Automatically Recommend Code Updates: Are We There Yet?

YUE LIU, CHAKKRIT TANTITHAMTHAVORN, and YONGHUI LIU, Monash University, Australia

PATANAMON THONGTANUNAM, The University of Melbourne, Australia

LI LI, Beihang University, China

In recent years, large pre-trained Language Models of Code (CodeLMs) have shown promising results on various software engineering tasks. One such task is automatic code update recommendation, which transforms outdated code snippets into their approved and revised counterparts. Although many CodeLM-based approaches have been proposed, claiming high accuracy, their effectiveness and reliability on real-world code update tasks remain questionable. In this paper, we present the first extensive evaluation of state-of-the-art CodeLMs for automatically recommending code updates. We assess their performance on two diverse datasets of paired updated methods, considering factors such as temporal evolution, project specificity, method size, and update complexity. Our results reveal that while CodeLMs perform well in settings that ignore temporal information, they struggle in more realistic time-wise scenarios and generalize poorly to new projects. Furthermore, CodeLM performance decreases significantly for larger methods and more complex updates. Furthermore, we observe that many CodeLM-generated “updates” are actually null, especially in time-wise settings, and meaningful edits remain challenging. Our findings highlight the significant gap between the perceived and actual effectiveness of CodeLMs for real-world code update recommendation and emphasize the need for more research on improving their practicality, robustness, and generalizability.

CCS Concepts: • Software and its engineering → Software testing and debugging.

Additional Key Words and Phrases: Code Updates, Neural Machine Translation

ACM Reference Format:
Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, Patanamon Thongtanunam, and Li Li. 2024. Automatically Recommend Code Updates: Are We There Yet?. 1, 1 (May 2024), 27 pages. https://doi.org/10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, large pre-trained language models for code generation and understanding have gained significant popularity [36]. These models, known as Language Models of Code (CodeLMs), have demonstrated promising results across a wide range of software engineering tasks [16, 36]. A number of CodeLMs such as CodeBERT [17], CodeT5 [89], and CodeGPT [61] have been released for software engineering tasks, including code search [30], code generation [47], defect prediction & localization [14, 37], and code review [81–83]. While recent efforts have aimed at enhancing the performance of these models, previous studies [59, 60, 75] have indicated the lack of standardized and realistic benchmarks that can measure their generalization and robustness. Prior works [59, 60, 75]...
have shown that even popular datasets like CodeSearchNet [38] and CodeXGlue [61] suffer from various issues that lead to unreliable performance measures. For instance, Liu et al. [59] discovered severe data duplication problems in several program generation datasets from CodeXGlue [61], which resulted in over-optimistic performance. Moreover, most datasets lack sufficient metadata, such as the source and temporal information, leaving us in the dark about the reasons behind the success or failure of CodeLMs [75]. As a result, the effectiveness of these models in real-world software development and maintenance scenarios remains an open question.

To better understand and examine the real-world applicability of CodeLMs, we focus on a prevalent task in software maintenance: code updates. Code updates represent a constant and evolving challenge in software development, as developers must continually integrate dependencies, adapt to new APIs, migrate between frameworks, fix newly discovered bugs, and add features to meet user needs. Given the frequent and repetitive nature of code updates, CodeLMs could be a valuable tool for automating such tasks, thereby reducing the manual effort required from developers. However, as highlighted by previous studies [59, 60, 75], the reliability and robustness of CodeLMs on practical software engineering tasks remain questionable. The question, “Can CodeLMs effectively recommend code updates?” and the factors contributing to their success or failure in recommending code updates, remain unanswered. Therefore, in this paper, we aim to investigate the potential of CodeLMs for automating code updates. We identify the key challenges and opportunities in this direction with the goal of providing insights that can guide future research towards making automated code updates a reality.

In this work, we present the first extensive evaluation of recent CodeLMs in automatically recommending code updates. To ensure a comprehensive and rigorous assessment, we utilize two unique datasets of paired updated methods. The first dataset is extracted from 3,195 Android apps published on Google Play and hosted on GitHub, spanning the period from 2008 to 2022. The second dataset is an existing benchmark dataset from Pornprasit et al. [67], which includes code updates from three diverse software projects: Android, Google, and Ovirt. These datasets provide a rich and diverse source of information, enabling us to rigorously assess the capabilities of CodeLMs in handling real-world code updates. To determine the effectiveness of CodeLMs in recommending code updates, we train and evaluate various CodeLMs (i.e., CodeT5 [89], CodeT5+[88], CodeBERT[17], CodeGPT [61], UniXcoder [31]) and two baselines (i.e., T5 [70], AutoTransform [81]). Our evaluation methodology is carefully designed to consider various factors that may impact the performance of CodeLMs. We take into account the temporal nature of code updates by evaluating the models in both time-wise and time-ignore scenarios. This allows us to assess the models’ ability to capture the evolutionary aspects of code changes. Furthermore, we analyze the performance of CodeLMs across different code update types, method sizes, and update complexities. We aim to uncover the strengths (good), limitations (bad), and areas for improvement (missing) of CodeLMs in handling diverse code update scenarios. We address the following five research questions:

(RQ1) How accurate are CodeLMs at recommending code updates?

Results. CodeLMs perform well in time-ignore scenarios, achieving up to 36.9% accuracy, and increasing the beam search size can improve their performance. However, they perform poorly in realistic time-wise evaluations and cross-project settings, with accuracy decreases ranging from 58.7% to 100%.

(RQ2) What is the syntactical correctness of the code generated by CodeLMs?

Results. State-of-the-art CodeLMs achieve high syntactical correctness rates, often above 90%, in most datasets and scenarios. However, even with high syntactical correctness rates, CodeLMs still generate syntactically incorrect code in some instances, such as mismatching parentheses.
(RQ3) Why do CodeLMs perform differently in time-wise and time-ignore scenarios

**Results.** Time-ignore data has substantial overlap (17-52% of test samples seen in training), while time-wise has much less (2-28%). CodeLMs do well on seen samples but struggle with novel and complex ones, especially in time-wise settings.

(RQ4) How do the method size and update size impact the accuracy of CodeLMs?

**Results.** CodeLMs perform better on smaller methods and updates with fewer changed tokens. However, their accuracy decreases significantly as method size and update complexity increase, dropping to 0% for larger methods and more complex updates.

(RQ5) What types of code updates are learned and predicted by CodeLMs?

**Results.** Many predicted "updates" are actually null (no changes made), especially in time-wise settings (up to 92%). When changes are made, they are often simple, frequent updates. Meaningful edits remain challenging.

In summary, while CodeLMs show potential for automating code updates, our results highlight significant challenges in adapting to new projects, generating valid code, and recommending meaningful changes, particularly in realistic time-wise settings. We hope our findings will motivate future work on improving the practicality of CodeLM-based code update tools. The key contributions of this paper are:

- Conduct an extensive evaluation of state-of-the-art CodeLMs for the task of automatically recommending code updates;
- An in-depth comparative analysis of CodeLM performance across different scenarios and granularities;
- Insights into the current challenges facing CodeLMs in this domain, with recommendations for future research directions.

**Open Science.** To support the open science initiative, we publish our dataset and a replication package, which is publicly available in GitHub.¹

**Paper Organization.** Section 2 presents the background and motivation. Section 3 describes the study design of this empirical study. Section 4 presents our studied datasets and the experimental setup, while Section 5 presents our research questions and the experimental results. Section 6 provides a critical discussion of our findings and their limitations. Section 7 discloses the threats to validity. Section 8 describes the related work. Section 9 concludes the paper.

# 2 BACKGROUND

This section provides an introduction to the concept of automated code recommendation and outlines the motivation for this study.

## 2.1 Language Models

Language models (LMs) play a crucial role in natural language processing (NLP) and artificial intelligence (AI) [55]. By modeling the probability distribution of sequences of words on a training corpus, LMs can be effectively applied to tasks such as text generation [46] and understanding [86]. LMs generally consist of two primary components: an encoder that processes the input sequence and generates a numerical, intermediate representation, and a decoder that utilizes this intermediate representation to produce the target sequence one token at a time. The development of language models has witnessed several significant milestones, from early n-gram models to more sophisticated techniques. Recurrent neural networks (RNNs) have gained prominence as an influential approach capable of capturing longer dependencies and contextual information [50].

¹https://github.com/yueyueL/CodeLM-CodeUpdateEval
However, the Transformer architecture revolutionized the field of language modeling with its use of self-attention mechanisms for handling long-range dependencies and parallelism more effectively [70, 86], opening new doors for AI research.

The introduction of the Transformer architecture sparked a proliferation of large-scale, pre-trained LMs, such as BERT [46], T5 [70], and GPT-3 [8], which have demonstrated remarkable improvements across various NLP tasks. These models benefit from the utilization of extensive training data and transfer learning techniques. By pre-training on large generic datasets through self-supervised tasks, these models acquire a comprehensive understanding of language, which can be further fine-tuned on smaller, specialized datasets for specific tasks. For example, BERT employs a bidirectional transformer and is pre-trained on 3.3 billion words, yielding outstanding results in tasks such as sentiment analysis and question answering [62].

Recently, there has been a growing interest in domain-specific pre-trained models, such as CodeT5 [89] and CodeBERT [17], which are specifically designed for programming language (PL) processing tasks. These models are pre-trained LMs of Code (CodeLMs) on extensive programming code corpora and, akin to their general-purpose counterparts, can be fine-tuned on specialized code-related tasks. These pre-trained models can be categorized into three groups: encoder-only models, decoder-only models, and encoder-decoder models. **Encoder-based models**, like CodeBERT [17], exclusively use a bidirectional transformer encoder with an attention mechanism to learn vectorized embeddings of input code sequences. These models are best suited for non-generation downstream tasks, such as code representation and code clone detection [17]. **Decoder-based models**, including CodeGPT [61] and Codex [11], employ an autoregressive transformer decoder to generate code sequences. Unlike encoder-only models that focus on input code embeddings, decoder-only models excel in open-ended code generation tasks [11]. Finally, **encoder-decoder-based models** like CodeT5 [89] feature both a bidirectional transformer encoder and an autoregressive transformer decoder. The encoder computes input code embeddings, while the decoder generates code. This flexibility allows encoder-decoder models to accommodate both non-generation and generation downstream tasks [63, 74, 87]. These code-focused pre-trained models have demonstrated exceptional performance in tasks like code summarization [74], code search [87], and code completion [39]. The development of advanced pre-trained CodeLMs, encompassing both general-purpose and domain-specific varieties, has significantly expanded the horizons of NLP and AI research, extending its applicability to the realm of software engineering.

### 2.2 Automated Code Update

Code updates are an important quality assurance practice in the software development process, aimed at ensuring that newly developed code meets the required standards before integration into the main software repository [81, 83, 99]. The software update process involves revising the source code to rectify defects and bugs [99], enhance code quality and readability [52], and support new functions [93]. However, during the software development lifecycle, developers are required to frequently update the source code, which is time-consuming and human-intensive. Given the large number of reviews (e.g., 3,000 reviews per month at Microsoft Bing [72]), prior studies have found that it is challenging for code authors to revise their code without introducing new defects while switching contexts and keeping track of other reviews [13, 49]. Automated code update recommendation could potentially save developers’ effort by automatically applying common revisions from past code reviews to newly-written code, allowing code authors to focus on revising more complex code. This approach would not only reduce the manual effort required from developers but also help maintain code quality and consistency throughout the development process.

By leveraging the knowledge gained from previous code updates, automated code recommendation
systems can provide valuable suggestions and assist developers in efficiently updating their code to meet the required standards.

Recent work has explored the use of language models (LMs) to support automated code update recommendation. For example, Tufano et al. [83] proposed an RNN Encoder-Decoder architecture to learn how to automatically apply Java code changes implemented by developers during pull requests. Their approach involves three main steps: code abstraction, building an LM model, and generating predictions. The model learns the relationship between the “before” and “after” code sequences and generates updated code based on this learned mapping. AutoTransform [81] further improved upon this approach by using a Transformer Encoder-Decoder-based architecture and Byte-Pair Encoding (BPE) subword tokenization to better handle new tokens and long code sequences. These LM-based approaches have shown promising results in automating code updates and have the potential to significantly reduce the manual effort required from developers.

### 2.3 Motivation

Despite the advancements in automated code update recommendation using LMs, existing approaches still exhibit certain limitations that warrant further exploration. For instance, Tufano et al.’s approach [83] struggles with handling new tokens in the updated code version and efficiently processing long code sequences. Although AutoTransform [81] improves upon Tufano et al.’s work by utilizing a Transformer-based architecture and BPE subword tokenization, its overall accuracy remains relatively low, with only 20.4% accuracy when using a beam search size of 10. In real-world scenarios, it is impractical and time-consuming for developers to select the correct recommendation from 10 candidates, making this level of accuracy insufficient for practical use. Secondly, previous studies have not thoroughly assessed the robustness of their proposed approaches concerning real-world constraints. For example, temporal factors and the relation between training and testing splits over time have not been adequately addressed in real-world settings. Liu et al. [58] demonstrate that ignoring time information when splitting train and test data can lead to over-optimistic performance of malware detection classifiers. Additionally, in real-world scenarios, cross-project and within-project prediction settings should be considered [45, 73]. However, these settings have not been discussed in previous studies [81, 83]. Therefore, comprehensive studies are required to understand the capabilities and limitations of advanced state-of-the-art language models for code updates in real-world settings.

Moreover, the reliability and trustworthiness of performance measures of CodeLMs remain questionable [73], making it challenging to discern why CodeLMs fail or succeed. For instance, dataset construction often lacks sufficient transparency and metadata to reliably attribute the failure or success of code update recommendations to specific factors [73, 75]. Liu et al. [59] demonstrated that the dataset used in the Tufano et al. and AutoTransform studies is not reliable, providing overestimated performance measures. Specifically, data duplications exist between the training and testing sets, and overlapping input samples in testing sets inflate performance claims. Without transparent and unbiased evaluations, the real-world effectiveness of CodeLMs on practical tasks like code updates remains uncertain.

To address these limitations and investigate the potential of CodeLMs for automated code updates, we conduct an extensive empirical study examining state-of-the-art CodeLMs on real-world settings. By curating datasets from the real world and evaluating various CodeLMs under different scenarios, considering factors like update types, temporal robustness, project specificity, and method lengths, we aim to provide the first systematic assessment of applying modern CodeLMs to code update recommendations. Our work aims to uncover the capabilities and limitations of existing models on this practical task, identifying the good, bad, and missing aspects of their performance. Furthermore,
Table 1. An overview statistics of the studied datasets

| Dataset          | # Train | # Validation | # Test  | # Total |
|------------------|---------|--------------|---------|---------|
| Android          | 14,690  | 1,836        | 1,835   | 18,361  |
| Google           | 9,899   | 1,237        | 1,235   | 12,371  |
| Ovirt            | 21,506  | 2,686        | 2,688   | 26,880  |
| AndroZooUpdate-S | 19,638  | 4,910        | 6,227   | 30,194  |
| AndroZooUpdate-L | 136,932 | 34,234       | 43,714  | 209,346 |

our study aims to provide insights into the factors contributing to the success or failure of CodeLMs in recommending code updates.

3 STUDY DESIGN

3.1 Data Preparation

To effectively evaluate the performance of CodeLMs in recommending code updates, we have curated two datasets: AndroZooUpdate and one existing benchmark dataset from Pornprasit et al. [67]. Both these datasets include commit dates of their corresponding patches, allowing for a time-wise evaluation. The combination of these two datasets allows us to comprehensively assess the performance of CodeLMs across diverse software projects and update types. Table 1 presents the overview of the studied datasets.

3.1.1 Pornprasit et al. [67]. Building upon Tufano et al. [83], which extracted method-level code change transformations mined from code review repositories Gerrit [2] of software projects such as Google, Ovirt, and Android, Pornprasit et al. [67] further retrieved the code update patches with temporal information. This collection forms a code-to-code corpus, where the source code prior to the pull request is transformed into the target code, reflecting the changes post-review. This code update transformation dataset has been widely used by prior work [59, 67, 83, 97].

3.1.2 AndroZooUpdate. In addition to the existing benchmark dataset, we have also collected our own dataset, AndroZooUpdate, to understand the performance of CodeLMs in real-world scenarios. This dataset focuses specifically on code updates in Android applications, which are among the most popular mobile software systems worldwide [57]. By curating this dataset, we aim to assess the effectiveness of CodeLMs in a practical context, providing a valuable resource for researchers and practitioners working on automated code update recommendation systems. We have undertaken the following steps to collect and prepare the dataset for code update recommendations:

To collect code updates, we start with a collection of Android apps from the AndroZooOpen dataset [54], which comprises 46,521 open-source Android apps released between 2008 and 2020. AndroZooOpen provides metadata for app repositories, as well as links to the Google Play pages if the apps are published on Google Play. In this study, we focus exclusively on open-source Android apps published on Google Play to exclude toys, experimental repositories, or low-quality apps. As a result, we procured 3,195 open-source Android apps from AndroZooOpen. Subsequently, utilizing the provided metadata, we cloned the Git repositories of these apps and identified 1.3 million human-written commits for further extraction.

To extract all code modifications in the 1.3M commits that we collected, we utilize the git diff command in Git to identify which parts of code are updated in the two versions (i.e., before and after the code update). In this study, we identify code updates at the method level for two reasons: (1) a method implements a single functionality, providing sufficient context, and (2) file-level code updates may encompass multiple method-level code transformations, with files containing large
Automatically Recommend Code Updates: Are We There Yet?

Fig. 1. An example of collected data triplet.

amounts of text, including significant unchanged code sections that could hinder CodeLM model performance [83]. To more accurately evaluate code updates, we removed developers’ comments within code blocks. We then constructed a list of triplets (i.e., the method block before updates (prior), the method block after updates (updated), and the corresponding commit messages) to represent the updated methods (i.e., as shown in Figure 1). Ultimately, we extracted 209,346 pairs of updated methods and their corresponding commit messages (i.e., AndroZooUpdate-L). The dataset is represented as a list of triplets, i.e., \{(m_1, m'_1, c_1), \ldots, (m_N, m'_N, c_N)\}, where \(m_i\) denotes the version of a method prior to the code update, \(m'_i\) signifies the updated version, and \(c_i\) indicates the corresponding commit message.

3.2 Pre-trained LMs for Code Recommendation

Pre-trained CodeLMs are exposed to an extensive variety of programming languages and code samples covering numerous software engineering concepts and constructs during the pre-training phase. This enables the model to learn a general understanding of code syntax, semantics, and common programming patterns. To gain deeper insight into the impacts of pre-trained CodeLMs on automated code update recommendations, we examined four popular pre-trained CodeLMs for code generation and understanding, namely, CodeBERT [17] (an encoder-based model), CodeGPT [61] (a decoder-based model), CodeT5 [89] (an encoder-decoder-based model), CodeT5+ [88], and UniXcoder [31] (a unified model). Additionally, we include T5 [70], a model without pre-training in programming languages but designed for natural language processing, to assess the importance of domain-specific pre-training in the context of automated code recommendation. Tufano et al. [85] proposed a T5-based pre-trained code model for automated code update activities (called TufanoT5).

**CodeBERT.** CodeBERT is one of the first attempts to pre-train a Transformer-based encoder model with the primary objective of enhancing source code representation learning and comprehension. Based on the popular BERT architecture [46], CodeBERT is an encoder-only model that has been pre-trained on the CodeSearchNet dataset (an open-sourced code corpus containing 4291 projects collected from GitHub) [38]. This dataset consists of 2.1 million bimodal NL-PL (comment-function) pairs and 6.4 million unimodal functions without comments, spanning across six programming languages.

**CodeGPT.** CodeGPT [61] is an innovative pre-trained neural language model that focuses on code generation tasks. Built upon the GPT architecture, CodeGPT leverages the power of the autoregressive Transformer-based decoder to effectively generate code sequences. The model undergoes pre-training on Python and Java corpora derived from the CodeSearchNet dataset, encompassing 1.1 million Python functions and 1.6 million Java methods. This extensive pre-training process empowers CodeGPT to excel in various code generation tasks, demonstrating its potential in programming language processing and software engineering.

**CodeT5.** Built upon the T5 architecture [70], CodeT5 employs an encoder-decoder framework, utilizing bidirectional Transformer encoders and autoregressive Transformer decoders. This combination allows the model to not only calculate the embedding of input code but also generate code sequences effectively. The pre-training dataset for CodeT5 is derived from the CodeSearchNet corpus, as well as C and CSharp programs collected via BigQuery [1], which comprises approximately 5 million natural language instances and 8 million programming language instances.
CodeT5+. CodeT5+ [88] is an advanced version of the CodeT5 model, designed to further enhance the capabilities of automated code generation and understanding. It extends the CodeT5 model by introducing a mixture of pretraining objectives that cover span denoising, contrastive learning, text-code matching, and causal language modeling, on both unimodal and bimodal multilingual code corpora. Moreover, it leverages frozen off-the-shelf LLMs to initialize its encoder and decoder modules without training from scratch, and explores instruction-tuning to align the model with natural language instructions.

UniXcoder. UniXcoder [31] is a unified, cross-modal pre-trained model for programming languages, designed to support both code-related understanding and generation tasks. This model effectively combines the strengths of encoder and decoder architectures, enabling it to efficiently learn code representation while performing code generation tasks. To control the model’s behavior, UniXcoder employs mask attention matrices with prefix adapters and harnesses cross-modal content, such as Abstract Syntax Trees (ASTs) and code comments, to enhance code representation. The model is pre-trained using a combination of several tasks: masked language modeling (MLM), unidirectional language modeling, denoising autoencoder, and two contrastive learning-related tasks. This comprehensive pre-training approach allows UniXcoder to excel in a variety of programming language tasks, making it a versatile and powerful tool for code-related understanding and generation applications.

3.3 Fine-turning CodeLMs

Fine-tuning is a critical step in adapting pre-trained neural language models to specific tasks or domains, such as automated code recommendation. After pre-training on large-scale, diverse datasets, CodeLMs possess a generalized understanding of language structure and patterns. However, to excel in targeted tasks, these models often require additional training on smaller, specialized datasets relevant to the task. In our study, we fine-tune pre-trained CodeLMs to learn automated code recommendations by transforming initial methods into updated versions. We use $X_i = [x_1, ..., x_n]$ to denote the initial version and $Y_i = [y_1, ..., y_m]$ to represent the updated version. In line with previous studies [53, 81], we fine-tune the models on the related training dataset by minimizing the cross-entropy losses:

$$L(\theta) = \sum_{i=1}^{N} \log P(Y_i|X_i; \theta),$$

where $\theta$ represents the model parameters, and $N$ is the number of training examples. During the fine-tuning process, we update the model’s parameters by training it for a few epochs on the task-specific dataset. This enables the model to adapt to the unique characteristics and nuances of the data, resulting in improved performance on automated code update recommendation tasks.

3.4 Recommending Code Updates

Once the CodeLMs have been fine-tuned for the automated code recommendation dataset, the next step is to generate code updates based on the learned representations. Given an input code snippet $X_i$, the fine-tuned CodeLM model generates a probability distribution over the possible updates. To select the most appropriate updates, the fine-tuned model employs beam search, which considers multiple code update candidates for an input sequence at each timestep based on conditional probabilities. Once the beam search is complete, the top-ranked update candidates are selected as the final recommendations. These recommendations are ranked according to their probabilities, allowing developers to choose the most suitable update for their specific needs.
4 EXPERIMENTAL SETUP

In this section, we describe the experimental setup for our study. Our experimental setup encompasses five key aspects: studied datasets, data splitting, evaluation measures, and implementation & hyperparameter settings.

**Studied Datasets.** As mentioned in Section 3, we evaluate the performance of automatically recommending code updates on two datasets: AndroZooUpdate and Pornprasit et al. [67]. The AndroZooUpdate dataset consists of two versions: AndroZooUpdate-L and AndroZooUpdate-S. AndroZooUpdate-L contains 209,346 pairs of updated methods extracted from 3,195 open-source Android apps, with each pair including the source code of the changed methods. However, prior work [67, 84] has noted that large methods have a long tail distribution of sequence lengths with high variance, which may present difficulties when training an NMT model. To address this issue, we created AndroZooUpdate-S, a smaller version of the dataset that excludes updated methods with sequences longer than 50 tokens, resulting in 30,194 pairs of smaller updated methods. For the Pornprasit et al. [67] dataset, we use the original subsets provided in the paper, which include code updates from three software projects: Android, Google, and Ovirt. Table 1 provided an overview of the statistics for both datasets, offering a comprehensive summary of the data used in our study to evaluate the performance of CodeLMs in automatically recommending code updates.

**Data Splitting.** To mimic realistic recommendation scenarios, we prepare two versions of each dataset: time-wise and time-ignore. These versions differ in how the data is split into training, validation, and testing sets. In the time-wise evaluation scenario, the dataset is first sorted in chronological order based on the commit timestamps. This ensures that method pairs occurring earlier in time are only present in the training dataset and not in the testing dataset. Likewise, method pairs occurring later in time are only present in the testing dataset and not in the training dataset. This approach simulates a real-world scenario where the model is trained on historical data and evaluated on future and unseen data. On the other hand, the time-ignore evaluation scenario does not consider the temporal information of the method pairs, which is consistent with the approach used in prior studies [81, 84, 85]. In this scenario, the dataset is randomly split into training, validation, and testing sets without considering the temporal order of the method pairs. For both time-wise and time-ignore evaluations, we split the data into train/validation/test sets with a proportion of 80%/10%/10%, respectively.

**Evaluation Measures.** To evaluate the accuracy, we count the number of updated methods that achieve a **perfect prediction** (PP), i.e., the generated updated method exactly matches the ground-truth (i.e., the actual updated method). In this work, we use beam search to obtain best-$k$ candidates (i.e., $k = 1, 5, 10, 15$). If one of the $k$ candidates exactly matches the ground-truth, we consider the prediction is perfect. In this work, we also use the **BLEU** and **CodeBLEU** [71] score to measure the similarity between the generated predictions and the ground-truth. BLEU is a popular metric for evaluating the quality of the generated text, initially developed for machine translation tasks [64]. The BLEU score measures the similarity between generated predictions and the ground truth by comparing the n-gram overlap between them. CodeBLEU is an extension of the traditional BLEU metric specifically designed for code generation tasks. CodeBLEU evaluates the generated code based on four criteria: syntactic similarity, semantic similarity, token similarity, and structure similarity. It combines these four criteria to produce a single score, which provides a more comprehensive evaluation of the generated code’s quality compared to the traditional BLEU score. In our work, we use both BLEU and CodeBLEU scores, which range from 0 to 1, to evaluate the quality of the code updates generated by our approach and the baseline approaches. We use the BLEU-4 variant, computed by considering the 4-grams in the generated text. BLEU-4 has been
Table 2. Performance of CodeLMs on the time-ignore setting

| Approach     | Android BLEU | CodeBLEU | Google BLEU | CodeBLEU | Ovirt BLEU | CodeBLEU | AndroZooUpdate-S BLEU | CodeBLEU | PP% | BLEU | CodeBLEU | PP% | BLEU | CodeBLEU | PP% | BLEU |
|--------------|--------------|----------|-------------|----------|------------|----------|------------------------|----------|-----|------|----------|-----|------|----------|-----|------|
| CodeT5       | 20.9%        | 0.86     | 0.85        | 18.4%    | 0.86       | 30.0%    | 0.86                   | 28.9%    | 0.95| 0.93|
| CodeT5+      | 24.9%        | 0.87     | 0.86        | 16.0%    | 0.86       | 30.8%    | 0.85                   | 29.4%    | 0.93| 0.92|
| TufanoT5     | 17.2%        | 0.81     | 0.84        | 12.5%    | 0.78       | 16.5%    | 0.81                   | 21.0%    | 0.76| 0.84|
| CodeBERT     | 21.8%        | 0.86     | 0.85        | 13.5%    | 0.81       | 16.5%    | 0.82                   | 27.8%    | 0.85| 0.92|
| CodeGPT      | 26.4%        | 0.85     | 0.85        | 16.9%    | 0.83       | 30.1%    | 0.84                   | 36.9%    | 0.96| 0.94|
| UniXcoder    | 20.5%        | 0.86     | 0.85        | 9.5%     | 0.85       | 18.6%    | 0.83                   | 37.6%    | 0.94| 0.93|
| AutoTransform| 18.7%        | 0.87     | 0.86        | 9.5%     | 0.85       | 21.5%    | 0.85                   | 29.9%    | 0.96| 0.93|

Previously used in other software engineering papers (e.g., [84, 90]). A higher score in both BLEU and CodeBLEU (close to 1) indicates better performance and higher similarity to the ground-truth code recommendation.

**Implementation & Hyperparameter Settings.** We obtain the studied pre-trained models from their respective publicly available repositories and use the Hugging Face library [92] to load the model weights and generate outputs. Given the nature of our code recommendation task, we append a transformer decoder to the encoder-based models (i.e., CodeBERT) for generating code in a regressive manner. In contrast, the decoder (CodeGPT) or encoder-decoder (CodeT5) models directly generate code in a regressive manner. The newly added fully connected layers or decoders during fine-tuning are first initialized randomly and then trained (i.e., fine-tuned). When multiple models are provided, we select the "base" version, consistent with prior studies on CodeLMs for software engineering [21, 52, 97]. The models are fine-tuned using an NVIDIA RTX 3090 graphics card for a total of 15 epochs. We set the learning rate to 5e-5 and the batch size to 4, as these are popular settings in code generation and understanding [97].

5 EXPERIMENTAL RESULTS

In this section, we present the results of our empirical study with respect to our five research questions.

**(RQ1) How accurate are CodeLMs at recommending code updates?**

**Motivation.** This research question aims to investigate the performance of pre-trained CodeLMs in recommending code updates in real-world scenarios. Previous studies [51, 81, 83, 97] have evaluated CodeLMs in time-ignore settings, where the models are trained and tested on data without considering the chronological order of the code updates. However, in real-world development, code updates occur in the time-wise manner. Therefore, this research question aims to investigate the performance of pre-trained CodeLMs in recommending code updates in both time-ignore and time-wise scenarios. By evaluating the accuracy of CodeLMs in these two scenarios, we aim to provide insights into their effectiveness and robustness in generating suitable code recommendations.

**Approach.** To answer this RQ, we evaluate the accuracy of CodeLMs and the baseline approaches by calculating the percentage of Perfect Predictions (%PP), BLEU, and CodeBLEU. We employ a beam search with a beam size of 1 to generate candidate code updates.

**Results.** Table 2 presents the performance of CodeLMs on four code update datasets in time-ignore scenarios. First, compared to the baseline approach TufanoT5, the pre-trained CodeT5 and CodeT5+ achieve higher accuracy across all datasets. Although the performance varies across different datasets and models, we observe that these CodeLMs in time-ignore scenarios can achieve...
perfect prediction rates of almost 30% at a beam search size of 1. These findings are consistent with prior research [51, 81, 83, 97] that also focused on automated code updates in time-ignore settings.

To further investigate the impact of temporal information on the performance of CodeLMs, we compare their accuracy in time-ignore and time-wise scenarios. Figure 2 presents the %PP in these two scenarios, and we calculated the decrease in %PP using the formula:

$$\text{Decrease in } %PP = \frac{(PP_{\text{time-ignore}} - PP_{\text{time-wise}})}{PP_{\text{time-ignore}}} \times 100%$$

The results reveal a significant decrease in performance when moving from time-ignore to time-wise settings across all CodeLMs and datasets. Compared to time-ignore settings, CodeLMs perform poorly in time-wise scenarios. In the Android, Google, and Ovirt datasets, the %PP at a beam search size of 1 is close to 1%, which is extremely low. The decrease rates range from 58.7% to 100%, highlighting the limitations of previous research that focused solely on time-ignore scenarios. This finding highlights the limitations of previous research that solely relied on time-ignore evaluations and emphasizes the need for more realistic assessments of CodeLMs’ performance.

To mitigate the low accuracy in time-wise scenarios, we explore the impact of increasing the beam search size. Figure 3 demonstrates the effect of varying beam sizes (1, 5, 10, and 15) on the %PP of CodeLMs in both time-ignore and time-wise scenarios for the AndroZooUpdates-S dataset. The results show that increasing the beam size can improve accuracy, especially in time-ignore scenarios. For instance, CodeT5 achieves a %PP of 56% with a beam size of 15 in the time-ignore scenario, compared to 29% with a beam size of 1. However, the improvements are less pronounced in time-wise scenarios, with CodeT5 reaching a %PP of 30% with a beam size of 15, compared to 6% with a beam size of 1. While increasing the beam size can improve accuracy, it may not be practical in real-world scenarios due to computational constraints and the need for developers to manually review multiple candidate updates.
We further investigate the generalizability of CodeLMs by comparing their performance on within-project and cross-project code updates in time-ignore settings. Figure 4 presents the %PP of CodeLMs for each dataset in time-ignore settings, calculated based on whether the test instances’ project is also present in the training data (within-project) or not (cross-project). In the within-project scenario, the test instances are from projects that are also represented in the training data, while in the cross-project scenario, the test instances are from projects that are not present in the training data. The results show a significant decrease in performance when moving from within-project to cross-project settings across all CodeLMs and datasets. The decrease rates range from 81.0% to 98.2%, indicating that CodeLMs struggle to generalize to unseen projects, even in time-ignore settings. This finding highlights the importance of considering the generalizability of
Automatically Recommend Code Updates: Are We There Yet?

In summary, our results show that CodeLMs perform well in time-ignore scenarios but struggle in more realistic time-wise evaluations and cross-project code updates. The significant decrease in accuracy when considering temporal information highlights the need for further research to improve the robustness and generalizability of CodeLMs. The varying performance across different datasets and models motivates our subsequent analyses to understand the factors contributing to CodeLMs’ successes and failures in recommending code updates. The following sections will delve deeper into these aspects, providing a more comprehensive understanding of the capabilities and limitations of CodeLMs in recommending code updates.

**Good:** CodeLMs perform well on time-ignore scenarios, achieving up to 36.9% accuracy, and increasing the beam search can improve their performance.

**Bad:** CodeLMs perform poorly in realistic time-wise evaluations and cross-project code updates, with accuracy decreases ranging from 58.7% to 100%.

**Missing:** Further research is needed to understand the factors contributing to CodeLMs’ low accuracy in time-wise and cross-project scenarios and to develop strategies for improving their robustness and generalizability in real-world applications.

(RQ2) **What is the syntactical correctness of the code generated by CodeLMs?**

**Motivation.** While accuracy metrics (like perfect prediction rate) are important for evaluating CodeLMs on code updates, they don’t guarantee that the generated code is syntactically correct or functional. Therefore, this research question aims to investigate whether the code generated
by CodeLMs passes syntactic checking, providing insights into the reliability and quality of the recommended code updates.

**Approach.** To assess the syntactical correctness of the generated code, we conduct syntactic checking on the generated programs for each instance in the testing dataset. Following prior work [48, 56], we utilize PMD [12], a cross-language static code analysis tool that identifies programming errors. We disregard style issues and consider an instance syntactically correct if it successfully passes the checking process.

**Results.** Figure 5 presents the syntactical correctness rate of the code generated by CodeLMs across the four datasets in both time-ignore and time-wise scenarios. The results show that, in general, the state-of-the-art CodeLMs achieve high syntactical correctness rates across all datasets, with most models surpassing 90% correctness in both time-ignore and time-wise scenarios. However, it is important to note that even with high syntactical correctness rates, the generated code is not always perfect. There are still instances where the CodeLMs produce syntactically incorrect code. For example, a common issue is the mismatching of parentheses, such as generating unbalanced or excessive parentheses like "return (((((((((()))))))))", even for better-performing models like CodeT5+. These errors highlight the need for further improvements in the CodeLMs’ ability to generate syntactically correct code. Comparing the syntactical correctness rates between time-ignore and time-wise scenarios, we observe that the rates are generally similar for most CodeLMs. This suggests that the temporal aspect does not significantly impact the syntactical correctness of the generated code. However, there are a few exceptions, such as AutoTransform in the AndroZooUpdate-S dataset, where the syntactical correctness rate drops from 96.0% in the time-ignore scenario to 38.2% in the time-wise scenario. This substantial decrease may be attributed to the model’s sensitivity to temporal information or its limited ability to capture the temporal dependencies present in the AndroZooUpdate-S dataset.

**Good:** State-of-the-art CodeLMs achieve high syntactical correctness rates, often above 90%, in most datasets and scenarios.

**Bad:** Even with high syntactical correctness rates, CodeLMs still generate syntactically incorrect code in some instances, such as mismatching parentheses.

**Missing:** Future work should focus on maintaining and improving the syntactical correctness of code generated by CodeLMs.

(RQ3) Why do CodeLMs perform differently in time-wise and time-ignore scenarios

**Motivation.** From the results of RQ1, we found that CodeLMs perform significantly differently in time-wise and time-ignore scenarios, with a substantial decrease in accuracy when considering temporal information. This raises the question of why there is such a big difference in performance between these two scenarios and what factors contribute to this discrepancy. By investigating the underlying reasons, we can gain insights into the limitations of current CodeLMs and identify potential areas for improvement in recommending code updates.

**Approach.** To understand the relationship between training and test splits, we focus on examining the similarity between the two sets at a fine-grained level. Although the benchmark datasets were cleaned to remove duplications, this process typically involves a high-level comparison rather than a detailed analysis of individual code pairs. In real-world scenarios, code updates often involve a series of similar changes, such as consistently updating a variable name throughout the codebase. To capture these nuances and gain a deeper understanding of the similarities between training and test sets, we obtain simplified versions of each method pair.
As shown in Figure 6, we remove the same code tokens between the prior version and the updated version code, keeping only the simplified updates for each method pair. This process is applied from the start and end of each method, respectively. For example, in Figure 6, the method pair is simplified to “" -> “mFaceDetectionRunning = false;”. In this way, we examine the similarity between training and test sets more closely. By comparing the distribution of seen (simplified test samples in the training) and unseen (simplified test samples not in the training) testing samples, we aim to provide a more nuanced understanding of the factors contributing to the performance difference between time-wise and time-ignore scenarios.

### Results

Table 3 presents the distribution of seen and unseen simplified test samples in the time-ignore and time-wise scenarios for the four datasets. In the time-ignore scenario, the percentage of seen examples ranges from 17.1% to 52.4%, indicating a significant overlap between the training and test sets. In contrast, the time-wise scenario exhibits a much lower percentage of seen examples, ranging from 1.6% to 28.2%. The decrease in the percentage of seen examples from time-ignore to time-wise scenarios is consistent with the decrease in performance observed in Figure 2 in RQ1. Furthermore, these seen examples in the time-ignore scenario usually originate from within the same project, suggesting that CodeLMs may be relying on project-specific similar patterns rather than learning generalizable code update patterns.

Figure 7 further illustrates the impact of seen and unseen simplified test samples on the %PP of CodeLMs. In the time-ignore scenario, CodeLMs achieve significantly higher %PP on seen examples compared to unseen ones. For instance, in the AndroidZooUpdate-S dataset, CodeT5 achieves a %PP of 46.2% on seen examples, while only 9.9% on unseen examples. This trend is consistent across all datasets and CodeLMs, highlighting the reliance of CodeLMs on the presence of similar examples in the training. In the time-wise scenario, the %PP on both seen and unseen examples is generally much lower compared to the time-ignore scenario. However, CodeLMs still perform relatively better on seen examples than unseen ones. For example, in the Google dataset, CodeT5+ achieves a %PP of 10.0% on seen examples, while only 0.4% on unseen examples. We also manually checked the top-seen simplified updates, such as removing nonNull "@ NonNull -> " or changing the "public -> private", and found that they can achieve accuracy as high as almost 90%, as these kinds of simple changes are more common and frequent in the code base. These results indicate that while CodeLM-based models are capable of handling common and frequently seen code updates,
they struggle with novel or complex updates that require a deeper understanding of the code’s temporal evolution.

**Good:** CodeLMs achieve high accuracy on seen test updates, especially in time-ignore scenarios.

**Bad:** CodeLMs struggle with novel or complex code updates, as shown by the low accuracy on unseen simplified test updates.

**Missing:** Future research should focus on improving CodeLMs’ ability to adapt to time-wise updates and generate creative and unseen code updates.

(RQ4) How do the method size and update size impact the accuracy of CodeLMs?

**Motivation.** In RQ1, we observed that the performance of CodeLMs was higher on AndroidZooUpdate-S, which consists of small method pairs, compared to the other three datasets. However, little is known about the accuracy of CodeLMs on pairs of larger methods and on larger or more complex updates (i.e., updates with more changed tokens between versions of a method). Therefore, this research question aims to investigate the impact of method size and update size on the performance of CodeLMs in recommending code updates.

**Approach.** For the Android, Google, and Ovirt datasets, we analyze the perfect prediction rates according to the method size (i.e., the number of tokens in the initial version) and the update size (i.e., the number of changed tokens between two versions) in time-ignore scenarios. For AndroidZooUpdate-S, since we have limited the token size to 50, we use AndroidZooUpdate-L to assess the impact of method and update sizes on CodeLMs’ performance. Due to the dataset’s size...
and the time-consuming nature of training, we focus on the best-performing models, CodeT5 (a pre-trained CodeLM), and AutoTransform (a normal Transformer without pre-training) for this analysis. We train, validate, and test both CodeT5 and AutoTransform accordingly. We further analyze the perfect prediction rates according to the method size and the update size at a beam size of 5.

**Results.** Figure 8 presents the impact of method size and update size on the %PP of CodeLMs for the Android, Google, and Ovirt datasets. The results demonstrate that the majority of the method pairs (69% to 84%) fall within the 0-50 token range, and CodeLMs achieve their highest %PP on these smaller methods. However, as the method size increases, the proportion of method pairs decreases, and the %PP of CodeLMs drops significantly, reaching 0% for methods with more than 150 tokens. A similar trend is observed for update sizes, with CodeLMs performing better on updates with fewer changed tokens (0-5) and struggling with larger updates (>20 changed tokens). This trend is consistent across all CodeLMs and datasets, highlighting the challenges faced by CodeLMs in handling larger and more complex code updates.

Fig. 8. The impacts of method size and the update size.
Table 4. The accuracy of CodeT5 according to the method size and the update size.

| Method size | 0-50 | 50-100 | 100-150 | 150-200 | 200+ |
|-------------|------|--------|---------|---------|------|
| #Updated tokens | PP% | B | CB | PP% | B | CB | PP% | B | CB | PP% | B | CB |
| 0-5 | 15% | 0.39 | 0.75 | 12% | 0.88 | 0.89 | 11% | 0.94 | 0.92 | 11% | 0.96 | 0.93 | 5% | 0.35 | 0.48 |
| 5-10 | 8% | 0.42 | 0.70 | 8% | 0.87 | 0.85 | 8% | 0.93 | 0.89 | 8% | 0.95 | 0.91 | 5% | 0.52 | 0.60 |
| 10-15 | 9% | 0.38 | 0.66 | 7% | 0.82 | 0.81 | 7% | 0.92 | 0.87 | 5% | 0.94 | 0.89 | 3% | 0.52 | 0.60 |
| 15-20 | 6% | 0.39 | 0.65 | 9% | 0.81 | 0.79 | 8% | 0.89 | 0.84 | 7% | 0.92 | 0.87 | 3% | 0.44 | 0.54 |
| 20-25 | 7% | 0.38 | 0.61 | 4% | 0.78 | 0.74 | 6% | 0.87 | 0.81 | 4% | 0.91 | 0.85 | 1% | 0.48 | 0.56 |
| 25+ | 5% | 0.24 | 0.62 | 2% | 0.54 | 0.61 | 3% | 0.70 | 0.66 | 2% | 0.71 | 0.68 | 1% | 0.02 | 0.24 |

Note: B represents BLEU score and CB represents CodeBLEU score.

We also evaluate the accuracy of CodeT5 and AutoTransform when considering the AndroidZooUpdate-L dataset, which includes method pairs of various sizes. CodeT5 achieves a %PP of 8%, while AutoTransform achieves a %PP of 4%, demonstrating the superiority of pre-trained CodeLMs over normal Transformers without pre-training. Table 4 provides a more detailed analysis of CodeT5’s accuracy according to the method size and update size on the AndroidZooUpdate-L dataset. The results show that CodeT5 performs better on smaller methods and updates, with a %PP of 15% for methods with 0-50 tokens and updates with 0-5 changed tokens. However, as the method size and update size increase, the %PP decreases. For methods with more than 200 tokens and updates with more than 25 changed tokens, the %PP drops to 1%. Additionally, the BLEU and CodeBLEU scores also decrease as the method and update sizes increase, indicating a decline in the quality of the generated code updates.

In summary, although our previous experiments show that CodeLMs achieve much higher performance in time-ignore scenarios, their accuracy is still limited when dealing with larger methods and more complex updates. Even in time-ignore settings, the %PP of CodeLMs approaches zero for methods with more than 150 tokens and updates with more than 20 changed tokens. This highlights the need for further research to develop CodeLMs that can effectively handle larger and more complex code updates, as these are common in real-world software development scenarios.

Good: CodeLMs perform better on smaller methods and updates with fewer changed tokens.
Bad: CodeLMs’ accuracy decreases significantly as method size and update complexity increase, dropping to 0% for larger methods and more complex updates.
Missing: Future research should focus on developing CodeLMs that can effectively handle larger and more complex code updates prevalent in real-world scenarios.

(RQ5) What types of code updates are learned and predicted by CodeLMs?

Motivation. While RQ1 to RQ4 focus on the perfect prediction rates (%PP) of CodeLMs in recommending code updates, this final research question aims to provide a qualitative analysis by examining the detailed code updates recommended by CodeLMs. By investigating the types of code updates learned and predicted by CodeLMs, we can gain insights into their strengths and limitations in capturing and generating specific code changes.

Approach. To analyze the types of code updates learned and predicted by CodeLMs, we employ a similar method to RQ3. However, instead of focusing on the simplified updates within testing and training sets, we examine the simplified updates between model inputs and model predictions. By removing the identical code tokens between the input and output, we obtain simplified versions.
Fig. 9. Rate of unchanged updates for each CodeLM across the four datasets.

Table 5. Top 10 Simplified Code Updates Recommended by CodeLMs in Android Dataset

| Simplified Code Update | Training% | CodeT5 | CodeT5+ | TufanoT5 | CodeBERT | CodeGPT | UniXcoder | AutoTransform |
|------------------------|-----------|--------|---------|----------|----------|---------|-----------|---------------|
| “os” -> “system”       | 2.2%      | 100%(28) | 97%(30) | 100%(27) | 100%(28) | 97%(29) | 100%(27) |
| “libcore.io” -> “android.system” | 1.3% | 94%(16) | 94%(16) | 94%(16) | 94%(16) | 94%(16) | 94%(16) |
| “Constants.RIL” -> “” | 0.7%      | 100%(14) | 100%(14) | 100%(10) | 100%(16) | 100%(16) | 100%(15) |
| “ monotone”            | 0.4%      | 42%(12) | 42%(12) | 43%(7)   | 67%(12)  | 60%(5)  | 60%(10)  |
| “-> “gtepuu.lang.Override” | 0.7% | 64%(11) | 100%(6) | 100%(6) | 82%(11) | 100%(7) | 67%(6) |
| “addProperty(Options.LAMBDA_MODE.getName(),Options.LAMBDA_MODE.toString()).” -> “” | 0.6% | 90%(10) | 100%(9) | 100%(9) | 100%(9) | 100%(9) | 100%(8) |
| “this.” -> “”          | 0.2%      | 63%(8)  | 67%(6)  | 86%(7)   | 75%(8)   | 57%(7)  | 60%(10)  |
| “jack” -> “”           | 0.6%      | 100%(7) | 100%(7) | 100%(7)  | 100%(7)  | 100%(7) | 100%(7) |
| “getKey(256)” -> “CRYPT_KEY” | 0.1% | 100%(7) | 100%(7) | 100%(7) | 100%(7) | 100%(7) | 100%(7) |

of the code updates recommended by CodeLMs. This allows us to identify and categorize the types of code changes captured by the models.

**Results.** By simplifying the input and output code, we found that a significant portion of the updates recommended by CodeLMs are unchanged, meaning that the models output the same content as the input. As shown in Figure 9, it is evident that the rate of unchanged updates is remarkably high, especially under the time-wise scenarios. In the Android dataset, CodeT5+ exhibits an unchanged update rate of 91.7% in the time-wise scenario, while CodeT5 and UniXcoder have rates of 84.5% and 73.4%, respectively. Similarly, in the Google dataset, CodeT5+ and UniXcoder have high rates of unchanged updates (88.4% and 84.4%, respectively) in the time-wise scenario. These findings align with our previous discussions, where we observed that the accuracy of CodeLMs is much lower under the time-wise scenarios compared to the time-ignore scenarios. Interestingly,
even in the time-ignore scenarios, the rate of unchanged updates is still substantial. For instance, in the Google dataset, CodeT5 and CodeT5+ have unchanged updates almost 60%. These findings suggest that face challenges in recommending meaningful code changes.

To further investigate the types of code updates learned and predicted by CodeLMs, we examine the top simplified code updates in each dataset. Table 5 presents the top 10 simplified code updates recommended by CodeLMs for the Android dataset under the time-ignore scenario, along with the percentage of the update existing in the training set and the relevant %PP. For the Android dataset, each model generates 1,836 updates. From Table 5, we observe that CodeLMs can accurately predict certain types of code updates, such as converting "os" to "system" and "libcore.io" to "android.system". These updates are relatively simple and occur frequently in the training set, enabling the models to learn and predict them effectively. For example, the update "os" -> "system" exists in 2.2% of the training set, and we also checked that these test examples are from the same repository. Code 1 shows an example of this type of update, where the package name is changed from "android.os" to "android.system".

Code 1. Examples of code update ("os" -> "system").

```java
- public android.os.StructUtname uname() {
+ public android.system.StructUtname uname() {
- public void setegid(int egid) throws android.os.ErrnoException;
+ public void setegid(int egid) throws android.system.ErrnoException;
```

**Good:** CodeLMs accurately predict simple and frequent code updates.

**Bad:** CodeLMs struggle to recommend meaningful changes, especially in time-wise scenarios, resulting in high rates of unchanged updates.

**Missing:** Future research should focus on improving CodeLMs’ adaptability and generalization capabilities to recommend meaningful changes in the presence of new and unseen code patterns.

6 DISCUSSION

**Efficiency.** One of the important factors when considering the adoption of CodeLMs in practice is the computation cost. Hence, we measure the fine-tuning and inference time for the CodeLMs. As we described, the training and inference time is based on a computer with an NVIDIA RTX 3090 graphics card. We find that the fine-tuning time of CodeLMs ranges from 7 hours to 13 hours depending on the model types and the number of epochs. Specifically, models such as T5 and CodeT5 required an average of seven hours for fine-tuning. In contrast, models like CodeBERT and CodeGPT exhibited longer fine-tuning times, often exceeding 10 hours. The average inference time for generating one sequence candidate per method is 0.3 seconds, which is relatively fast and acceptable. However, when we train the CodeLMs on the larger AndroidZooUpdate-L dataset for the same 15 epochs (RQ4), the fine-tuning time increases significantly, requiring almost four days. This indicates that the computation cost of CodeLMs increases dramatically with increasing method size and dataset size. This poses a challenge to the scalability and applicability of CodeLMs for code update recommendation. To address this challenge, future research should explore techniques for improving their efficiency, such as distillation, pruning, and quantization [36].

**Evaluation metrics.** In our study, we used three evaluation metrics to assess the performances of CodeLMs: %PP, BLEU, and CodeBLEU. As shown in Table 2, these metrics generally exhibit a consistent trend when comparing different CodeLMs under the same beam size. However, an interesting observation is that while CodeT5 only achieves a 20.9% perfect prediction rate on
Automatically Recommend Code Updates: Are We There Yet?

Android dataset, it scores high on BLEU (0.86) and CodeBLEU (0.85). This suggests that a non-perfect prediction may still achieve a high BLEU score if it shares many tokens with the ground truth. Although we primarily focus on the %PP metric in our analysis, it is important to note that other aspects, such as syntactical correctness and functional equivalence, are also crucial for evaluating the quality of code update recommendations. To address this limitation, we suggest that future research should explore more comprehensive evaluation metrics that consider multiple dimensions of code quality. For instance, incorporating static code analysis tools to assess the syntactical correctness of the generated code updates can provide a more robust measure of their validity. Additionally, manual analysis by domain experts can offer valuable insights into the functional equivalence and semantic correctness of the recommended code updates.

7 THREATS TO VALIDITY

In this section, we discuss potential threats that may influence the outcomes of our study.

Threats to the external validity concern the generalizability of our work. Our study is based on the AndroidZooUpdate and Pornprasit et al. [67]. For AndroidZooUpdate, we mainly collect Java method blocks from these Android apps. There is a possibility that these apps may not be representative of open-source Android apps developed in the Kotlin programming language, which Google officially supported for Android development starting in 2017 [3]. However, Java is a widely used programming language, extensively employed in Android development [28, 54]. Additionally, our selection criteria, which required apps to be published on Google Play and have more than one commit message on GitHub, led to the exclusion of a large number of apps from the AndroZooOpen dataset. While this was necessary to ensure that we were working with apps that are actively maintained, it does introduce a potential threat to the generalizability of our results. However, to mitigate this threat and maintain the generality of our findings, we have included popular benchmark datasets Pornprasit et al. [67] (i.e., Android, Google, Ovirt) in our analysis alongside our collected dataset.

Threats to the internal validity relate to the impact of the hyperparameter settings on the performance of CodeLM-based models. During fine-tuning, we also did not alter the model’s architecture (e.g., number of layers). We acknowledge that a comprehensive calibration would certainly yield better results. However, optimizing the settings of the parameter for deep learning approaches is computationally expensive. Furthermore, our research objective does not involve determining the optimal settings but rather aims to fairly investigate the feasibility of learning and recommending code updates using CodeLMs.

Threats to the construct validity relate to the data collection process for AndroidZooUpdate, as it may have an impact on the accuracy of the CodeLMs. We perform data cleaning and filtering steps to mitigate the noise in AndroidZooUpdate (e.g., removing duplicated methods pairs). However, unknown noise may still exist in our curated dataset. Future work should further investigate unknown noise and explore its impact on the evaluation of code update recommendation approaches using CodeLMs.

8 RELATED WORK

8.1 Deep Learning for Software Engineering

In the past decade, Deep Learning (DL) has emerged as a dominant research domain, bringing several breakthroughs in image, text, video, and speech processing [50]. DL has also been extensively applied to natural language processing and source code analysis in software engineering [94]. Unlike traditional machine learning techniques that rely heavily on manually-crafted feature representations, deep learning enables the automatic learning of feature representations from raw
data [29, 50]. Motivated by the remarkable success of deep learning techniques, researchers have demonstrated considerable interest in applying these techniques to various software engineering tasks (e.g., code search [30], code generation [47], defect prediction & localization [14, 37, 68, 69, 91], vulnerability prediction [15, 18, 19, 21–24], test case generation [6, 7, 90], code review [34, 35, 66, 81–83], CI/CD [33], and DevSecOps [20]. Specific to the mobile-software engineering context, researchers also proposed various automated approaches for UI generation [9, 10], mobile malware classification [96, 98] and mobile privacy policy analysis [32].

8.2 AI/ML-based Experimental Bias in Software Engineering

According to the survey results presented by Yang et al. [94], machine learning or deep learning technologies have been increasingly used in software engineering to improve developers’ productivity and software quality. However, careful consideration must be given when evaluating AI/ML-based approaches in software engineering, such as defect prediction [76], malware classification [65], and large language models [73]. Prior studies have raised various potential issues that may impact the accuracy and interpretation of AI/ML models in SE. These issues include the quality of datasets [26, 78], data labeling techniques [95], feature selection techniques [27, 42], collinearity analysis [40–42], class rebalancing techniques [77], model construction [26], parameter optimization [4, 5, 25, 77, 79], model evaluation [44, 68, 80], and model interpretation [40, 43].

Extending the concerns related to evaluation techniques [44, 68, 80], this paper is the first to demonstrate the impact of the time-wise evaluation scenario for various CodeLM-based code generation tasks (including our own code update recommendation tool and the automated code transformation tool in prior work [81, 83]). Thus, our finding highlights the importance of the time-wise evaluation scenario, which should be considered in future work.

9 CONCLUSION

In this work, we investigated the application of CodeLMs, specifically CodeT5, CodeT5+, CodeBERT, CodeGPT, and UniXcoder, to the task of automatically recommending code updates. Through an extensive evaluation on two diverse datasets, we assessed the performance of various CodeLMs across different scenarios, considering factors such as temporal evolution, project specificity, method size, and update complexity. Our findings reveal that while CodeLMs demonstrate promising results in settings that ignore temporal information, their performance significantly declines in more realistic time-wise scenarios. The accuracy of CodeLMs drops by 58.7% to 100% when faced with the challenges of adapting to new projects and handling the temporal nature of code updates. Furthermore, we find that CodeLMs often generate syntactically incorrect or null updates, especially for larger methods and more complex edits, highlighting their limitations in producing reliable and meaningful code changes. Our analysis also uncovers that CodeLMs heavily rely on previously seen samples, struggling to generalize to novel and challenging updates. These findings underscore the significant gap between the perceived and actual effectiveness of CodeLMs for practical code update recommendation. While CodeLMs have demonstrated impressive results on various software engineering benchmarks, our study reveals that their performance does not directly translate to real-world scenarios, emphasizing the need for further research and development efforts to bridge this gap and realize the full potential of CodeLMs in automating code updates.

REFERENCES

[1] (2021) Google bigquery. [Online]. Available: https://console.cloud.google.com/marketplace/details/github/github-repos
[2] (2022) Gerrit code review. [Online]. Available: https://www.gerritcodereview.com/
[3] Kotlin, 2022. [Online]. Available: https://kotlinlang.org/
Automatically Recommend Code Updates: Are We There Yet?

[4] A. Agrawal and T. Menzies, “Is ‘better data’ better than ‘better data miners’?” in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 1050–1061.

[5] A. Agrawal, T. Menzies, L. L. Minku, M. Wagner, and Z. Yu, “Better software analytics via "duo": Data mining algorithms using/used-by optimizers,” Empirical Software Engineering, vol. 25, no. 3, pp. 2099–2136, 2020.

[6] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, “A3test: assertion-augmented automated test case generation (2023),” DOI: https://doi.org/10.48550/ARXIV, vol. 2023.

[7] S. Alagarsamy, C. Tantithamthavorn, C. Arora, and A. Aleti, “Enhancing large language models for text-to-testcase generation,” arXiv preprint arXiv:2402.11910, 2024.

[8] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., “Language models are few-shot learners,” Advances in neural information processing systems, vol. 33, pp. 1877–1901, 2020.

[9] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, “From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation,” in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 665–676.

[10] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhut, G. Li, and J. Wang, “Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning,” in 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, 2020, pp. 322–334.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., “Evaluating large language models trained on code,” arXiv preprint arXiv:2107.03374, 2021.

[12] T. Copeland, PDM applied. Centennial Books San Francisco, 2005, vol. 10.

[13] J. Czerwonka, M. Greiler, and J. Tilford, “Code reviews do not find bugs. how the current code review best practice slows us down,” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2. IEEE, 2015, pp. 27–28.

[14] H. K. Dam, Th. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, “Lessons learned from using a deep tree-based model for software defect prediction in practice,” in 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, 2019, pp. 46–57.

[15] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, “Automatic feature learning for predicting vulnerable software components,” IEEE Transactions on Software Engineering, vol. 47, no. 1, pp. 67–85, 2018.

[16] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, “Large language models for software engineering: Survey and open problems,” arXiv preprint arXiv:2310.03533, 2023.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., “Codebert: A pre-trained model for programming and natural languages,” in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[18] M. Fu, V. Nguyen, C. Tantithamthavorn, D. Phung, and T. Le, “Vision transformer inspired automated vulnerability repair,” ACM Transactions on Software Engineering and Methodology, vol. 33, no. 3, pp. 1–29, 2024.

[19] M. Fu, V. Nguyen, C. K. Tantithamthavorn, T. Le, and D. Phung, “Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types,” IEEE Transactions on Software Engineering, vol. 49, no. 10, pp. 4550–4565, 2023.

[20] M. Fu, J. Pasuksmit, and C. Tantithamthavorn, “Ai for devsecops: A landscape and future opportunities,” arXiv preprint arXiv:2404.04839, 2024.

[21] M. Fu and C. Tantithamthavorn, “Linevul: A transformer-based line-level vulnerability prediction,” in Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 608–620.

[22] M. Fu, C. Tantithamthavorn, T. Le, Y. Kume, V. Nguyen, D. Phung, and J. Grundy, “Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities,” Empirical Software Engineering, vol. 29, no. 1, p. 4, 2024.

[23] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, “Vulrepair: a t5-based automated software vulnerability repair,” in Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, 2022, pp. 935–947.

[24] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, “Chatgpt for vulnerability detection, classification, and repair: How far are we?” in 2023 30th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2023, pp. 632–636.

[25] W. Fu, T. Menzies, and X. Shen, “Tuning for software analytics: Is it really necessary?” Information and Software Technology, vol. 76, pp. 135–146, 2016.

[26] B. Ghotra, S. McIntosh, and A. E. Hassan, “Revisiting the impact of classification techniques on the performance of defect prediction models,” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1. IEEE, 2015, pp. 789–800.

[27] ——, “A large-scale study of the impact of feature selection techniques on defect classification models,” in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, 2017, pp. 146–157.
2018.

[78] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, “The impact of mislabelling on the performance and interpretation of defect prediction models,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 812–823.

[79] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “Automated parameter optimization of classification techniques for defect prediction models,” in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 321–332.

[80] ——, “An empirical comparison of model validation techniques for defect prediction models,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2016.

[81] P. Thongtanunam, C. Pornprasis, and C. Tantithamthavorn, “Autotransform: Automated code transformation to support modern code review process,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 237–248.

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

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

[84] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshvyvanyk, “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 832–837.

[85] R. Tufano, S. Masiero, A. Mastropaulo, L. Pascarella, D. Poshvyvanyk, and G. Bavota, “Using pre-trained models to boost code review automation,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 2291–2302.

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

[87] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, “Bridging pre-trained models and downstream tasks for source code understanding,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 287–298.

[88] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, “Codet5+: Open code large language models for code understanding and generation,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 1069–1088.

[89] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[90] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshvyvanyk, “On learning meaningful assert statements for unit test cases,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[91] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, “Predicting defective lines using a model-agnostic technique,” *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1480–1496, 2020.

[92] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz et al., “Huggingface’s transformers: State-of-the-art natural language processing,” arXiv preprint arXiv:1910.03771, 2019.

[93] H. Xia, Y. Zhang, Y. Zhou, X. Chen, Y. Wang, X. Zhang, S. Cui, G. Hong, X. Zhang, M. Yang et al., “How android developers handle evolution-induced api compatibility issues: a large-scale study,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 886–898.

[94] Y. Yang, X. Xia, D. Lo, and J. Grundy, “A survey on deep learning for software engineering,” *ACM Computing Surveys*, vol. 54, no. 10, p. 206, 2022.

[95] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, “Mining software defects: Should we consider affected releases?” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 654–665.

[96] Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[97] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, “An extensive study on pre-trained models for program understanding and generation,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 39–51.

[98] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, “Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware,” in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 757–770.
[99] Y. Zhao, L. Li, K. Liu, and J. Grundy, “Towards automatically repairing compatibility issues in published android apps,” in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2142–2153.