequence: 1) randomly masking spans with the special `[MASK]` token which requires the model to replace it with the correct spans, 2) inserting `[MASK]` token at random positions which requires the model to identify the useless spans, and delete them and 3) deleting spans of tokens in the input sequence which requires the model to pinpoint the position and add back the missing pieces.

Table 2.2: Statistics collected from downstream tasks for creating pretraining dataset. Avg. Number of Tokens represents the average number of tokens in each edited span; Avg. Number of Spans represents the average number of edited spans in each input sequence.

	PL	**NL**
Probability of `Delete` edit	0.49	0.07
Probability of `Insert` edit	0.21	0.11
Probability of `Replace` edit	0.30	0.82
Avg. Number of Tokens	6.50	3.00
Avg. Number of Spans	1.90	1.40

2.3.2 Pretraining Data

We describe data collection, data preparation, and pretraining setup in this section.

2.3.2.1 Data Collection

Following prior work, we pretrain CoditT5 on large amounts of source code and natural language comments from the CodeSearchNet [42] dataset which consists of functions of six programming languages (Java, Python, Ruby, PHP, Go and JavaScript) together with the natural language comments. CodeSearchNet is widely used to pretrain LLMs, such as CodeT5 [110] and UniXcoder [35]. We use the training set of the processed CodeSearchNet dataset provided by Guo et al. [35] which contains 6.1 million code snippets (functions/methods) and 1.9 million natural language comments.

2.3.2.2 Data Preparation

To enable CoditT5 to capture common edit patterns, we want the pretraining dataset to reflect the common activities conducted by software developers. Specifically, in the pretraining dataset, the probability of each edit operations applied to the spans in the input sequence and the length (number of tokens) of the corrupted span should be consistent with the distributions and sizes of real-world edits in downstream editing tasks.

To this end, we collect statistics for source code edits from the training sets of the bug fixing and automated code review downstream tasks and statistics for natural language edits from the comment updating's training set. As shown in Table 2.2, we collect the probability of each edit operation (insert, delete and replace) to be performed on a span; the average number of tokens in each span that is edited; and the average number of spans that are edited in each input sequence. For each example in the pretraining dataset, we then uniformly sample the spans and the edit operations that should be applied in accordance with the statistics collected from the downstream datasets.

Similar to CodeT5 [110], we use the RoBERTa [58] tokenizer to tokenize all sequences (input, edit plan, target). More concretely, the tokenizer splits words in the sequence into *tokens* (subwords) that are used by the model. Moreover, we remove input sequences that are shorter than 3 tokens and longer than 512 tokens after tokenization which leave us with 5.9 million programming language code snippets and 1.6 million natural language comments. This is because too short inputs are usually incomplete and CodeT5 is designed to only handle sequence of length 512. Table 2.3 presents the statistics of the pretraining dataset.

2.3.3 Pretraining Setup

Model Architecture. CoditT5 consists of 12 encoder and decoder layers, 12 attention heads, and a hidden dimension size of 768. The total number of parameters is 223M. Model parameters are initialized from the CodeT5-base model, and we further pretrain it on the CodeSearchNet pretraining dataset (Section 2.3.2) using our proposed objective (Section 2.3.1).

Training. We implement CoditT5 using PyTorch 1.9.0 and use 16 NVidia 1080-TI

Table 2.3: Statistics of the datasets used to pretrain CoditT5. First row: number of programming language and natural language; second row: average number of tokens in corrupted input sequences; third row: average number of tokens in the output sequence (edit plan + target sequence).

	PL	**NL**
# Examples	5,956,069	1,675,277
Avg. Corrupted Input Tokens	102.01	15.42
Avg. Target Output Tokens	120.23	26.57

GPUs, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz for pretraining for 4 days. For fine-tuning, we run the experiments on 4 NVidia 1080-TI GPUs, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with the same hyper-parameters as CodeT5.

2.4 Experimental Design

To assess CoditT5 and our proposed pretraining objective, we fine-tune the model on three software maintenance downstream tasks. Note that during fine-tuning, the model is still trained to generate the edit-based output sequence. However, at test time, we discard the edit plan and take the generated target sequence as the final model output. Namely, we use the generated sequence after the separation token `<s>` as model's prediction.

2.4.1 Downstream Tasks

Comment Updating. The task of comment updating entails automatically updating a natural language comment to reflect changes in the corresponding body of code [30, 56, 60, 77]. For instance, in Example 2 in Figure 2.5, the old `@return` comment needs to be revised based on the changes in the method. Instead of directly returning the yaw Euler angle measured in radians, the unit of the return value is changed to degrees in the new version, with the method call `Math.toDegrees()`.

Table 2.4: Statistics for the datasets used for downstream tasks.

Task		**Train**	**Valid**	**Test**
Comment Updating	clean	16,494	1,878	150
	full	16,494	1,878	1,971
Bug Fixing	B2F$_s$	46,628	5,828	5,831
	B2F$_m$	52,324	6,542	6,538
Automated Code Review		13,753	1,719	1,718

Bug Fixing. Given a *buggy* code snippet, the task of bug fixing entails generating a *fixed* code snippet, which no longer contains the bug [101].

Automated Code Review. Given a code snippet under review and a brief natural language sentence prescribing code edits, automated code review requires automatically generating the revised code snippet, which captures the recommended changes [103]. For example, in Figure 2.1, `emptyList()` should be changed to `Collections.emptyList()` because the reviewer suggests *not* using static import.

2.4.2 Data for Downstream Tasks

We use datasets that have been established and previously used for each of the three tasks. The statistics of the datasets is shown in Table 2.4. Unlike pretraining where the goal is to recover the corrupted input sequences, during fine-tuning, CoditT5 is trained to generate an edit plan for completing the downstream editing task, that can be applied to a part of the input (e.g., old comment), followed by the target sequence (e.g., new comment).

Comment Updating. For this task, Panthaplackel et al. [79] has released a corpus of Java method changes paired with changes in the corresponding comments (spanning `@return`, `@param`, and summary comments). This dataset also comes with a *clean* subset of the test set which was manually curated. The input sequence used for fine-tuning is formed by concatenating the old comment and code edits. The code edits follow the representation described in Section 2.3.1.1, except that an additional `Keep` operation is included to denote spans that are left unchanged.

Bug Fixing. We consider the Java BugFixPairs-Small (B2F$_s$) and BugFixPairs-Medium (B2F$_m$) datasets, originally released by Tufano et al. [101]. Chakraborty and Ray [14] supplemented these datasets with additional context, namely natural language guidance from the developer, and the method where the patch should be applied. B2F$_s$ contains shorter methods with a maximum token length 50, and B2F$_m$ contains longer methods with up to 100 tokens in length. The input sequence used for fine-tuning is formed with the buggy code, natural language guidance, and code context.

Automated Code Review. We use the automated code review dataset released by Tufano et al. [103], which consists of Java methods (before and after the review) paired with pull request comments, derived from pull request reviews on GitHub and Gerrit. To reduce the vocabulary size, they further abstracted Java methods by replacing identifiers and literals with special tokens. In this work, we use the data with concrete tokens. The input sequence used for fine-tuning is formed using the code snippet before review and the pull request comment from reviewers.

2.4.3 Baselines

2.4.3.1 Generation Baselines

We consider two large standard generation language models of similar size trained with denoising autoencoding pretraining objectives which are not edit-based: **PLBART** and **CodeT5**. Both of these are fine-tuned to directly generate the target output sequence. Furthermore, to better assess the value of actually pretraining using the proposed objective instead of simply fine-tuning a model to generate an edit-based output sequence, we also consider fine-tuning CodeT5 to generate the specialized edit-based output sequence representation. We refer to this as **CodeT5**

(w/ edit-based output). We fine-tune each of these models using the same input context as CoditT5.

2.4.3.2 Task-Specific Baselines

We additionally compare against the state-of-the-art models for each of the downstream tasks.

For comment updating, the state-of-the-art model is Panthaplackel et al. [77], which entails Recurrent Neural Network (RNN) based encoders for representing the old comment and code edits, and an RNN-based decoder for decoding edits. These edits are parsed at test time and reranked based on similarity to the old comment and likelihood based on a comment generation model.

For bug fixing, the state-of-the-art model is essentially PLBART fine-tuned on the $\mathrm{B2F_s}$ and $\mathrm{B2F_m}$ to generate the fixed code [14].

For automated code review, no baselines are available for the specific version of the dataset we used with concrete identifiers and literals (rather than the one with abstracted identifiers and literals). Therefore, we rely on those described in Section 2.4.3.1 and establish new baselines for this version of the dataset.

2.4.4 Evaluation Metrics

For comment updating, we report performance on the same metrics that have been used previously to benchmark models for this task [77]. This includes: xMatch (whether the model prediction *exactly matches* the ground truth), common metrics that measure lexical overlap for evaluating text generation (BLEU-4 [3] [80] and METEOR [7]), and common metrics for measuring text editing (GLEU [68] and SARI [114]). For bug fixing, we use xMatch, as done in prior work [14]. For automated code review, we report performance on xMatch and BLEU-4, which have been used

[3]We measure 1∼4-gram overlap and compute the average.

Table 2.5: Results for comment updating on the clean test set. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	**BLEU-4**	**METEOR**	**GLEU**	**SARI**
Panthaplackel et al. [77]	33.33	56.55	52.26	51.88	56.23
PLBART	35.33	62.04^{γ}	56.79	54.75	52.83
CodeT5	38.00	65.20^{α}	59.63	58.84^{β}	58.80
CodeT5 (w/ edit-based output)	40.00	62.97^{γ}	59.08	58.72^{β}	$61.11^{\epsilon\eta}$
CoditT5	43.33^{χ}	64.56	60.75	59.53	$61.41^{\delta\epsilon}$
CoditT5 (reranked with CodeT5)	**45.33**	**66.80**	**63.33**	**61.60**	$61.48^{\delta\eta}$
CodeT5 (reranked with CoditT5)	44.00^{χ}	65.58^{α}	62.44	60.48	**62.57**

previously to benchmark models for this task [103].

2.5 Evaluation

We organize our evaluation around three main research questions:

RQ1: How does our edit-based model, CoditT5, compare to generation and task-specific baselines for edit-related software maintenance tasks?

RQ2: Does our proposed pretraining objective help a model in better reasoning about and performing edits?

RQ3: Can a standard generation model complement CoditT5 by integrating the two models?

2.5.1 Comparing CoditT5 to Baselines

We present results in Tables 2.5-2.8. Note that the results shown in the last two rows in each of the tables are explained later in Section 2.5.3. We perform statistical significance testing using bootstrap tests [9] with confidence level 95%.

Table 2.6: Results for comment updating on the full test set. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	**BLEU-4**	**METEOR**	**GLEU**	**SARI**
Panthaplackel et al. [77]	24.81	48.89	44.58	45.69	47.93
PLBART	22.98	$55.42^{\hbar\iota}$	49.12	47.83	43.40
CodeT5	28.56	58.37^{α}	53.13	51.90	49.23
CodeT5 (w/ edit-based output)	$29.83^{\delta\gamma}$	54.83	50.71	50.67^{ϵ}	52.01^{η}
CoditT5	29.38^{δ}	$55.30^{\beta\iota}$	51.14^{χ}	50.62^{ϵ}	51.39
CoditT5 (reranked with CodeT5)	$\mathbf{30.14}^{\gamma}$	$\mathbf{58.72}^{\alpha}$	**53.60**	**52.81**	50.47
CodeT5 (reranked with CoditT5)	27.80	$55.54^{\beta\hbar}$	51.44^{χ}	50.02	$\mathbf{52.24}^{\eta}$

***RQ1**: How does our edit-based model, CoditT5, compare to generation and task-specific baselines for edit-related tasks?*

We find that CoditT5 (and most of the pretrained models) drastically outperforms Panthaplackel et al. [77] (a non-pretrained model) across metrics for comment updating. This demonstrates the value of LLM pretrained on vast amounts of data using unsupervised pretraining objectives.

Next, across all three tasks, CoditT5 achieves higher performance than the two standard generation-based pretrained models, significantly outperforming PLBART and CodeT5 for most of the metrics, highlighting the benefit of explicitly modeling edits for these software maintenance tasks that require editing. In fact, CodeT5 (w/ edit-based output), which explicitly models edits only during fine-tuning rather than pretraining, outperforms CodeT5 on edit-based metrics (xMatch, SARI). This further underlines the utility of the edit-based output sequence representation that we developed.

Nonetheless, across most metrics, CoditT5 still outperforms CodeT5 (w/ edit-based output), which is not pretrained using the pretraining objective but uses the same edit-based output sequence representation during fine-tuning. This demonstrates the importance of actually pretraining with this representation rather than relying on

Table 2.7: Results on bug fixing dataset. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	
	B2F$_s$	**B2F$_m$**
PLBART	31.09	24.18
CodeT5	34.81	26.66
CodeT5 (w/ edit-based output)	36.37	29.28$^\alpha$
CoditT5	37.52	29.96$^\alpha$
CoditT5 (reranked with CodeT5)	**40.22**	**32.06**$^\beta$
CodeT5 (reranked with CoditT5)	39.56	32.24$^\beta$

Table 2.8: Results for automated code review. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	**BLEU-4**
PLBART	26.78	79.38
CodeT5	34.98	83.20
CodeT5 (w/ edit-based output)	36.38$^\alpha$	80.06$^\beta$
CoditT5	37.19$^\alpha$	80.50$^\beta$
CoditT5 (reranked with CodeT5)	40.98	**84.12**$^\chi$
CodeT5 (reranked with CoditT5)	**43.42**	83.92$^\chi$

fine-tuning alone.

2.5.2 Evaluating our Pretraining Objective

While we observe that CoditT5 tends to achieve slightly lower performance than CodeT5 on generation-based metrics (BLEU-4, METEOR) for two of the tasks, we find that it significantly outperforms other metrics which capture whether the correct edits are generated, such as xMatch and GLEU and SARI for comment updating. This suggests that CoditT5 is indeed better at *editing*. By inspecting the outputs of

Before Editing:

```
public HashConfigurationBuilder capacityFactor (float capacityFactor) {
    if ( numSegments < 0 )
        throw new IllegalArgumentException ("capacityFactor must be positive");
    this.capacityFactor = capacityFactor ;
    return this;
}
```

Reviewer's Comment:

typo: capacityFactor instead of numSegments

CodeT5:

```
public HashConfigurationBuilder capacityFactor(float capacityFactor) {
    this.capacityFactor = capacityFactor;
    return this;
}
```

CoditT5:

```
public HashConfigurationBuilder capacityFactor (float capacityFactor) {
    if ( capacityFactor < 0 )
        throw new IllegalArgumentException ("capacityFactor must be positive") ;
    this.capacityFactor = capacityFactor;
    return this;
}
```

Figure 2.3: Comparing the output of CodeT5 and CoditT5 for a automated code review example. CodeT5 generates incorrect output that drastically deviates from the input code while CoditT5 generates the correct output, performing only relevant edits.

the two models, we find that CodeT5 tends to make drastic and unnecessary edits while CoditT5 appears to be better at making more fine-grained edits. For example, in Figure 2.3, CodeT5 generates output that completely discards critical statements in the code, whereas CoditT5 is able to correctly localize the part of the input code that needs to be changed and make edits properly. We attribute this to the fact that CodeT5 is not designed to reason about edits while CoditT5 is. We further evaluate the influence of our proposed pretraining objective on this editing capability.

RQ2: *Does our proposed pretraining objective help a model in better reasoning about and performing edits?*

Table 2.9: Percentages of target sequence generated by CoditT5 being consistent with the edit plan.

Datasets	**Is Consistent (%)**
$B2F_s$	92%
$B2F_m$	88%
Comment Updating (clean)	87%
Comment Updating (full)	85%
Automated Code Review	74%

First, we compare how often CoditT5 naively copies the input content without actually performing any edits, to two pretrained models which use generation-based pretraining objectives. We report the percentages in Table 2.1. By copying substantially less often than the PLBART and CodeT5, we find that CoditT5 learns to more frequently perform edits with our proposed edit-based pretraining objective which indicates it is suitable for editing tasks.

CoditT5's decoder is encouraged to generate a target sequence that follows the outlined edit plan; however, we do not constrain the decoder in any way to do this[4]. Nonetheless, we find that in the majority of cases (74%-92%), the target sequence is consistent with the edit plan, as shown in Table 2.9. More concretely, the target sequence generally resembles what would be produced if the edit operations in the edit plan were applied to the original content. This suggests that the pretraining objective does in fact guide the model in reasoning about edits.

For cases in which there is ambiguity or errors in the edit plan, we find that CoditT5 still often manages to generate the correct target sequence, by disregarding unreasonable edits or disambiguating ambiguous edits. We show two examples in automated code review in Figure 2.4 with the Java method before review, the generated edit plan, and the generated target sequence. In Example 1, the edit plan is ambiguous

[4]We do not want potential errors in the edit plan to propagate to the target sequence.

Example 1

Before Editing:

```
protected boolean isProcessed(ChronicleLogOffsetTracker tracker, long offset) {
    long last = tracker.readLastCommittedOffset();
    return ( last > 0 ) && ( last >= offset );
}
```

Reviewer's Comment:

No need for parentheses.

Edit plan

```
⟨Delete⟩ ( ⟨Delete_End⟩ ⟨Delete⟩ ) ⟨Delete_End⟩
```

Target sequence:

```
protected boolean isProcessed(ChronicleLogOffsetTracker tracker, long offset) {
    long last = tracker.readLastCommittedOffset();
    return last > 0 && last >= offset;
}
```

Example 2

Before Editing:

```
public Builder setDataSize(Estimate dataSize) {
    this.dataSize = requireNonNull(dataSize, "dataSize can not be null");
    return this;
}
```

Reviewer's Comment

you don't validate in other builders method (and you don't have to)

Edit plan

```
⟨Delete⟩ requireNonNull(dataSize, "dataSize can not be null"); ⟨Delete_End⟩
```

Target sequence:

```
public Builder setDataSize(Estimate dataSize) {
    this.dataSize = dataSize;
    return this;
}
```

Figure 2.4: Examples for automated code review for which CoditT5 generated ambiguous or erroneous edit plans but still managed to generate the correct target sequences.

since there are multiple instances of "(" and it does not specify which one(s) should be deleted. However, the generated target sequence is correct, as the model was able to correctly reason about the most appropriate edit locations. In Example 2, the edit plan is imprecise and blindly following this plan would result in syntactically incorrect code, but the model still managed to perform the correct edits and produced valid output by ignoring the fallacious edit. Overall, we find that both components of the edit-based output sequence representation used in the pretraining objective—edit plan and target sequence—are critical.

2.5.3 Integrating CoditT5 and CodeT5

CoditT5 is designed to complement a generation model by providing more explicit guidance for edits. However, a model that is trained to generate edits can struggle with coherence and fluency since it is not actually trained to generate consecutive text [77]. By also including the generation of the target sequence in the pretraining objective, we do mitigate this to some extent, even when there are ambiguities or errors in the edit plan. However, there appears to be a trade-off between performing the correct edits while maintaining performance with respect to generation metrics. More specifically, in Tables 2.5-2.8, CoditT5 outperforms CodeT5 with respect to xMatch (and SARI for comment updating), but underperforms with respect to BLEU-4. To exploit the slight superiority of CodeT5 in this respect, we consider incorporating CodeT5 into our approach.

RQ3: *Can a pure generation model complement CoditT5 by integrating the two models?*

2.5.3.1 Experiment Setup

We combine the two models using simple likelihood-based reranking strategies at test time (with no additional training). Namely, at test time, CoditT5 and CodeT5 each generate 20 candidates using beam search. While we have been only looking at the top one prediction for all previous experiments, we will consider all 20 candidates for reranking. We compute a reranking score for each of these to essentially re-score them. The candidate which has the highest reranking score will be the final model prediction. We investigate two different reranking strategies:

CoditT5 (reranked with CodeT5):. To exploit the language-specific norms learned by CodeT5, we rerank the candidates generated by CoditT5 based on the probability score CodeT5's language model assigns to the corresponding target sequences (namely after `<s>`).

Example 1

Buggy Code

```java
public List<TagVFilter> getFilters() {
    if ((filters) == null ) {
        filters = new ArrayList<TagVFilter>();
    }
    return filters;
}
```

CoditT5:

```java
public List<TagVFilter> getFilters() {
    if ((filters) == null ) {
        filters = new ArrayList<TagVFilter>();
    }
    return new ArrayList(filters);
}
```

CoditT5 (reranked with CodeT5):

```java
public List<TagVFilter> getFilters() {
    if ((filters) == null ) {
        filters = new ArrayList<TagVFilter>();
    }
    return new ArrayList<TagVFilter>(filters);
}
```

Example 2

```java
/** @return double The yaw Euler angle. */
public double getRotY() {
    return mOrientation.getRotationY();
}

/** @return ? */
public double getRotY() {
    return Math.toDegrees(mOrientation.getRotationY());
}
```

CodeT5: @return double The yaw Euler angle.

Reranked CodeT5: @return double The yaw Euler angle in degrees.

Figure 2.5: Examples from comment updating and bug fixing which demonstrate the impact of reranking.

We compute the length-normalized conditional log probability score of CodeT5 generating the target sequence, conditioned on the same input:

$$score = log(P(T|I)^{\frac{1}{N}})$$

where T is the target sequence, I is the model's input, N is the length of T. We also length-normalize the log probability of the candidate, as scored by CoditT5, and then add the two probability scores together to obtain the reranking score.

CodeT5 (reranked with CoditT5):. Conversely, we also rerank the output of CodeT5 based on the likelihood of CoditT5, such that the generated sequence can be assessed in terms of explicit edits. We first parse the output of CodeT5 into the edit-based output sequence representation (as described in Section 2.3.1.1) and then concatenate it with the model's output using `<s>`. Then we compute the likelihood of CoditT5 generating this sequence, conditioned on the same input. We then add the length-normalized log probability score of CoditT5 with the score originally assigned by CodeT5 (after length-normalizing and applying log).

2.5.3.2 Results

We provide results in the bottom two rows of Tables 2.5-2.8. By reranking the output of CoditT5 using CodeT5, we are able to achieve improved performance on all the metrics including BLEU-4 across tasks (and the other generation-based metric, METEOR, for comment updating). To illustrate this, consider Example 1 in Figure 2.5, with a buggy code snippet and outputs corresponding to CoditT5 before and after reranking. We observe that CoditT5 correctly localizes the bug and correctly identifies that the edit entails initializing an `ArrayList` in the return statement. However, the generated target sequence is a defective code snippet which does not properly initialize an `ArrayList` with the correct type `TagVFilter`. By leveraging CodeT5's likelihood score, we are able to effectively filter out the defective prediction and obtain the correct output.

By reranking the output of CodeT5 using CoditT5, we see significant improvements with respect to CodeT5 on metrics that more directly evaluate whether the correct edits were performed, including xMatch as well as GLEU and SARI for comment updating. This suggests that the edit-based and generation-based models are indeed complementary to one another. As a case study, consider Example 2 in Figure 2.5. CodeT5 produces a sequence which simply copies the old comment, without capturing the code changes. While this may be a likely comment sequence, according to CodeT5's language model, copying without applying any edits is not a likely edit plan to be generated for CoditT5.

By combining CoditT5 and CodeT5 through reranking, we can further boost performance substantially across most metrics for all three tasks, outperforming the two models individually, and achieving new state-of-the-art.

2.6 Limitations

Other Programming Languages. The downstream editing tasks we studied in this work use Java. Since CoditT5's pretraining is on the dataset consisting of six programming languages, we expect it to also perform well on editing tasks in other programming languages, but we leave empirically verifying this as future work.

Data Contamination. CoditT5 is pretrained on data collected from open-source projects. It is possible that similar examples in pretraining data exist in downstream tasks' test set. While prior work [12] has shown that data contamination may have little impact on the performance of pretrained models in natural language processing tasks, future work can investigate this problem for pretrained models for software engineering.

2.7 Conclusion

We present a novel edit-driven pretraining objective and use it to develop CoditT5, a pretrained language model for software-related editing tasks. CoditT5 is pretrained on large amounts of source code and natural language comments to perform edits, and we evaluate this model by fine-tuning it on three distinct downstream software maintenance tasks: comment updating, bug fixing, and automated code review. By outperforming task-specific baselines and pure generation baselines across tasks, we demonstrate the suitability of CoditT5 (and our pretraining objective) for editing tasks and its generalizability. We additionally find that a pure generation-based model and CoditT5 can complement one another through simple reranking strategies, which outperform each of the models individually and also achieve new state-of-the-art performance for the three software maintenance tasks that we consider.

Chapter 3: Multilingual Code Co-evolution using Large Language Models

Many software projects implement APIs and algorithms in multiple programming languages. Maintaining such projects is tiresome, as developers have to ensure that any change (e.g., a bug fix or a new feature) is being propagated, timely and without errors, to implementations in other programming languages. In the world of ever-changing software, using rule-based translation tools (i.e., transpilers) or machine learning models for translating code from one language to another provides limited value. Translating each time the entire codebase from one language to another is not the way developers work. In this chapter, we target a novel task: translating code changes from one programming language to another using Large Language Models (LLMs). We design and implement the first LLM, dubbed CODEDITOR, to tackle this task. CODEDITOR explicitly models code changes as edit sequences and learns to correlate changes across programming languages. To evaluate CODEDITOR, we collect a corpus of 6,613 aligned code changes from 8 pairs of open-source software projects implementing similar functionalities in two programming languages (Java and C#). Results show that CODEDITOR outperforms the state-of-the-art approaches by a large margin on all commonly used automatic metrics. Our work also reveals that CODEDITOR is complementary to the existing generation-based models, and their combination ensures even greater performance [1]

3.1 Introduction

To ensure flexibility and a wide adoption of their software, companies provide application programming interfaces (APIs) for their services in several programming

[1]Parts of this chapter are published at FSE 2023 [125]. I led the design, implementation, and evaluation of the model, as well as writing the paper.

languages. Services, such as Google Cloud [32] and MongoDB [43], offer APIs written in most popular programming languages, including C++, C#, Java, and Python. Furthermore, popular software packages, like ANTLR [81] and Lucene [93], have options to target different programming languages for the purpose of being used across various platforms easily.

Maintaining software that offers the same functionality in multiple programming languages is challenging. Any code change, due to a feature request or a bug fix, has to be propagated timely to all programming languages. At present, developers have to manually *co-evolve* code. This requires developers to manually find the correspondence between code snippets and apply necessary *edits*.

There has been work that could, in theory, help with translation. Rule-based migration tools [4, 29, 100] have been designed to translate between high-level programming languages (e.g., Java and C#). However, rule-based systems require developers who have expertise with both programming languages to manually write rules to specify the translation mappings. And the rules need to be updated with the evolution of programming languages themselves; they quickly become outdated [4, 13]. Recent work on automatic code translation [51, 61, 90, 99, 119, 120, 129] aim to directly translate between a source and a target programming language with the help of LLMs, which are pretrained on multiple programming languages. While these techniques could be used to produce code snippets that look correct, they make irrelevant changes that deviate substantially from the newly introduced features in the source programming language, or they fail to precisely infer the project-specific data types and class names.

Figure 3.1 illustrates the limitation of existing models. Developers changed `PdfException` to `LayoutExceptionMessageConstant` in method `docWithInvalid-Mapping02` in the Java project `itext/itext7` [45]. In a later commit in the corresponding C# project `itext/itext7-dotnet` [44], developers revised method `Doc-WithInvalidMapping02` with exactly the same edits while keeping other parts of the

```
@Test
public void docWithInvalidMapping02() throws IOException {
  ...
  customRolePara.getAccessibilityProperties().setRole(HtmlRoles.p);
  Exception e = Assert.assertThrows(PdfException.class,()->document.add(
      customRolePara));
  - Assert.assertEquals(MessageFormat.format(PdfException.
      ROLE_IS_NOT_MAPPED_TO_ANY_STANDARD_ROLE, "p"), e.getMessage());
  + Assert.assertEquals(MessageFormat.format(LayoutExceptionMessageConstant.
      ROLE_IS_NOT_MAPPED_TO_ANY_STANDARD_ROLE, "p"), e.getMessage());
}
```

Java Change Made by Developers

```
[NUnit.Framework.Test]
public virtual void DocWithInvalidMapping02() {
  ...
  - customRolePara.GetAccessibilityProperties().SetRole(LayoutTaggingPdf2Test.
      HtmlRoles.p);
  + customRolePara.GetAccessibilityProperties().SetRole(HtmlRoles.p);
  - Exception e = NUnit.Framework.Assert.Catch(typeof(PdfException),()=>
      document.Add(customRolePara));
  + Exception e = NUnit.Framework.Assert.IsThrows(PdfException.class,()=>
      document.Add(customRolePara));
  - NUnit.Framework.Assert.AreEqual(String.Format(PdfException.
      ROLE_IS_NOT_MAPPED_TO_ANY_STANDARD_ROLE, "p"), e.Message);
  + NUnit.Framework.Assert.AreEqual(String.Format(
      LayoutExceptionMessageConstant.ROLE_IS_NOT_MAPPED_TO_ANY_STANDARD_ROLE, "
      p"), e.Message);
}
```

C# Change Predicted by Existing Generation-based Model ✗

```
[NUnit.Framework.Test]
public virtual void DocWithInvalidMapping02() {
  ...
  customRolePara.GetAccessibilityProperties().SetRole(
    LayoutTaggingPdf2Test.HtmlRoles.p);
  Exception e = NUnit.Framework.Assert.Catch(typeof(PdfException),()=>document.
      Add(customRolePara));
  - NUnit.Framework.Assert.AreEqual(String.Format(PdfException.
      ROLE_IS_NOT_MAPPED_TO_ANY_STANDARD_ROLE, "p"), e.Message);
  + NUnit.Framework.Assert.AreEqual(String.Format(
      LayoutExceptionMessageConstant.ROLE_IS_NOT_MAPPED_TO_ANY_STANDARD_ROLE, "
      p"), e.Message);
}
```

C# Change Made by Developers and Predicted by Our Codeditor ✓

Figure 3.1: Example of using LLMs to help developers co-evolve code in two programming languages. The top box shows developer-made changes in a Java project `itext/itext7`, which needs to be propagated to the corresponding C# project `itext/itext7-dotnet`. The middle box shows the prediction by an existing generation-based large language model, which incorrectly changes irrelevant parts of the code. The bottom box shows the correct prediction by our model, CODEDITOR.

method unchanged. We provide the Java code change, the prediction of an existing large language model, CodeT5 [110], fine-tuned for code translation, and the correct C# code change in Figure 3.1. The added lines of code are highlighted in green and the removed ones are highlighted in red. Although the existing model is able to correctly translate the updated exception type from Java to C#, it misses the class name for the field `HtmlRoles` and incorrectly infers the function call `Assert.Catch` as it does *not* use the prior version of C# code for reference.

To build more robust and accurate techniques that help software developers co-evolve projects implemented in different languages, we propose to explicitly model the *changes* that need to be made. We formulate a novel task: automatically *updating* code snippets in a target programming language, based on the *changes* made in the source programming language.

Most of the existing models implicitly tackle the code evolution tasks by generating tokens one by one in accordance with the underlying learned probability instead of focusing on how the code should be *modified* or retained. Prior work [14, 15, 22, 67, 77, 101, 116, 122] have shown that standard generation-based models underperform models that explicitly model the edits on software-editing tasks.

To support code evolution across programming languages, we design an LLM, dubbed CODEDITOR, which 1) learns to align the edits across programming languages and 2) has access to the code evolution history to perform edits on the *old version* of the code in a target programming language. Additionally, it is fine-tuned on the real-world multilingual code evolution data. Following prior work [22, 77, 94, 122], we enable the model to reason about necessary edits and learn to apply them by directly generating an edit sequence.

For training and evaluation, we collect the first dataset with aligned Java and C# code changes on the methods with similar functionality and implementations. Specifically, we extract 6,613 pairs of code changes from 8 open-source Java projects and the corresponding C# projects on GitHub by mining the commit histories. This is the first dataset containing parallel code changes of two programming languages. We conduct the evaluation in two directions, updating C# method based on the Java changes (source language is Java and target language is C#) and updating Java method based on the C# changes (source language is C# and target language is Java).

Our results show that CODEDITOR outperforms all existing models across all the chosen automatic metrics, including the large pretrained generative models: Codex [17] under few-shot setting and gpt-3.5-turbo [74] under zero-shot setting. CODEDITOR achieves 96 (out of 100) CodeBLEU score on the task of updating C# code based on Java changes, which is more than 25% higher than the large pretrained generation-based model fine-tuned on this task.

Further, we find that CODEDITOR and generation-based models are complementary to each other as CODEDITOR is better at updating longer code snippets while generation model is better at handling the shorter ones. Thus, we combine the two models by choosing either model's prediction based on the size of the input code. Our results show that the combination can further improve our CODEDITOR model's exact-match accuracy by 6%.

The main contributions of this chapter include:

- **Task**. To assist developers in multilingual code evolution, we formulate a novel task of automatically updating code written in one programming language based on the changes in the corresponding code in another programming language.
- **Model**. We design and implement CODEDITOR, the first LLM for this task which learns to align the edits across programming languages and explicitly performs edits on the old version of the code in target programming language.
- **Dataset**. We create the first dataset with aligned code changes for two programming languages (Java and C#) from 8 open-source project pairs.
- **Results**. We show that CODEDITOR significantly outperforms the existing LLMs

fine-tuned for code translation on exact-match accuracy by 77%. We also show that CODEDITOR is complementary to generation-based LLMs and the combination can further improve CODEDITOR's exact-match accuracy by 6%.

CODEDITOR and our corpus are publicly available on GitHub: `https://github.com/EngineeringSoftware/codeditor`.

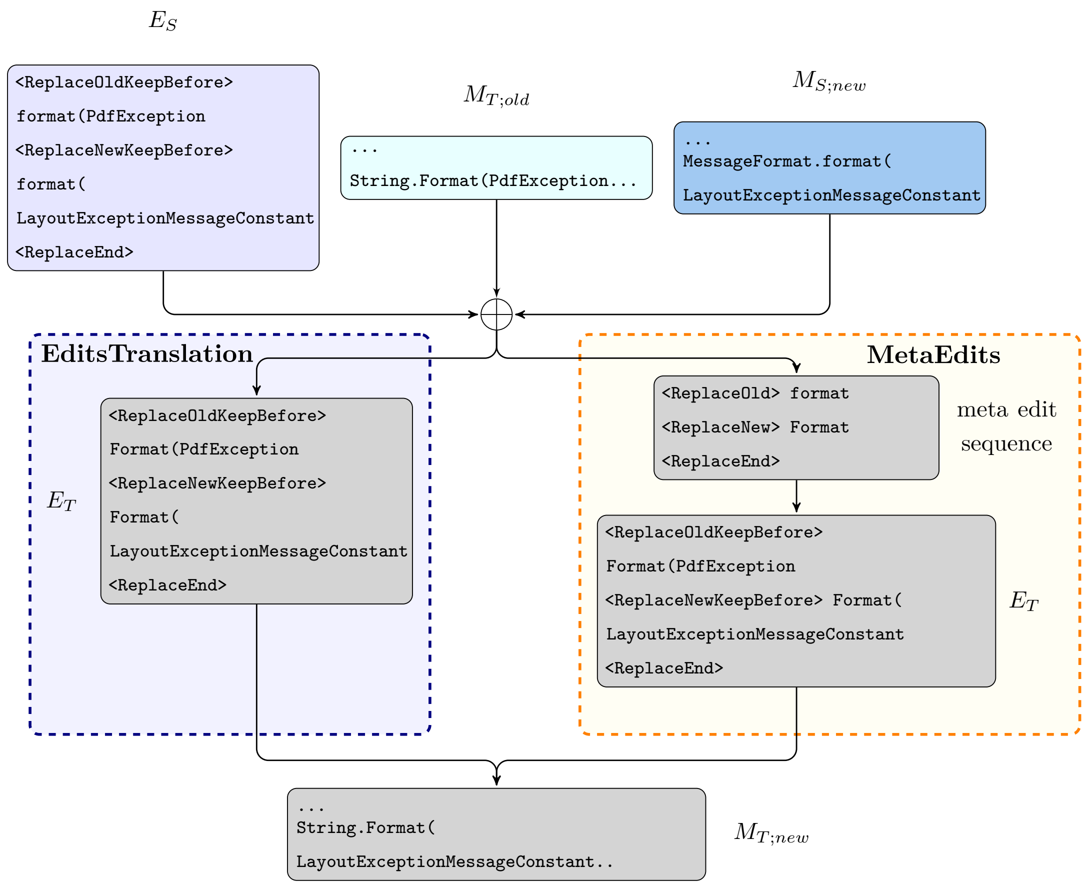
3.2 Task

At a high level, we work on a system that is triggered when a software developer, who maintains projects written in multiple programming languages, makes changes to one method in one of the languages, i.e., the "source" language. The system would automatically suggest updates to the methods with identical functionality in other language(s), i.e., the "target" language(s). To scope our work in this chapter, we focus on Java as the source language, and C# as the target language. We leave evaluation that targets other programming languages as future work.

In Figure 3.1, consider a method $M_{S;old}$ (`docWithInvalidMapping02`) written in the source language S and a method $M_{T;old}$ (`DocWithInvalidMapping02`) written in the target language T with identical functionality (hence similar implementation). Given the updated method $M_{S;new}$ in S, we define the task to generate the new method $M_{T;new}$ in T leveraging context provided by the code changes E_S, such that its functionality is consistent with $M_{S;new}$. Namely, we model the conditional probability distribution

$$P(M_{T;new}|M_{T;old},\ M_{S;new}, E_S)$$

and generate $M_{T;new}$ by sampling from the distribution.



```
<ReplaceOldKeepBefore>
format(PdfException
<ReplaceNewKeepBefore>
format(
LayoutExceptionMessageConstant
<ReplaceEnd>
```

```
...
String.Format(PdfException...
```

```
...
MessageFormat.format(
LayoutExceptionMessageConstant
```

```
<ReplaceOldKeepBefore>
Format(PdfException
<ReplaceNewKeepBefore>
Format(
LayoutExceptionMessageConstant
<ReplaceEnd>
```

```
<ReplaceOld> format
<ReplaceNew> Format
<ReplaceEnd>
```

```
<ReplaceOldKeepBefore>
Format(PdfException
<ReplaceNewKeepBefore> Format(
LayoutExceptionMessageConstant
<ReplaceEnd>
```

```
...
String.Format(
LayoutExceptionMessageConstant..
```

Figure 3.2: Workflow of CodeEditor for multilingual co-evolution. CodeEditor leverages the context of code change histories of multiple programming languages from three sources: code changes on the source programming language (E_S), the old version of code in the target programming language ($M_{T;old}$), and the new version of code in the source programming language ($M_{S;new}$). CodeEditor has two variants that both generate the code changes in the target programming language (E_T) but in different formats: EditsTranslation directly generates the code changes; MetaEdits generates the meta edit plan which edits E_S to E_T, followed by the code changes. Finally, we apply the code changes (E_T) on the old version of code ($M_{T;old}$) to obtain the new version of code ($M_{T;new}$) in the target programming language.

3.3 Model

We present the overview of the proposed CodeEditor model in Figure 3.2. CodeEditor is built upon the encoder-decoder framework which consists of a transformer-based encoder and a transformer-based decoder [106]. Many conditional generation tasks, including code summarization and translation, are being addressed with encoder-decoder models [2, 35, 70, 110, 111].

We initialize CODEDITOR's parameters with the pretrained language model CoditT5 (Chapter 2). CoditT5 has shown promising results on various software-related editing tasks *in a single programming language*, but nonetheless would provide us with a "warm-start" that carries the necessary inductive biases towards modeling edits across programming languages. To adapt to the multilingual co-evolution task, we then fine-tune the CODEDITOR model exploring two key components: (i) the context fed into the model; (ii) the output format of the model.

To encourage our CODEDITOR model to leverage the (synchronous) code change histories of multiple programming languages in its training data, we provide the model with context from three sources as shown in Figure 3.2: (i) code changes on source programming language (E_S); (ii) old version of the code written in target programming language ($M_{T;old}$); (iii) new version of the code written in source programming language ($M_{S;new}$).

We explore two formats to represent the generated code changes: (i) the code edits in the target programming language (E_T); (ii) a meta edit sequence that translates the code edits from the source programming language to the target programming language, followed by the code edits in the target programming language (this is similar to the output format of CoditT5). In both cases, we then apply the generated code edits in the target programming language (E_T) to the old version of the code ($M_{T;old}$) to obtain the new version of the code ($M_{T;new}$).

3.3.1 Edit Representations

3.3.1.1 Concise Edit Sequence

We first represent edits using a sequence of edits identical to that used in CoditT5 in Chapter 2, which we call concise edit sequence. Each edit is represented

Table 3.1: The mappings between concise edit sequence and unambiguous edit sequence.

Edit	**Concise**	**Unambiguous**
Insertion	`<Insert>`	`<ReplaceKeepBefore>` `<ReplaceKeepAfter>`
Deletion	`<Delete>`	`<Delete>` `<ReplaceKeepBefore>` `<ReplaceKeepAfter>`
Replacement	`<Replace>`	`<Replace>` `<ReplaceKeepBefore>` `<ReplaceKeepAfter>`

as:

```
<Operation> [token span] <OperationEnd>
```

Here, `<Operation>` is either `Insert`, `Delete` or `Replace`. Note that the `Replace` is represented in a slightly different structure since we must specify both the old contents to be replaced and the new contents to replace with:

```
<ReplaceOld> [old contents] <ReplaceNew>
 [new contents] <ReplaceEnd>
```

For example, in Figure 3.1, the code change on the old Java method can be represented by "`<ReplaceOld> PdfException <ReplaceNew> LayoutExceptionMessageConstant <ReplaceEnd>`". We use `difflib` [28] to compute the set of minimal edit sequence from the old and new versions of code.

3.3.1.2 Unambiguous Edit Sequence

One drawback of the concise edit sequence specified above is that it can be ambiguous due to the absence of positional information. For example, the Java code change in Figure 3.1 can be represented using `Replace` as: "`<ReplaceOld> PdfException <ReplaceNew> LayoutExceptionMessageConstant <ReplaceEnd>`". Without further specification, the edit does not contain any clues regarding which `PdfException` should be replaced as there are two occurrences of `PdfException` in the old code sequence. For similar reasons, `Delete` is ambiguous in cases where multiple occurrences of token spans can be removed and `Insert` is always ambiguous because of not indicating where to add the new contents.

To eliminate the potential ambiguity in the concise edit sequence, we design the format of unambiguous edit sequence by adjusting the condensed edit sequence proposed by Panthaplackel et al. [77], which uses anchor tokens to specify the location to perform edits.

Insertion. We do not use `Insert` since it will always introduce ambiguity without location information. To represent insertion, we first find unique anchor tokens that are the shortest span of tokens that is either before or after the edit location and is unique in the input sequence. Then we use `ReplaceKeepBefore` or `ReplaceKeepAfter`, which represents replacing the anchor tokens with the inserted contents and the anchor tokens. For example, in Figure 3.1, suppose the Java code change entails adding a blank return statement after the `assertEquals` statement on line 7. The token span "`getMessage());`" will serve as the minimal span of anchor tokens because it is unique among the old Java code sequence, and it occurs right before the edit to be performed. We disambiguate the edit sequence:

```
<Insert> return; <InsertEnd>
```

with the unambiguous edit sequence:

```
<ReplaceOldKeepBefore> getMessage());
<ReplaceNewKeepBefore> getMessage()); return;
<ReplaceEnd>
```

This edit sequence indicates that "`getMessage());`" should be replaced with "`getMessage()); return;`". We introduce `ReplaceKeepBefore` operation where the tokens that follows the `<ReplaceOldKeepBefore>` should be removed and the tokens following `<ReplaceNewKeepBefore>` should be inserted. Different from `Replace`, there is some overlap between the tokens to be removed and tokens to be inserted. If anchor tokens do not exist before the edit location, we use `ReplaceKeepAfter` with the tokens after the edit location instead.

Replacement. If the span of tokens to be replaced is unique in the old sequence, regular `Replace` sequence is sufficient and deterministic; in that case we will keep using it. Otherwise, it is unclear which occurrence of token span should be replaced. As an example, in Figure 3.1, the Java code change is changing from `PdfException` to `LayoutExceptionMessageConstant` in the `assertEquals` statement on line 6. The replacement in the concise edit sequence is ambiguous because there are two usages of `PdfException` (on lines 5 and 6) in the old Java code sequence after tokenization. To address this, similar to the insertion case, we search for the minimal anchor tokens before or after the edit location that can form a unique span in the old sequence. For example, the concise edit sequence:

```
<ReplaceOld> PdfException <ReplaceNew>
LayoutExceptionMessageConstant <ReplaceEnd>
```

can be disambiguate into the following unambiguous edit sequence:

```
<ReplaceOldKeepBefore> format(PdfException
<ReplaceNewKeepBefore> format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

Deletion. Similar to replacement, if the span of tokens to be deleted is unique across the old sequence, we will keep using `Delete` because it is unambiguous. Otherwise, it will be transformed to `ReplaceKeepBefore` or `ReplaceKeepAfter`. For example, suppose "`PdfException.`" should be removed from the old Java method on line 6 in Figure 3.1. The concise edit sequence:

```
<Delete> PdfException.  <DeleteEnd>
```

will be transformed to:

```
<ReplaceOldKeepBefore> format(PdfException.
<ReplaceNewKeepBefore> format( <ReplaceEnd>
```

This edit sequence indicates that "`format(PdfException.`" should be replaced with "`format(`", unambiguously implying the deletion of "`PdfException.`".

To summarize, the unambiguous edit sequence contains 4 types of edits: `<Replace>`, `<Delete>`, `<ReplaceKeepBefore>` and `<ReplaceKeepAfter>`. The mappings between concise edit sequence and unambiguous edit sequence are summarized in Table 3.1. The unambiguous edit sequence can be applied to the old code to derive the new version of code deterministically.

3.3.2 Model Input

We aim to build performant machine learning models for the multilingual co-evolution task by providing the model with code evolution information, namely the revisions of code of both source and target programming languages. Instead of directly translating the entire code snippet between programming languages, Codeditor translates the code *changes* between programming languages.

3.3.2.1 Source Code Edits

To encourage the model to learn the alignment between developer-made changes across programming languages, we provide Codeditor with code changes in the source programming language (E_S). To maintain both precision and conciseness of the edits, we adopt the unambiguous edit sequence (Section 3.3.1.2) to represent the code changes. As shown in Figure 3.2, the Java code changes (E_S) of replacing the `PdfException` with `LayoutExceptionMessageConstant` is structured in the form of

```
<ReplaceOldKeepBefore> format(PdfException
<ReplaceNewKeepBefore> format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

3.3.2.2 History-Related Context

In addition to the edit representation of code changes in source programming language (E_S), we provide CODEDITOR with the old code in target programming language ($M_{T;old}$) to better help the model to infer the correlated code changes in the target programming language. The intuition is that the model will reason about how to transfer and adjust the edits in source programming language grounding the specific implementation of the method in target programming language.

Furthermore, we append the new code in source programming language ($M_{S;new}$) as one of the contexts. We believe this will give the model more context to understand the edits in source programming language and promote the consistency of the updated methods in two programming languages.

To sum up, we combine history-related context from three sources: code changes in the source programming language (E_S), old code in the target programming language ($M_{T;old}$), and new code in the source programming language ($M_{S;new}$). We concatenate them into a sequence separated by a special SEP token as the model input.

3.3.3 Model Output

We propose two formats as the model's target output which lead to two modes of CODEDITOR: *EditsTranslation* and *MetaEdits*. Both modes use the same input and both modes' target outputs entail a sequence of edits on the target programming language.

EditsTranslation. The output of EditsTranslation mode is the unambiguous edit sequence in target programming language which suggests how the code in target programming language should be changed. Note that the model-generated unambiguous edit sequence can be parsed and applied to old version of code deterministically. EditsTranslation essentially learns to translate the code edits from the source programming language (E_S) to the target programming language (E_T) grounding the provided code history context. EditsTranslation mode's target output for the C# example in Figure 3.1 is:

```
<ReplaceOldKeepBefore> Format(PdfException
<ReplaceNewKeepBefore> Format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

MetaEdits. In this mode, we adopt the output format of CoditT5 for multilingual co-evolution since our model is built upon CoditT5, and it had showed promising performance on software-related editing tasks. CoditT5 is pretrained to generate the following output format: "[Edit Plan] `<s>` [Target Sequence]". The edit plan is a concise edit sequence that represents the steps to edit the input sequence; the target sequence is the edited sequence after applying the proceeding edit plan. We tailored this format to the multilingual co-evolution task; the edit plan represents the edits between the code edits on source programming language (E_S) and target programming language (E_T) which we call the *meta edit sequence*. And the final target sequence should be the unambiguous edit sequence on the target programming language (E_T). For the example in Figure 3.1, the expected meta edit sequence that converts Java edit to C# edit is the following:

```
<ReplaceOld> format <ReplaceNew> Format <ReplaceEnd>
```

The target sequence after applying the meta edit sequence is:

```
<ReplaceOldKeepBefore> Format(PdfException
<ReplaceNewKeepBefore> Format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

Note that during inference, we only use the target unambiguous edit sequence to get the updated code in target programming language as MetaEdits mode's prediction.

3.4 Dataset

This is the first work to consider the evolution of software projects in the multilingual setting; hence, we also created a new dataset that includes aligned code changes between programming languages. As the first step, we build the dataset by mining histories of the open-source Java and C# projects. We first collect the changed methods from the commits of the Java and C# projects. We then design heuristics to pair (i.e., align) those changes on methods with similar implementations and functionalities. We consider two directions on our dataset: J2CS (updating C# method based on Java changes) and CS2J (updating Java method based on C# changes). In this section, we describe the approach we use to collect the data (Section 3.4.1), split and preprocess data (Section 3.4.2), and finally present the statistics of our dataset (Section 3.4.3).

3.4.1 Data Collection

To build the dataset, we extract aligned Java and C# code changes at the method level as tuples (Java old method; Java new method, C# old method; C# new method).

The code changes are mined from the git commits. We consider 8 open-source projects as listed in Table 3.2 which have both Java and C# implementations and are used in prior work [18, 61, 69]. All the projects were first developed in Java and then ported to C#.

Table 3.2: Open-source projects used in our dataset and number of examples from each project.

Java Project	**C# Project**	**Count**
antlr/antlr4	tunnelvisionlabs/antlr4cs	12
apache/lucene	apache/lucenenet	40
apache/poi	nissl-lab/npoi	5
eclipse/jgit	mono/ngit	808
formicary/fpml-toolkit-java	formicary/fpml-toolkit-csharp	20
itext/itext7	itext/itext7-dotnet	5,121
quartz-scheduler/quartz	quartznet/quartznet	17
terabyte/jgit	mono/ngit	590
SUM		6,613

To collect the paired changes, we first assign a unique identifier to each method in the projects (for both Java and C# projects) based on the method signature, class name and path to the file where the method is defined. Similar to the strategy used by Lu et al. [61], we then pair the Java methods and C# methods according to the similarity of their unique identifiers. This strategy is effective because the ported C# project has very similar structure and naming rules for classes and methods to the corresponding Java project.

We use the following rules to extract the aligned code changes:

1. For each Java method change, we extract the code changes in the paired C# method that happen no later than 90 days of the Java change as the *possible matched code change*. We use the commit date as the time of the change.
2. To filter unrelated code changes, we compute the Jaccard similarity [47] between C# and Java added and deleted lines. We further refine the filtering by sub-tokenizing these lines based on camelCase conventions (e.g., `lastModified` to `last modified`) and compute Jaccard similarity only for the added and deleted tokens. We only keep possible matched code changes that have the token-level Jaccard similarity higher than 0.4 and the line-level Jaccard similarity higher than 0.5.

3. For each Java code change and C# code change, we only select the most similar corresponding code change if there are multiple possible matched code changes.

3.4.2 Data Preprocessing and Splitting

For both Java and C# methods, we remove the inline natural language comments and tokenize the method into tokens using the language-specific lexers generated by ANTLR [81].

We envision the following use case for the machine learning model: whenever a developer makes a change in the project written in the source programming language, the developer will use the model trained on the existing historical aligned code changes to migrate that change to projects written in other target programming languages. To evaluate the models under this use case, following the recommendations from prior work [71], we split the dataset into training, validation and test sets using the *time-segmented* approach. Namely, the changes in the training data took place before the changes in the validation set, which in turn took place before the changes in the test data for evaluation. More specific, for each Java and C# code change pair, we first collect the time of the C# commit and then sort the code change pairs in chronological order. We then select the oldest 70% of the code change pairs from each project as training data, next oldest 10% as validation data, the remaining as test data.

To more rigorously assess the generalization capabilities of the models, we also evaluated them when splitting the dataset using the *cross-project* approach [71], which is frequently used in prior work on machine learning models for code. Specifically, the aligned code changes in the training set are from different projects compared to those in the validation and test sets.

Table 3.3: Statistics of our dataset. Number of examples of training, validation and test data; average number of tokens in the old version of method and new version of method; average number of edits for the code change; average number of added and deleted tokens.

		Training	**Validation**	**Test**
Count		4,391	623	1,599
Java	Avg. len(M_{old})	193.05	192.88	159.06
	Avg. len(M_{new})	195.99	192.36	159.37
	Avg. # edits	2.71	2.68	2.43
	Avg. # add. tokens	19.57	16.64	10.90
	Avg. # del. tokens	16.62	17.16	10.59
C#	Avg. len(M_{old})	200.37	199.71	168.60
	Avg. len(M_{new})	203.49	199.47	169.22
	Avg. # edits	2.73	2.75	2.47
	Avg. # add. tokens	20.30	17.69	11.86
	Avg. # del. tokens	17.18	17.92	11.25

3.4.3 Statistics

The statistics of the collected dataset are shown in Table 3.3. We present the number of examples in the training, validation, and test dataset using time-segmented split approach. We show the average number of tokens in the old methods (Avg. len(M_{old})) and new methods (Avg. len(M_{new})) after tokenization by the lexers. To measure the size of the code changes, we calculate the average number of added tokens (Avg. # add. tks) and deleted tokens (Avg. # del. tks) in the changed Java and C# methods as well as the average number of edits (Avg. # edits) needed for those changes. For computing these edit-related statistics, we represent the code changes using concise edit sequences (Section 3.3.1.1).

For both Java and C# code changes, the difference between average number of added tokens and deleted tokens is usually small, fewer than 4 tokens. Similarly, we find that the average number of edits needed is fewer than 3 and the edits happened in the newer commits are generally smaller than prior ones. This is expected as the software projects are becoming more stable as they evolve, and thus there will be smaller code changes to be made. For evaluation, we run all the models and baselines on this dataset in two directions: (1) updating C# method based on Java changes, and (2) updating Java method based on C# changes. We denote the former one as J2CS and the latter one as CS2J.

3.5 Experiments

In this section, we describe the baselines we compare to with our Codeditor model (Section 3.5.1), the evaluation metrics (Section 3.5.2) and the detailed experiment setup (Section 3.5.3).

3.5.1 Baselines

We evaluate our approach against rule-based models, pretrained encoder-decoder models, the state-of-the-art code-editing model (which targets a single programming language), and large generative models pretrained on billions of lines of code.

Copy. This is a rule-based model which copies the old code in target programming language ($M_{T;old}$) as the prediction. This is not a trivial baseline since there are quite a few examples in the dataset that entail small edits between two versions. We include this to benchmark the models that actually update the code.

CopyEdits. Based on our observations, there are cases where the code change in source programming language (E_S) is exactly the same as the change in target programming language (E_T) , such as changing the variable name or updating the log message. This rule-based model copies the E_S and directly applies it to the old code in target programming language ($M_{T;old}$).

CodeT5-Translation. We consider a state-of-the-art model that does not have access to the code change history. Namely, a code translation model that translates code between the programming languages (from $M_{S;new}$ to $M_{T;new}$). We use CodeT5 [110], an LLM pretrained on large amount of developer-written code from GitHub, which we fine-tune on our constructed dataset.

CodeT5-Update. This model has the same architecture as CodeT5-Translation except that we supply it with code change history. The model input is the same as for our CODEDITOR models, i.e., with extra context of the old code in target programming language ($M_{T;old}$) and the code change in source programming language (E_S). Different from CODEDITOR model, it is trained to directly generate the new code in target programming language ($M_{T;new}$).

CoditT5. This is the state-of-the-art model for software editing tasks. It has the same model architecture and input as CODEDITOR, while the output consists of the edit plan to represent the edits on the target programming language and the target sequence which represents the updated code ($M_{T;new}$) after applying the edit plan.

Codex-few-shot [17]. Large pretrained generative models such as GPT-3 [12] have shown impressive results under the context of *few-shot learning* or even *zero-shot learning* on various generation tasks. They are able to generalize to new tasks they have not seen during pretraining with only a few or even no labeled examples. To compare the fine-tuned CODEDITOR model with generative models, we include Codex, a large generative model built on GPT-3 and is further pretrained on billions of GitHub data. Following prior work [3, 50], for each example in test data, we randomly select several labeled examples in the training data as the context. Note that the labeled examples are selected from the same project as the test data. For J2CS dataset, each labeled example is formed as: “Java: $M_{S;old} \Rightarrow M_{S;new}$ C#: $M_{T;old} \Rightarrow M_{T;new}$” to inform the model the aligned updates between two programming languages. The designed prompt for inference is “Java: $M_{S;old} \Rightarrow M_{S;new}$ C#: $M_{T;old} \Rightarrow$ ”. The model output is the prediction for the new code in target programming language ($M_{T;new}$). To conform to the required input length limit, we include 2 labeled examples in the prompt.

gpt-3.5-turbo-zero-shot [74]. gpt-3.5-turbo is an upgraded version of GPT-3 model and is further fine-tuned for conversation generation following human instructions with the help of supervised and reinforcement learning methods. It has showed strong performance on code completion benchmarks like HumanEval and MBPP [1, 5, 17]. For each example in test data, we provide instructions including both the previous and the updated versions of the code written in the source programming language, subsequently prompting gpt-3.5-turbo to update the old code in the target programming language accordingly. For J2CS dataset, the prompt is formed as: "The developer updates the Java method from: $M_{S;old}$ to: $M_{S;new}$. Please update the C# method accordingly. This is the old C# method: $M_{T;old}$."

3.5.2 Evaluation Metrics

Following prior work [2, 77, 110], as well as our work presented in Chapter 2, we use metrics for evaluating the quality of code generation: BLEU [80], CodeBLEU [89], xMatch, and metrics for evaluating the quality of software editing: SARI [114] and GLEU [68]. Note that for all the metrics we report in this chapter, they range from 0 to 100 and higher scores are better.

xMatch. When the generated code matches exactly with the expected code in target programming language, this metric is 100; otherwise, this metric is 0. This metric reflects the percentage of exact matches among the models' predictions on test data.

BLEU. It is a widely used metric originally proposed for evaluating the quality of machine translation. It measures the n-gram overlap between the generated sequence and the expected one. Concretely, we report the 1$\sim$4-grams overlap between the tokens in the predictions and tokens in the ground truth.

CodeBLEU. The metric is proposed for evaluating the quality of code generation. In addition to measuring the n-gram overlap, it considers the overlap of the Abstract Syntax Tree (AST) and data-flow graph between generated code and the expected code.

SARI. It measures quality of the systems that are designed to make edits. Specifically, it is computed as the average of the F1 score for kept and inserted spans of tokens, and the precision of deleted spans of tokens.

GLEU. It is a variant of BLEU. It was originally proposed for grammatical error correction and designed for rewarding the correct edits while penalizing the incorrect ones.

3.5.3 Experiment Setup

We run all experiments on machines with 4 NVidia 1080-TI GPUs, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz for training. We implement our models using PyTorch 1.9.0. All the hyper-parameters of the CodeT5 and CoditT5 baselines are set to the same values as in prior work [110, 122]. For CODEDITOR, CodeT5-Translation, CodeT5-Update, and CoditT5, we early stop the training when the BLEU score on the validation set does not improve for 5 epochs, and use beam search with beam size 20 during inference. For Codex and gpt-3.5-turbo, we set temperature to 0.2 during inference.

Note that Codex and gpt-3.5-turbo are closed-source and may be updated/deprecated over time. We use gpt-3.5-turbo-0613 [2] in this chapter. We used the code-davinci-002 version of Codex when performing experiments in the time-segmented split; however, OpenAI deprecated Codex in March 2023 before we could complete our experiments in the cross-project split, as such we did not include Codex in this part of results.

3.6 Results

We organize our evaluation around three main research questions:

RQ1: What is the benefit of using code change history in multilingual co-evolution?

[2]https://platform.openai.com/docs/models/gpt-3.5-turbo

Table 3.4: Results on the J2CS dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	**BLEU-4**	**CodeBLEU**	**SARI**	**GLEU**
Copy	0.00	83.11	90.42	30.68	74.58
CopyEdits	38.21^{β}	$90.29^{\alpha\chi}$	91.34	76.92	87.93
CodeT5-Translation	38.02^{β}	87.45	77.15	83.77	85.59
CodeT5-Update	60.41^{ϵ}	$90.00^{\chi\eta}$	76.63	80.11	88.72
CoditT5	60.29^{ϵ}	$89.84^{\alpha\eta}$	75.20	80.99	89.29
Codex-few-shot	48.84	80.71	59.63	72.80	79.74
gpt-3.5-turbo-zero-shot	29.52	85.60	73.00	68.44	84.74
CODEDITOR (MetaEdits)	63.48	94.55	94.78	85.63	93.20
CODEDITOR (EditsTranslation)	67.23	95.44	96.02	**87.23^{δ}**	94.21
Hybrid	**71.79**	**96.12**	**96.09**	87.08^{δ}	**95.07**

RQ2: How does our edit-based model, CODEDITOR, compare to generation-based models for the multilingual co-evolution?

RQ3: How can a generation-based model complement CODEDITOR model to further improve the performance?

3.6.1 Quantitative Analysis

In tables 3.4-3.7, we present results for baselines and our proposed CODEDITOR models on J2CS, CS2J for both time-segmented and cross-project splits. We conducted statistical significance testing through bootstrap tests [9] under confidence level 95%.

RQ1: Contribution of code change histories. We divide models into two categories with respect to whether a model has access to the information on code change histories: Copy and CodeT5-Translation are history-agnostic models, and the remaining are history-aware models. Overall, the history-aware models outperform the history-agnostic ones. The rule-based model CopyEdits, which directly applies the code change in source programming language (E_S) to the old code in target programming language without any adaptation, has comparable performance to the

Table 3.5: Results on the CS2J dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	**BLEU-4**	**CodeBLEU**	**SARI**	**GLEU**
Copy	0.00	83.06	89.82	30.66	74.55
CopyEdits	38.15	89.36^{α}	90.31^{β}	75.86	87.02^{χ}
CodeT5-Translation	40.21	89.10^{α}	77.99^{β}	83.99	87.21^{χ}
CodeT5-Update	55.97	90.62	76.38^{γ}	79.65	89.72
CoditT5	60.98	90.88	75.87^{γ}	81.41	90.15
Codex-few-shot	55.53	82.54	60.35	76.23	82.13
gpt-3.5-turbo-zero-shot	32.52	86.95	76.01	69.05	86.33
CODEDITOR (MetaEdits)	**68.61**$^{\epsilon\eta}$	93.98	94.43	85.74	92.61
CODEDITOR (EditsTranslation)	$67.92^{\delta\epsilon}$	95.29	94.83	**86.24**	94.23
Hybrid	$67.67^{\delta\eta}$	**96.44**	**95.36**	84.46	**95.75**

machine learning history-agnostic model CodeT5-Translation. This emphasizes the importance of contextual information provided by code change histories in multilingual co-evolution. Interestingly, we find that Codex-few-shot, which is used under the few-shot learning setting without fine-tuning, performs better than fine-tuned CodeT5-Translation on xMatch, while worse than other history-aware fine-tuned machine learning models. This again underlines the value of code change histories and suggests that fine-tuning will give better performance by leveraging more code history contexts in the training data.

RQ2: CODEDITOR vs. generation-based models. Among all the history-aware models, machine learning models, such as CodeT5-Update and CoditT5, achieve much higher performance than the rule-based CopyEdits, which demonstrates that the machine learning models effectively learn to reason about the correlated code changes and adjust them to the target programming language. We observe that CODEDITOR (in both EditsTranslation and MetaEdits modes), which is trained to first translate code changes on source programming language to target programming language and then apply the edits to the old code in target programming language, achieves even higher performance across all the metrics than the large pretrained generation-based

Table 3.6: Results on the cross-project split using J2CS dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	**BLEU-4**	**CodeBLEU**	**SARI**	**GLEU**
Copy	0.00	79.96^{α}	89.08	30.53	71.89
CopyEdits	14.19	87.95	89.73	67.58	85.55
CodeT5-Translation	10.64	77.34	64.35	71.73	74.16
CodeT5-Update	29.38	80.56^{α}	66.15	64.70	79.65
CoditT5	34.59	81.59	65.17	83.29	80.91
gpt-3.5-turbo-zero-shot	39.58	86.97	74.90	70.15	86.14
CODEDITOR (MetaEdits)	38.36	90.79^{β}	91.60	73.45	88.94^{χ}
CODEDITOR (EditsTranslation)	41.91	90.86^{β}	91.35	74.59	88.94^{χ}
Hybrid	**43.35**	**92.51**	**91.70**	**89.13**	**91.18**

model (CodeT5-Update) which directly generates the new code in target programming language from scratch. This highlights that the models that are trained to explicitly perform edits by predicting the edit sequence are better suited for editing tasks in the software domain than generation-based models.

To further investigate the advantages of CODEDITOR over the best generation-based model (CodeT5-Update), we break down the performance of EditsTranslation and CodeT5-Update on each example in the test data of J2CS. In Figure 3.3a, we show the average percentage of CODEDITOR (EditsTranslation) and CodeT5-Update's predictions that exactly match the ground truth with respect to the number of subtokens in the input old code ($M_{T;old}$). Note that the code are subtokenized using the Roberta tokenizer [58], which is used by both models. We exclude the examples that have more than 500 subtokens from being shown in this figure as those outliers only account for less than 5% of the test data. We can see that the performance of CodeT5-Update drastically drops with the increase of number of subtokens in the code to be edited ($M_{T;old}$), but EditsTranslation's performance is rather stable. This illustrates another benefit of CODEDITOR in accurately handling longer input, because of focusing on transforming the edits instead of generating the entire new code like

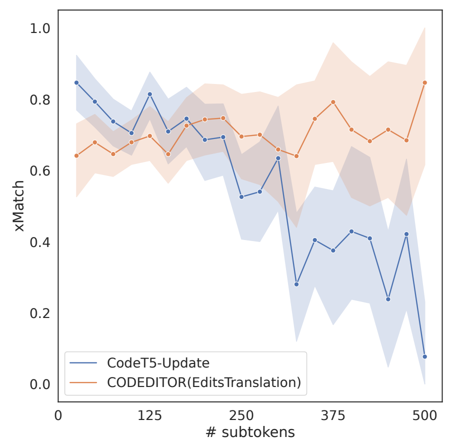
Table 3.7: Results on the cross-project split using CS2J dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	**xMatch**	**BLEU-4**	**CodeBLEU**	**SARI**	**GLEU**
Copy	0.00	80.02	88.60	30.51	71.94
CopyEdits	13.86	86.99	88.70	66.50	84.14
CodeT5-Translation	6.21	76.84	62.27	67.37	69.81
CodeT5-Update	31.60	81.98	65.82	65.49	81.07
CoditT5	35.81	82.89	65.20	83.27	81.67
gpt-3.5-turbo-zero-shot	39.80	89.35	76.69	72.36	88.62
CODEDITOR (MetaEdits)	41.91	91.54^{α}	91.21	73.46	89.36^{β}
CODEDITOR (EditsTranslation)	40.35	91.63^{α}	90.99	74.15	89.58^{β}
Hybrid	**46.34**	**93.17**	**91.32**	**89.62**	**91.24**

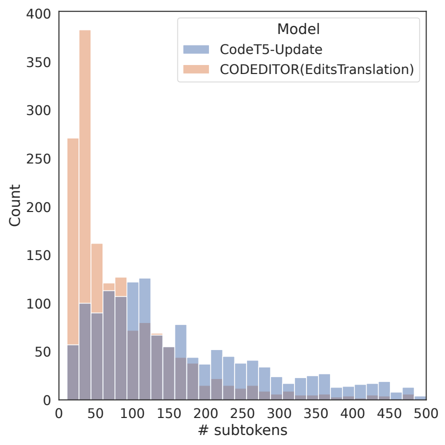
CodeT5-Update.

Meanwhile, most of the existing transformer-based models have a length limit for the input sequence because the naive self-attention has quadratic complexity with regard to the input length. In Figure 3.3b, we present the distribution of the number of subtokens in the models' target outputs for CODEDITOR (EditsTranslation) and CodeT5-Update on the test data of J2CS. We only show the distribution of target outputs with fewer than 500 subtokens for the same reason described in the previous paragraph. Most of CODEDITOR's target outputs (the sequence of edit operations) are shorter than CodeT5-Update's output (new code in target programming language). This might explain why CODEDITOR achieves better performance than generation-based models on longer code as generating longer sequence are generally more challenging to machine learning models. Recent studies [8, 10, 21] have focused on exploring approaches to address the limitation of the model's input context window size. Future research should examine the performance difference between translating edit sequences and generating entirely new code using models capable of handling longer context.

RQ3: Combining generation-based model with CODEDITOR . To exploit



(a) Average percentage of model's predictions that exactly match the ground truth on examples that have different number of subtokens. The bands represent the 95% confidence interval.



(b) Distribution of number of sub tokens in models' target outputs.

Figure 3.3: Comparison of model performance by input length (left) and distribution of output token lengths (right).

the superiority of generation-based model on short code snippets, we combine our strongest generation model, CodeT5-Update, with the strongest CODEDITOR model, EditsTranslation, based on the size of the code snippet. Specifically, we use CodeT5-Update if the code to be updated has fewer subtokens than a threshold and use CODEDITOR (EditsTranslation) otherwise. To pick the threshold for combining two models, we performed a grid-search on the validation set and selected the one that gives optimal xMatch score.

We refer to the combined model as the *Hybrid* model and provide its results on the bottom row of Table 3.4 to Table 3.7. By combining generation-based model with CODEDITOR, we can achieve improved performance on most of the reported automatic metrics.

3.6.2 Qualitative Analysis

Figure 3.4 shows an example in J2CS dataset and the models' predictions. We show the code changes from Java project `itext/itext7` in the method (`parseBodyFragment`). The newly added code is highlighted in green and removed code is highlighted in red. We also present the old version of the corresponding C# method (`ParseBody-Fragment`) from `itext/itext7-dotnet`, and the predicted code changes from three models: CODEDITOR (EditsTranslation), CodeT5-Update, CodeT5-Translation. Note that CodeT5-Translation only has access to the new version of Java method.

```
public static Document parseBodyFragment(String bodyHtml, String baseUri) {
  ...
  List <Node> nodeList = parseFragment(bodyHtml, body, baseUri);
  Node[] nodes = nodeList.toArray(new Node[0]);
  - for (int i = nodes.length - 1; i > nodeList.size(); i--) {
  + for (int i = nodes.length - 1; i > 0; i--) {
      nodes[i].remove();
  }
  ...
}
```

Java Change

```
public static Document ParseBodyFragment(String bodyHtml, String baseUri) {
  ...
  IList <iText.StyledXmlParser.Jsoup.Nodes.Node> nodeList = ParseFragment(
      bodyHtml, body, baseUri);
  iText.StyledXmlParser.Jsoup.Nodes.Node[] nodes = nodeList.ToArray(new iText.
      StyledXmlParser.Jsoup.Nodes.Node[nodeList.Count]);
  for (int i = nodes.Length - 1; i > nodeList.Count; i--) {
      nodes[i].Remove();
  }
  ...
}
```

C# Old Method

```
...
- for (int i = nodes.Length - 1; i > nodeList.Count; i--) {
+ for (int i = nodes.Length - 1; i > 0; i--) {
...
```

Codeditor (EditsTranslation) Prediction

```
...
- iText.StyledXmlParser.Jsoup.Nodes.Node[] nodes =
    nodeList.ToArray(new iText.StyledXmlParser.Jsoup.Nodes.Node[nodeList.Count]);
+ iText.StyledXmlParser.Jsoup.Nodes.Node[] nodes =
    nodeList.ToArray(new iText.StyledXmlParser.Jsoup.Nodes.Node[0]);
- for (int i = nodes.Length - 1; i > nodeList.Count; i--) {
+ for (int i = nodes.Length - 1; i > 0; i--) {
...
```

CodeT5-Update Prediction

```
...
- iText.StyledXmlParser.Jsoup.Nodes.Node[] nodes =
    nodeList.ToArray(new iText.StyledXmlParser.Jsoup.Nodes.Node[nodeList.Count]);
+ Node[] nodes = nodeList.ToArray(new Node[0]);
- for (int i = nodes.Length - 1; i > nodeList.Count; i--) {
+ for (int i = nodes.Length - 1; i > 0; i--) {
...
```

CodeT5-Translation Prediction

Figure 3.4: Qualitative analysis of all the models on one example in the test data of J2CS dataset.

Although CodeT5-Translation is able to correctly translate the code change in Java, it fails to infer the full name of the type `Node` and makes an irrelevant edit, because it does not have the context of the old version of C# code. CodeT5-Update correctly captures the Java change while making an extra irrelevant edit on the C# code. Our proposed model, CODEDITOR (EditsTranslation) accurately identifies the position in the C# method to make edits and correctly adjusts the Java edits.

3.7 Limitations

Studied programming languages. We study the translation of code changes between two programming languages. In this Chapter, we focus on open-source Java and C# projects due to the ease of locating corresponding changes using heuristics. Nevertheless, it is important to note that our approach can be applied to other programming language pairs as well, and we leave the investigation of such pairs for future research.

Correspondence between programming languages. Our model, CODEDITOR, is intended for developers to migrate code changes from a project written in a source programming language to projects written in target programming languages, leveraging known correspondences (e.g., methods with similar functionalities) between the source and target programming languages. In this work, we adopt a similar strategy used in [61] to match Java and C# methods. In practice, a code retrieval system can be used as a first step to identify the locations where the code changes should be propagated. We leave the combination of code retrieval tool and CODEDITOR as future work.

Empirical evaluation. This chapter presents the empirical study results for internal metrics that are of interest to researchers. However, the external measurements of the impact on software engineering effort are not included in this study. These measurements could be addressed by conducting user studies.

3.8 Conclusion

We formulated a new task: translating code changes across programming languages with the goal to synchronize projects that provide the same APIs or implementations in multiple programming languages. We proposed CODEDITOR, a model which uses code change history as contextual information and learns to make edits on the existing version of code written in the target programming language to help with the multilingual code co-evolution. We showed that our model outperforms existing code translation models and is better than the generation-based models even if they use historical context. CODEDITOR is a significant advancement in supporting developers with the maintenance of their projects that incrementally provide identical functionalities in multiple programming languages.

Chapter 4: Related Work

In this chapter, we describe work most closely related to this dissertation. First, we review the research on modeling edits with machine learning techniques (Section 4.1). Next, we describe research on pretrained LLMs for code-editing tasks (Section 4.2). Then, we discuss rule-based code translation tools (Section 4.3), existing machine learning models designed for code translation (Section 4.4), and machine learning models proposed to reduce software evolution (Section 4.5).

4.1 Learning Edits

Prior work has studied learning edits in both natural and programming languages. We followed the approach of explicitly representing edits as sequences with edit actions. Our edit representation is inspired by Panthaplackel et al. [77, 79], who studied learning comment editing based on code edits. Brody et al [11], Tarlow et al [95], Chen et al [19], and Yao et al [116] represented code as ASTs (abstract syntax trees) and the code edits as edit actions over the AST nodes rather than tokens. We do not focus on editing structured data (AST) as it can not be easily combined with large pretrained models which are primarily based on sequence of tokens. To model code edits, instead of generating entirely new code, different LLMs adopt different edit formats [96]. For example, after the publication of the results in this dissertation, Gemini family of models [98] use "diff-fenced" format where LLMs should specify the series of search and replace blocks with the path to the file. The GPT-4 Turbo family of models [75] use the widely used unified diff format similar to the git diff format because of its efficiency.

Alternatively, edits can be encoded into vector representations (or embeddings). Guu et al. [37] studied learning edit embeddings for natural language generation in a prototype-then-edit style. Yin et al. [118] studied learning code edits as embeddings and then applying them to natural language editing tasks and code bug fixing. Hashimoto et al. [39] developed a retrieve-and-edit framework for text-to-code generation, where the edits are learned as parameters of a seq2seq model. Similarly, Li et al. [54] proposed a retrieve-and-edit framework for code summarization task where the model first learns an edit vector and then generates the revised summary conditioned on it. Although learning edits as embeddings can be effective for individual tasks, it is not suitable to be used in the pretraining and fine-tuning paradigm, because there is a large domain gap between the edit embeddings learned on different tasks. Moreover, edit embeddings are less explainable compared to the explicit edit representations we use.

Another line of work that carries out the idea of learning edits is copying mechanism, including copying individual tokens [33, 107] and spans [78, 128], which helps the model to "keep" unchanged tokens and focus on generating the edited part. Iv et al. [46] built a T5-based model to update the existing articles based on the given new evidence. The model is trained to output a *copy* token instead of the copied sentence and a special *reference* token before the updated text which identifies the evidence to support the update. Ding et al. [22] trained the model to emit pointers that indicate the positions for editions and new tokens to be inserted at the same time. Similarly, Tarlow et al. [95] and Chen et al. [19] augmented the transformer-based decoder with pointers to the input graph representation of the code which specify the input locations to edit. Although related, it is orthogonal to our work on learning edits with pretraining.

4.2 Large Language Models for Code Edits

Motivated by the success of large pretrained models for many NLP tasks, domain-specific models that are pretrained on source code and technical text have emerged, including CodeBERT [26], GraphCodeBERT [34], CodeT5 [110], PLBART [2], PyMT5 [20], SynCoBERT [108], Codex [17], UniXcoder [35], CodeLlama [91], etc.

Similar to our approach, GraphCodeBERT, CodeT5, SynCoBERT, and UniXcoder also designed specialized pretraining objectives driven by their targeted tasks.

Prior work already explored applying pretrained models, despite not well-suited, on editing tasks. Chakraborty and Ray [14] used PLBART for code bug fixing, which we compared to in our work. Similarly, Drain et al. [23] further pretrained BART model on 67K Java repositories mined from GitHub and fine-tuned specifically on the bug fixing dataset [102]. Wang et al. [110] and Mastropaolo et al. [65] both pretrained T5 model on CodeSerchNet and used it for bug fixing, which we included as a baseline (CodeT5). Codex [17] showed promising performance on editing tasks by specifying the existing code as a prompt and providing an edit instruction to the model. Tufano et al. [104] and Li et al. [55] both proposed a transformer-based encoder-decoder models pretrained on large code reviewer specific data for code review related tasks including code change quality estimation, review comment generation, and code refinement. While they demonstrate impressive performance on various tasks, none of them are fundamentally well-suited for editing tasks.

4.3 Rule-based Code Translation

Researchers and practitioners have designed rule-based tools for translating the source code between programming languages. Such tools, usually called transpilers, were built for pairs like Java and C# [4], C and Rust [29], C and Go [24]. Nguyen et al. [69] proposed PBSMT, a phrase-based statistical machine translation models for source code translation. Gyori et al. [38] proposed LambdaFicator to translate imperative Java code to using the functional Stream APIs. Radoi et al. [84] presented the rule-based model to translate sequential Java code to MapReduce framework. Prior work [63] has shown that existing rule-based code refactoring tools can only deal with stylized code snippets over common code patterns.

4.4 Learning-based Code Translation

Researchers have proposed various machine learning models for the code translation task. Chen et al. [18] proposed a tree-to-tree neural network with a tree-RNN encoder and a tree-RNN decoder. Motivated by the success of large pretrained LLMs for many Natural Language Processing tasks, domain-specific models that are pretrained on source code and technical text have emerged. Researchers have applied them to the code translation task. Lu et al. [61] proposed CodeXGLUE, a benchmark including the code translation dataset consisting of Java and C# methods with equivalent functionality. They fine-tuned and evaluated CodeBERT on the translation dataset. Results showed that it produced the best results among all the existing baselines. LLMs that are built on the encoder-decoder paradigm and pretrained with general unsupervised denoising auto-encoding objectives showed promising results on wide range of code generation tasks including code translation. Such models include CodeT5 [110], PLBART [2], and UniXcoder [35]. For the comparison of CODEDITOR with state-of-the-art code translation models, we include two variants of the CodeT5-based translation models (with history context and without) in our evaluation.

Researchers designed LLMs that are pretrained with the objective tailored for code translation. Tipirneni et al. [99] introduced tasks on predicting AST paths and data flows during pretraining. Lachaux et al. [51] proposed TransCoder which is pretrained to do code translation with back-translation objective. To improve the quality of pretraining data, Roziere et al. [90] leveraged an automated unit-testing system to filter out invalid generated programs during back-translation. Zhu et al. [129] proposed MuST, which is a multilingual code snippet translation pretraining objective. None of the above work leverages the code change history, which is the main contribution of this dissertation. We leave improving CODEDITOR with pretraining objectives tailored for code translation as future work.

4.5 Software Evolution and Machine Learning

New research initiatives have emerged around building and evaluating models that aid the process of software evolution. Prior work [30, 56, 57, 60, 77] proposed to update the comment given the changes in the associated method, e.g., Panthaplackel et al. [77] built a model that takes the code change as context to make edits on the outdated comment. Nie et al. [71] presented different approaches to split dataset into training, validation, and test sets and studied how different approaches affect the evaluation of machine learning models. Kamezawa et al. [49] presented a dataset, RNSum, which consists of release notes and the associated commit messages collected from GitHub repositories and designed models to generate release notes based on the commit messages. Li et al. [55], Tufano et al. [104], Zhang et al. [124] proposed models that targeted various tasks through the code review process. The models are trained on the historical data and evaluated on the new data pull requests submitted for code review. Our CODEDITOR model incorporates the context from the code changes in source programming language and the old version of method in target programming languages to improve its performance on the multilingual co-evolution task, which helps developers co-evolve the projects implemented in different programming languages.

Chapter 5: Future Work

We now present our visions for future work and research questions that can be explored upon our contributions as described in chapters 2 and 3.

5.1 Edit Representation

In this dissertation, we propose a novel edit output format that includes both the edit operations and the final edited sequence. Training models to generate this format improves performance across multiple software maintenance tasks compared to standard generation models. Exploring alternative edit representations may further enhance model performance.

The current edit plan uses token-level edit sequences, which can be difficult to interpret when changes span multiple lines or files. A more interpretable alternative is line-level edits, such as the unified diff format, which represents code changes at the line level. This format clearly shows both deletions and additions, along with the surrounding context. Another limitation of token-level edits is the lack of code structure, which makes them prone to producing syntactically incorrect edits. For example, an edited span might cut through the middle of a statement, such as "`<Delete> Math.abs( <DeleteEnd>`". A possible solution is to use code *blocks* as the unit of change, leveraging the fact that LLMs perform well at function-level code generation. A code block could be a function, a basic block, or a node in the AST. Natural language descriptions of the intended edits could also be used as edit operations and included in the output. This format may better align with the capabilities of instruction-tuned and reasoning-augmented LLMs.

5.2 Edit Localization

Both CoditT5 and CodeEditor are applied to tasks where the code or comment to be edited is already specified. However, accurately localizing the edit is essential in most software maintenance tasks and is not a trivial task. For example, in the codebase co-evolution task, functionally similar code snippets may not be readily available or easy to identify. Similarly, fault localization is a critical and challenging step in program repair before generating the patch. Future work can design LLMs that can first localize or suggest the next place to edit, and then generate the corresponding edit. This capability is important for advancing toward the goal of automating software maintenance using artificial intelligence.

Another important problem is how to suggest the next edit based on previous edits. Models can be built through being trained on the developers' editing activities. Within the same file, edits can be collected and paired as *relevant* edits. Cross-file relevant edits could be grouped using heuristics that identify related changes from version control commits made within a short time window.

5.3 Software Maintenance Agents

Software maintenance requires the collaboration between multiple experienced developers. Using artificial intelligence to fully automate this process, the collaboration of multiple software engineering AI agents just like humans is necessary and promising. To achieve this, AI agents need to have the reasoning capability to solve real-world software maintenance tasks in addition to having the basic knowledge of coding. Then, they should learn to collaborate and communicate like a team of software developers to accomplish the complicated task.

Learning from developer activities. Different development teams use different tools and follow different practices for maintaining software. To effectively mimic human behavior, an AI agent should be trained on the daily activities of software developers. One promising approach is to train models on commit histories from version control systems. Specifically, models should be able to predict future commits based on previous ones. This helps equip the models with the ability to infer developer's intent, enabling them to make decisions and take actions similar to human developers.

Multi-agent collaboration. A single action is often not sufficient to complete a software maintenance task. Therefore, a team of LLMs should collaborate to automate different subtasks of one complicated task. For example, in bug fixing, one agent may localize the buggy file, another may propose a patch, and a third may run regression tests. These agents should be able to communicate effectively and iterate together through the entire bug fixing process. Similarly, for code review, a code editing agent can apply changes to the codebase, while a reviewer agent provides feedback on each proposed modification. As the capabilities of foundation LLMs improve, task-specific agents will also become more effective especially after fine-tuned on the task-specific developer activity data.

Chapter 6: Conclusion

Software maintenance is essential for keeping software systems functional, secure, and up-to-date, but it often requires significant human effort. Existing generative models perform well on generation tasks, but fall short on editing tasks, which are common in software maintenance. This dissertation proposes designing software evolution-aware LLMs to better assist developers with software maintenance tasks.

First, we presented a novel edit-driven pretraining objective and used it to develop CoditT5, a pretrained language model for software maintenance tasks. CoditT5 is pretrained on large amounts of source code and natural language comments to perform edits, and we evaluated this model by fine-tuning it on three distinct downstream tasks: comment updating, bug fixing, and automated code review. By outperforming task-specific baselines and generation baselines across tasks, we demonstrate the suitability of CoditT5 (and our pretraining objective) for software maintenance tasks and its generalizability.

Second, we presented a new task: translating code changes across programming languages with the goal to synchronize projects that provide the same APIs or implementations in multiple programming languages. We proposed Codeditor, a model which uses code change history as contextual information and learns to make edits on the existing version of code written in the target programming language. We showed that our model outperforms existing code translation models and is better than the generation-based models even if they use historical context.

We envision that with the rapid development of LLMs, software maintenance tasks will be partially automated through the use of LLMs and AI agents. We argue that designing software evolution-aware LLMs and training them on software evolution data, like demonstrated in this document, is key to achieving human-level intelligence in maintaining software systems.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. https://doi.org/10.48550/arXiv.2303.08774.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, 2021. https://doi.org/10.18653/v1/2021.naacl-main.211.

[3] Toufique Ahmed and Premkumar Devanbu. Few-shot training LLMs for project-specific code-summarization. In *Automated Software Engineering*, pages 1–5, 2022. https://doi.org/10.1145/3551349.3559555.

[4] Christian Mauceri Alexandre FAU. Java2csharp, 2013. http://sourceforge.net/projects/j2cstranslator/.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. https://doi.org/10.48550/arXiv.2108.07732.

[6] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015. https://arxiv.org/abs/1409.0473.

[7] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *ACL Workshop*

on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, pages 65–72, 2005. https://aclanthology.org/W05-0909/.

[8] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020. https://doi.org/10.48550/arXiv.2004.05150.

[9] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. An empirical investigation of statistical significance in NLP. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005, 2012. https://www.aclweb.org/anthology/D12-1091/.

[10] Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew Gormley. Unlimiformer: Long-range transformers with unlimited length input. In *Advances in Neural Information Processing Systems*, pages 35522–35543, 2023. https://proceedings.neurips.cc/paper_files/paper/2023/file/6f9806a5adc72b5b834b27e4c7c0df9b-Paper-Conference.pdf.

[11] Shaked Brody, Uri Alon, and Eran Yahav. A structural model for contextual code changes. *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–28, 2020. https://doi.org/10.1145/3428283.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, pages 1877–1901, 2020. https://papers.nips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[13] Nghi DQ Bui and Lingxiao Jiang. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *International*

Conference on Software Engineering, New Ideas and Emerging Results, pages 33–36, 2018. `https://doi.org/10.1145/3183399.3183427`.

[14] Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. In *Automated Software Engineering*, pages 443–455, 2021. `https://doi.org/10.1109/ASE51524.2021.9678559`.

[15] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *Transactions on Software Engineering*, pages 1385–1399, 2020. `https://doi.org/10.1109/TSE.2020.3020502`.

[16] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022. `https://doi.org/10.48550/arXiv.2207.10397`.

[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. `https://doi.org/10.48550/arXiv.2107.03374`.

[18] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, volume 31, 2018. `https://proceedings.neurips.cc/paper_files/paper/2018/file/d759175de8ea5b1d9a2660e45554894f-Paper.pdf`.

[19] Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. PLUR: A unifying, graph-based view of program learning, understanding, and repair. In *Advances in Neural Information Processing Systems*, pages 23089–23101, 2021. `https://proceedings.neurips.cc/paper_files/paper/2021/file/c2937f3a1b3a177d2408574da0245a19-Paper.pdf`.

[20] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. PyMT5: multi-mode translation of natural language and python code with transformers. In *Empirical Methods in Natural Language Processing*, pages 9052–9065, 2020. `https://doi.org/10.18653/v1/2020.emnlp-main.728`.

[21] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022. `https://papers.nips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf`.

[22] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. Patching as translation: the data and the metaphor. In *Automated Software Engineering*, pages 275–286, 2020. `https://doi.org/10.1145/3324884.3416587`.

[23] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. Generating bug-fixes using pretrained transformers. In *International Symposium on Machine Programming*, pages 1–8, 2021. `https://doi.org/10.1145/3460945.3464951`.

[24] Elliot Chance et al. A tool for transpiling c to go, 2021. `https://github.com/elliotchance/c2go`.

[25] Angela Fan, Mike Lewis, and Yann Dauphin. Strategies for structuring story generation. In *Annual Meeting of the Association for Computational Linguistics*, pages 2650–2660, 2019. `https://doi.org/10.18653/v1/P19-1254`.

[26] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. In *Empirical Methods in Natural Language Processing: Findings*, pages 1536–1547, 2020. `https://doi.org/10.18653/v1/2020.findings-emnlp.139`.

[27] Christopher Foster, Abhishek Gulati, Mark Harman, Inna Harper, Ke Mao, Jillian Ritchey, Hervé Robert, and Shubho Sengupta. Mutation-guided LLM-based test generation at meta. *arXiv preprint arXiv:2501.12862*, 2025. https://doi.org/10.48550/arXiv.2501.12862.

[28] Python Software Foundation. difflib — helpers for computing deltas, 2023. https://docs.python.org/3/library/difflib.html.

[29] Galois and Immunant. C2rust, 2023. https://github.com/immunant/c2rust.

[30] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. Automating the removal of obsolete TODO comments. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 218–229, 2021. https://doi.org/10.1145/3468264.3468553.

[31] Paul Gauthier. Aider is ai pair programming in your terminal, 2024. https://aider.chat/.

[32] Google. Google cloud, 2023. https://cloud.google.com/.

[33] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Annual Meeting of the Association for Computational Linguistics*, pages 1631–1640, 2016. https://doi.org/10.18653/v1/P16-1154.

[34] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Liu Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2020. https://openreview.net/pdf?id=jLoC4ez43PZ.

[35] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Annual Meeting of the Association for Computational Linguistics*, pages 7212–7225, 2022. https://doi.org/10.18653/v1/2022.acl-long.499.

[36] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. https://doi.org/10.48550/arXiv.2501.12948.

[37] Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics*, 6:437–450, 2018. https://doi.org/10.1162/tacl_a_00030.

[38] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *International Symposium on the Foundations of Software Engineering*, pages 543–553, 2013. https://doi.org/10.1145/2491411.2491461.

[39] Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. A retrieve-and-edit framework for predicting structured outputs. In *Advances in Neural Information Processing Systems*, pages 10073–10083, 2018. https://proceedings.neurips.cc/paper_files/paper/2018/file/cd17d3ce3b64f227987cd92cd701cc58-Paper.pdf.

[40] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, pages 837–847, 2012. https://dl.acm.org/doi/10.5555/2337223.2337322.

[41] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024. `https://doi.org/10.48550/arXiv.2409.12186`.

[42] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. `https://doi.org/10.48550/arXiv.1909.09436`.

[43] MongoDB Inc. Mongodb, 2023. `https://www.mongodb.com/`.

[44] itext Software. itext-dotnet, 2025. `https://github.com/itext/itext-dotnet`.

[45] itext Software. itext-java, 2025. `https://github.com/itext/itext-java`.

[46] Robert Iv, Alexandre Passos, Sameer Singh, and Ming-Wei Chang. FRUIT: Faithfully reflecting updated information in text. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3670–3686, 2022. `https://doi.org/10.18653/v1/2022.naacl-main.269`.

[47] Paul Jaccard. The distribution of the flora in the alpine zone. *New phytologist*, pages 37–50, 1912. `https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x`.

[48] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with LLMs. In *International Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023. `https://doi.org/10.1145/3611643.3613892`.

[49] Hisashi Kamezawa, Noriki Nishida, Nobuyuki Shimizu, Takashi Miyazaki, and Hideki Nakayama. RNSum: A large-scale dataset for automatic release note generation via commit logs summarization. In *Annual Meeting of the Association for Computational Linguistics*, pages 8718–8735, 2022. `https://doi.org/10.18653/v1/2022.acl-long.597`.

[50] Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using GPT-3. In *Automated Software Engineering*, pages 1–6, 2022. `https://doi.org/10.1145/3551349.3559548`.

[51] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, pages 20601–20611, 2020. `https://proceedings.neurips.cc/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf`.

[52] Jaeseong Lee, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. On the naturalness of hardware descriptions. In *Symposium on the Foundations of Software Engineering*, pages 530–542, 2020. `https://doi.org/10.1145/3368089.3409692`.

[53] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020. `https://doi.org/10.18653/v1/2020.acl-main.703`.

[54] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. EditSum: A retrieve-and-edit framework for source code summarization. In *Automated Software Engineering*, pages 155–166, 2021. `https://doi.org/10.1109/ASE51524.2021.9678724`.

[55] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. In *International Symposium on the Foundations of Software Engineering*, 2022. `https://doi.org/10.1145/3540250.3549081`.

[56] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. Automated comment update: How far are we? In *International Conference on Program Comprehension*, pages 36–46, 2021. `https://doi.org/10.1109/ICPC52881.2021.00013`.

[57] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. Predictive comment updating with heuristics and AST-path-based neural learning: A two-phase approach. *Transactions on Software Engineering*, 49(4):1640–1660, 2022. `https://doi.org/10.1109/TSE.2022.3185458`.

[58] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. `https://doi.org/10.48550/arXiv.1907.11692`.

[59] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. More precise regression test selection via reasoning about semantics-modifying changes. In *International Symposium on Software Testing and Analysis*, pages 664–676, 2023. `https://doi.org/10.1145/3597926.3598086`.

[60] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. Just-In-Time obsolete comment detection and update. *Transactions on Software Engineering*, 2021. `https://doi.org/10.1109/TSE.2021.3138909`.

[61] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al.

CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021. https://doi.org/10.48550/arXiv.2102.04664.

[62] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering code large language models with evol-instruct. In *International Conference on Learning Representations*, 2024. https://openreview.net/pdf?id=UnUwSIgK5W.

[63] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Isil Dillig. Automated transpilation of imperative to functional code using neural-guided program synthesis. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–27, 2022. https://doi.org/10.1145/3527315.

[64] Lara Martin, Prithviraj Ammanabrolu, Xinyu Wang, William Hancock, Shruti Singh, Brent Harrison, and Mark Riedl. Event representations for automated story generation with deep neural nets. In *AAAI Conference on Artificial Intelligence*, pages 868–875, 2018. https://dl.acm.org/doi/pdf/10.5555/3504035.3504141.

[65] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *International Conference on Software Engineering*, pages 336–347, 2021. https://doi.org/10.1109/ICSE43902.2021.00041.

[66] Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. An empirical study on LLM-based agents for automated bug fixing. *arXiv preprint arXiv:2411.10213*, 2024. https://doi.org/10.48550/arXiv.2411.10213.

[67] Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. OctoPack: Instruction tuning code large language models. In *International Conference on Learning Representations*, 2024. https://openreview.net/pdf?id=mw1PWNSWZP.

[68] Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. Ground truth for grammatical error correction metrics. In *Annual Meeting of the Association for Computational Linguistics*, pages 588–593, 2015. https://doi.org/10.3115/v1/P15-2097.

[69] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Automated Software Engineering*, pages 585–596, 2015. https://doi.org/10.1109/ASE.2015.74.

[70] Pengyu Nie. *Machine Learning for Executable Code in Software Testing and Verification*. PhD thesis, The University of Texas at Austin, 2023. https://hdl.handle.net/2152/123170.

[71] Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Impact of evaluation methodologies on code summarization. In *Annual Meeting of the Association for Computational Linguistics*, pages 4936–4960, 2022. https://doi.org/10.18653/v1/2022.acl-long.339.

[72] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *International Conference on Software Engineering*, 2023. https://doi.org/10.1109/ICSE48619.2023.00178.

[73] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for

code with multi-turn program synthesis. In *International Conference on Learning Representations*, 2023. https://openreview.net/pdf?id=iaYcJKpY2B_.

[74] OpenAI. Introducing ChatGPT, 2023. https://openai.com/blog/chatgpt.

[75] OpenAI. Introducing ChatGPT-4, 2024. https://platform.openai.com/docs/models.

[76] Jialing Pan, Adrien Sadé, Jin Kim, Eric Soriano, Guillem Sole, and Sylvain Flamant. SteloCoder: a decoder-only LLM for multi-language to python code translation. *arXiv preprint arXiv:2310.15539*, 2023. https://doi.org/10.48550/arXiv.2310.15539.

[77] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. Learning to update natural language comments based on code changes. In *Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868, 2020. https://doi.org/10.18653/v1/2020.acl-main.168.

[78] Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. Copy that! editing sequences by copying spans. In *AAAI Conference on Artificial Intelligence*, pages 13622–13630, 2021. https://cdn.aaai.org/ojs/17606/17606-13-21100-1-2-20210518.pdf.

[79] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. In *AAAI Conference on Artificial Intelligence*, pages 427–435, 2021. https://cdn.aaai.org/ojs/16119/16119-13-19613-1-2-20210518.pdf.

[80] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Annual Meeting of the Association for Computational Linguistics*, pages 311–318, 2002. https://doi.org/10.3115/1073083.1073135.

[81] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. `https://www.antlr2.org/article/1055550346383/antlr.pdf`.

[82] Karl Pichotta and Raymond Mooney. Learning statistical scripts with LSTM recurrent neural networks. In *AAAI Conference on Artificial Intelligence*, pages 2800–2806, 2016. `https://doi.org/10.1609/aaai.v30i1.10347`.

[83] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1:9, 2019. `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`.

[84] Cosmin Radoi, Stephen J Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to MapReduce. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 909–927, 2014. `https://doi.org/10.1145/2714064.2660228`.

[85] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020. `https://dl.acm.org/doi/pdf/10.5555/3455716.3455856`.

[86] Musfiqur Rahman, Dharani Palani, and Peter C. Rigby. Natural software revisited. In *International Conference on Software Engineering*, pages 37–48, 2019. `https://doi.org/10.1109/ICSE.2019.00022`.

[87] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. AI-powered code review with LLMs: Early results. *arXiv preprint arXiv:2404.18496*, 2024. `https://doi.org/10.48550/arXiv.2404.18496`.

[88] Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3:57–87, 1997. https://doi.org/10.1017/S1351324997001502.

[89] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020. https://doi.org/10.48550/arXiv.2009.10297.

[90] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021. https://doi.org/10.48550/arXiv.2110.06773.

[91] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. https://doi.org/10.48550/arXiv.2308.12950.

[92] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *Empirical Methods in Natural Language Processing*, pages 379–389, 2015. https://doi.org/10.18653/v1/D15-1044.

[93] Apache Software. Apache Lucene, 2022. https://lucene.apache.org/.

[94] Felix Stahlberg and Shankar Kumar. Seq2edits: Sequence transduction using span-level edit operations. In *Empirical Methods in Natural Language Processing*, pages 5147–5159, 2020. https://doi.org/10.18653/v1/2020.emnlp-main.418.

[95] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors

with graph2diff neural networks. In *International Conference on Software Engineering Workshops*, pages 19–20, 2020. https://doi.org/10.1145/3387940.3392181.

[96] Aider Team. Edit formats, 2023. https://aider.chat/docs/more/edit-formats.html.

[97] Cursor Team. The ai code editor, 2024. https://cursor.com/.

[98] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023. https://doi.org/10.48550/arXiv.2312.11805.

[99] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. Structcoder: Structure-aware transformer for code generation. *ACM Trans. Knowl. Discov. Data*, 18 (3), 2024. https://doi.org/10.1145/3636430.

[100] Marco Trudel, Manuel Oriol, Carlo A Furia, and Martin Nordio. Automated translation of Java source code to Eiffel. In *International Conference on Objects, Models, Components, Patterns*, pages 20–35, 2011. https://dl.acm.org/doi/10.5555/2025896.2025900.

[101] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *International Conference on Software Engineering*, pages 25–36, 2019. https://doi.org/10.1109/ICSE.2019.00021.

[102] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *Transactions on Software Engineering*, 28:1–29, 2019. https://doi.org/10.1145/3340544.

[103] Rosalia Tufano, Luca Pascarella, Michele Tufanoy, Denys Poshyvanykz, and Gabriele Bavota. Towards automating code review activities. In *International Conference on Software Engineering*, pages 163–174, 2021. https://doi.org/10.1109/ICSE43902.2021.00027.

[104] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *International Conference on Software Engineering*, pages 2291–2302, 2022. https://doi.org/10.1145/3510003.3510621.

[105] Rosalia Tufano, Alberto Martin-Lopez, Ahmad Tayeb, Sonia Haiduc, Gabriele Bavota, et al. Deep learning-based code reviews: A paradigm shift or a double-edged sword? In *International Conference on Software Engineering*, 2025. https://doi.org/10.1109/ICSE55347.2025.00060.

[106] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[107] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, 2015. https://proceedings.neurips.cc/paper_files/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf.

[108] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. SynCoBERT: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*, 2021. https://doi.org/10.48550/arXiv.2108.04556.

[109] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. `https://openreview.net/pdf?id=OJd3ayDDoF`.

[110] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021. `https://doi.org/10.18653/v1/2021.emnlp-main.685`.

[111] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. Codet5+: Open code large language models for code understanding and generation. In *Empirical Methods in Natural Language Processing*, pages 1069–1088, 2023. `https://doi.org/10.18653/v1/2023.emnlp-main.68`.

[112] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-Thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, pages 24824–24837, 2022. `https://papers.nips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf`.

[113] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. SWE-RL: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025. `https://doi.org/10.48550/arXiv.2502.18449`.

[114] Wei Xu, Courtney Napoles, Ellie Pavlick, Quanze Chen, and Chris Callison-Burch. Optimizing statistical machine translation for text simplification.

Transactions of the Association for Computational Linguistics, 4:401–415, 2016. https://doi.org/10.1162/tacl_a_00107.

[115] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, pages 50528–50652, 2024. https://papers.nips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

[116] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. Learning structural edits via incremental tree transformations. In *International Conference on Learning Representations*, 2021. https://openreview.net/pdf?id=v9hAX77--cZ.

[117] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Annual Meeting of the Association for Computational Linguistics*, pages 440–450, 2017. https://doi.org/10.18653/v1/P17-1041.

[118] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. Learning to represent edits. In *International Conference on Learning Representations*, 2018. https://openreview.net/pdf?id=BJl6AjC5F7.

[119] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. Rectifier: Code translation with corrector via LLMs. *arXiv preprint arXiv:2407.07472*, 2024. https://doi.org/10.48550/arXiv.2407.07472.

[120] Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. Transagent: An LLM-based multi-agent system for code translation. *arXiv preprint arXiv:2409.19894*, 2024. https://doi.org/10.48550/arXiv.2409.19894.

[121] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. Comparing and combining analysis-based and learning-based regression test selection. In *International Conference on Automation of Software Test*, pages 17–28, 2022. `https://doi.org/10.1145/3524481.3527230`.

[122] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. CoditT5: Pretraining for source code and natural language editing. In *International Conference on Automated Software Engineering*, 2022. `https://doi.org/10.1145/3551349.3556955`.

[123] Jiyang Zhang, Marko Ristin, Phillip Schanely, Hans Wernher Van De Venn, and Milos Gligoric. Python-by-contract dataset. In *International Symposium on the Foundations of Software Engineering*, pages 1652–1656, 2022. `https://doi.org/10.1145/3540250.3558917`.

[124] Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, and Arie van Deursen. Using large-scale heterogeneous graph representation learning for code review recommendations at Microsoft. *International Conference on Software Engineering*, 2023. `https://doi.org/10.1109/ICSE-SEIP58684.2023.00020`.

[125] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Multilingual code co-evolution using large language models. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023. `https://doi.org/10.1145/3611643.3616350`.

[126] Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. exLong: Generating exceptional behavior tests with large language models. In *International Conference on Software Engineering*, 2025. `https://doi.org/10.1109/ICSE55347.2025.00176`.

[127] Linghan Zhong, Samuel Yuan, Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. A tool for generating exceptional behavior tests with large language models. In *International Symposium on the Foundations of Software Engineering, Demo Track*, 2025. `https://doi.org/10.1145/3696630.3728608`.

[128] Qingyu Zhou, Nan Yang, Furu Wei, and Ming Zhou. Sequential copying networks. In *AAAI Conference on Artificial Intelligence*, pages 4987–4994, 2018. `https://cdn.aaai.org/ojs/11915/11915-13-15443-1-2-20201228.pdf`.

[129] Ming Zhu, Karthik Suresh, and Chandan K Reddy. Multilingual code snippets training for program translation. In *AAAI Conference on Artificial Intelligence*, pages 11783–11790, 2022. `https://doi.org/10.1609/aaai.v36i10.21434`.

[130] Yifan Zong, Yuntian Deng, and Pengyu Nie. Mix-of-language-experts architecture for multilingual programming. In *International Workshop on Large Language Models for Code (LLM4Code)*, pages 200–208, 2025. `https://doi.org/10.1109/LLM4Code66737.2025.00030`.