# AutoTransform: Automated Code Transformation to Support Modern Code Review Process

Patanamon Thongtanunam*
patanamon.t@unimelb.edu.au
The University of Melbourne
Australia

Chanathip Pornprasit†
chanathip.pornprasit@monash.edu
Monash University
Australia

Chakkrit Tantithamthavorn
chakkrit@monash.edu
Monash University
Australia

## ABSTRACT

Code review is effective, but human-intensive (e.g., developers need to manually modify source code until it is approved). Recently, prior work proposed a Neural Machine Translation (NMT) approach to automatically transform source code to the version that is reviewed and approved (i.e., the after version). Yet, its performance is still suboptimal when the after version has new identifiers or literals (e.g., renamed variables) or has many code tokens. To address these limitations, we propose AutoTransform which leverages a Byte-Pair Encoding (BPE) approach to handle new tokens and a Transformer-based NMT architecture to handle long sequences. We evaluate our approach based on 14,750 changed methods with and without new tokens for both small and medium sizes. The results show that when generating one candidate for the after version (i.e., beam width = 1), our AutoTransform can correctly transform 1,413 changed methods, which is 567% higher than the prior work, highlighting the substantial improvement of our approach for code transformation in the context of code review. This work contributes towards automated code transformation for code reviews, which could help developers reduce their effort in modifying source code during the code review process.

**ACM Reference Format:**
Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. AutoTransform: Automated Code Transformation to Support Modern Code Review Process. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510067

## 1 INTRODUCTION

Code review is one of the important quality assurance practices in a software development process. One of the main goals of code review is to ensure that the quality of newly developed code meets a standard before integrating into the main software repository [11, 37]. Hence, in the code review process, new source code written by a code author has to be examined and revised until reviewers (i.e., developers other than a code author) agree that this source code is of sufficient quality to be integrated (i.e., the code is approved). Several studies also showed that during the code review process, source code is revised not only to remove functional defects [15, 17, 40, 47], but also to improve design quality [41], maintainability [50], and readability [16, 50, 57].

Despite the benefits of code review, the code review process is still human-intensive where developers have to manually review and revise code. To support reviewing activities, prior studies proposed approaches to save the reviewers' effort (e.g., review prioritization [14, 23, 55]). However, the revising activities still require manual effort from code authors. Indeed, given a large number of reviews (e.g., 3K reviews per month at Microsoft Bing [47]), prior studies 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 [21, 32]. Thus, automated code transformation would be beneficial to augment code authors and to save their effort by automatically applying the common revisions during code reviews in the past to the newly-written code, while allowing code authors to focus on revising more complex code.

To the best of our knowledge, the work of Tufano *et al.* [60] is the most recent work that proposed an approach to automatically transform code to the version that is reviewed and approved. In the prior work, they leveraged code abstraction and a Recurrent Neural Network (RNN) architecture. They have shown that their NMT approach can correctly transform code by applying a wide range of meaningful code changes including refactoring and bug-fixing. While the results of the prior work [60] highlighted the potential of using NMT to help code authors automatically modify code during the code review process, the applicability of their approach is still limited. Specifically, their code abstraction hinders the Tufano *et al.* approach in transforming source code when new identifiers or literals appear in the version that has been approved. In addition, due to the nature of the RNN architecture, the Tufano *et al.* approach may be suboptimal when the size of source code is longer (i.e., source code has many tokens).

In this paper, we propose a new framework called AutoTransform, which leverages Byte-Pair Encoding (BPE) subword tokenization [52] and Transformer [62] to address the limitations of the prior approach [60]. We evaluated our AutoTransform based on two types of changed methods: (1) the changed methods **without** newly-introduced identifiers/literals (i.e., changed methods w/o new tokens); and (2) the changed methods **with** newly-introduced identifiers/literals (i.e., the changed methods w/ new tokens). Through a case study of 147,553 changed methods extracted from three Gerrit code review repositories of Android, Google, and Ovirt, we addressed the following research questions:

*A corresponding author.
†The first and second authors contributed equally to this research.


*ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA*

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510067

**(RQ1) Can AutoTransform transform code better than Tufano *et al.* approach?**
**Results.** When generating one candidate of the after version for 14,750 changed methods in the testing datasets, our AutoTransform can correctly transform source code for 1,413 changed methods which is 567% higher than the Tufano *et al.* approach. Specifically, for the changed methods w/ new tokens, our AutoTransform can correctly transform 1,060 methods, while the Tufano *et al.* approach could not correctly transform any of the changed methods w/ new tokens. For the changed methods w/o new tokens, our AutoTransform can correctly transform 353 changed methods, while the Tufano *et al.* approach can correctly transform only 212 changed methods.

**(RQ2) What are the contributions of AutoTransform's components?**
**Results.** Using subword tokenization by BPE enables our AutoTransform to achieve perfect predictions 284% higher than using code abstraction like the Tufano *et al.* approach. Furthermore, using the Transformer architecture can increase perfect predictions at least by 17%, compared to using the RNN architecture. In particular, we found that the percentage improvement in perfect predictions is much higher for the medium methods (i.e., 183% - 507%).

**Significance & Contributions.** The results of our work highlight the substantial improvement of our approach for code transformation in the context of code review. More specifically, our AutoTransform can transform a wider range of changed methods (i.e., methods with new tokens and methods with long sequences) than the state-of-the-art approach [60]. The proposed approach, results, and insights presented in this paper contribute towards automated code transformation for code reviews, which could help developers reduce their effort in modifying source code and expedite the code review process.

**Novelty.** This paper is the first to present:

- AutoTransform, i.e., a Transformer-based NMT approach to transform source code from the version before the implementation of code changes to the version that is reviewed and eventually merged.
- The use of Byte-Pair Encoding (BPE) subword tokenization to address the limitation of transforming changed methods that have newly-introduced identifiers/literals.
- An empirical evaluation of our AutoTransform and the Tufano *et al.* approach [60] based on a large-scale dataset with two types of changed methods.
- An ablation study to quantify the contributions of the two components (i.e., BPE and Transformer) in our proposed approach.

**Open Science.** To facilitate future work, the datasets of 147,553 extracted changed methods, scripts of our AutoTransform, and experimental results (e.g., raw predictions) are available online [4].

**Paper Organization.** Section 2 presents and discusses the limitations of the state-of-the-art approach [60]. Section 3 presents our AutoTransform. Section 4 describes our case study design. Section 5 presents the results, while Section 6 discusses the results. Section 7 discusses related work. Section 8 discusses possible threats to the validity. Section 9 draws the conclusions.

## 2 THE STATE-OF-THE-ART CODE TRANSFORMATION FOR CODE REVIEWS

Recently, Tufano *et al.* [60] proposed a Neural Machine Translation (NMT) approach to automatically transform the source code of the *before version* of a method (i.e., the version before the implementation of code changes) to its *after version* (i.e., the version after the code changes are reviewed and merged). Broadly speaking, Tufano *et al.* (1) performed **code abstraction** to reduce vocabulary size by replacing actual identifier/literals with reusable IDs and (2) built an NMT model based on **a Recurrent Neural Network (RNN) architecture** to transform the token sequence of the before version to the token sequence of the after version. Their approach was evaluated based on the changed methods that were extracted from three Gerrit code review repositories, namely Android [3], Google [6], and Ovirt [7]. We briefly describe their approach below.

*(Step 1) Code Abstraction:* Since NMT models are likely to be inaccurate and slow when dealing with a large vocabulary size [26], Tufano *et al.* proposed to abstract code tokens into *reusable IDs* to reduce the vocabulary size. More specifically, for both before and after versions $(m_b, m_a)$ of each changed method, Tufano *et al.* replaced identifiers (i.e., type, method, and variable names) and literals (i.e., int, double, char, string values) with a reusable ID. A reusable ID means that an ID is allowed to be reused across different changed methods (e.g., the first variable appearing in a changed method will be always replaced with `VAR_1`). At the end of this step, from the original source code of $(m_b, m_a)$, the ***abstracted*** code sequences $(am_b, am_a)$ were obtained. In addition, for each changed method, a map $M(am_b)$ was also generated, which will be used to map the IDs back to the actual identifiers/literals.

*(Step 2) Build an NMT model:* To build an NMT model, Tufano *et al.* used a Recurrent Neural Network (RNN) Encoder-Decoder architecture with the attention mechanism [12, 18, 36, 53]. Given the abstracted code sequences $(am_b, am_a)$ from Step 1, an RNN Encoder will learn these sequences by estimating the conditional probability $p(y_1, ..., y_{t'}|x_1, ..., x_t)$, where $x_1, ..., x_t$ is the sequence of $am_b$ and $y_1, ..., y_{t'}$ is the sequence of $am_a$ while the sequence lengths $t$ and $t'$ may differ. A bi-directional RNN Encoder [18] was used to learn the abstracted code sequence of the before version $am_b$ from left-to-right and right-to-left when creating sequence representations. An RNN Decoder was then used to estimate the probability for each token $y_i$ in the abstracted code sequence of the after version $am_a$ based on the recurrent state $s_i$, the previous tokens $y_{1..i}$, and a context vector $c_i$. The vector $c_i$ is the attention vector which is computed as a weighted average of the hidden states, allowing the RNN model to pay attention to particular parts of $am_b$, when predicting token $y_i$ for $am_a$.

*(Step 3) Generate predictions:* Given the abstracted code sequence of the before version of an unseen method (i.e., a testing instance) $am_b^t$, the RNN model in the Tufano *et al.* approach generated the abstracted code sequence of the after version $am_a^t$. A beam search strategy was used to obtain $k$ sequence candidates for $am_a^t$. Since the generated output sequences are the abstracted code sequences, Tufano *et al.* replaced the IDs found in $am_a^t$ back to the actual identifier/literals based on the map $M(am_b^t)$.

**Limitations.** Although the study of Tufano *et al.* sheds light that their NMT approach can transform source code with meaningful

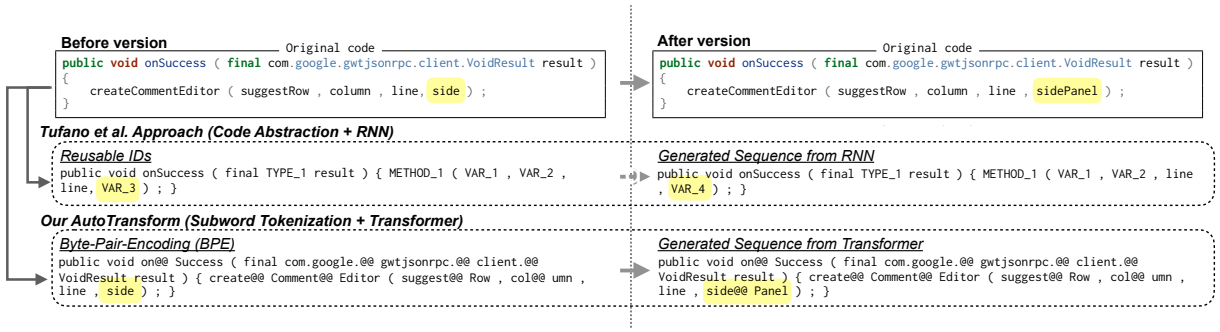


**Figure 1: A motivating example for an unknown identifier/literal for the newly-introduced abstracted token (Limitation 1).**

**Table 1: The number of changed methods with and without new tokens in the after version.**

| Dataset | Method size | w/o New Tokens | w/ New Tokens |
|---|---|---|---|
| Android | Small | 4,437 (18%) | 20,640 (82%) |
| | Medium | 4,593 (16%) | 24,547 (84%) |
| Google | Small | 2,289 (20%) | 9,070 (80%) |
| | Medium | 2,833 (20%) | 11,624 (80%) |
| Ovirt | Small | 4,734 (17%) | 23,282 (83%) |
| | Medium | 6,229 (16%) | 33,275 (84%) |

The detail of data preparation is provided in Section 4.2.

code changes (e.g., bug-fix, refactoring) [60], their approach still has the following limitations.

**Limitation 1: Unknown identifiers/literals for the new tokens appearing in the after version.** We hypothesize that the Tufano *et al.* approach may not be able to correctly transform code when new tokens appear in the *after version* but did not appear in the *before version*. For such a case, the code abstraction approach with ReusableIDs of Tufano *et al.* cannot map the ID of a new token back to the actual identifiers or actual literals.

Indeed, it is possible that developers introduced new tokens that did not appear in the before version. Figure 1 illustrates a motivating example for Limitation 1. As shown in the example, the `side` variable is changed to `sidePanel`. Thus, the `sidePanel` variable is a new token appearing in the after version. By the code abstraction with ReusableID of Tufano *et al.*, a new ID will be assigned to `sidePanel`, i.e., `VAR_4 = sidePanel`. Note that this limitation is different from the Out-Of-Vocabulary (OOV) problem [31] since the Tufano *et al.* approach may still be able to predict the correct ID (i.e., `VAR_4`) for the after version $am_a$ instead of assigning a special tokens (e.g., `<UNK>`). However, this `VAR_4` cannot be realistically mapped back to the actual identifier (i.e., `sidePanel`) since it does not appear in the before version $m_b$ nor its mapping $M(am_b)$.

Moreover, when we analyze the changed methods in the datasets of Tufano *et al.* [60], we find that as much as 80%-84% of the changed method have new tokens appearing in the after version (see Table 1). However, Tufano *et al.* exclude these changed methods with new tokens, while using only the remaining 16% - 20% of the changed methods without new tokens for their experiment [60]. Thus, the Tufano *et al.* approach may be applicable to only a small set of changed methods, limiting its applicability in real-world contexts.

**Limitation 2: Suboptimal performance when the sequences become longer.** Prior studies have raised a concern that the performance of the RNN architecture will be suboptimal when sequences become longer [12, 36]. Although Tufano *et al.* leveraged the attention mechanism to handle changed methods that have long sequences, a recent study by Ding *et al.* [22] noted that such attention mechanism for the RNN architecture still has difficulties in remembering long-term dependencies between tokens in a sequence. In particular, the attention mechanism only computes attention weights based on the final hidden states of the RNN architecture, instead of using any given intermediate states from the encoder, causing the RNN model to forget tokens seen long ago. Thus, we hypothesize that the performance of the Tufano *et al.* approach which is based on the RNN architecture may be suboptimal for the changed methods with long sequences.

## 3 AUTOTRANSFORM

In this section, we present our AutoTransform, a Neural Machine Translation (NMT) approach that can transform code (1) when new tokens appear in the after version; and (2) when code sequences become longer. AutoTransform leverages a Byte-Pair-Encoding (BPE) approach [52] to handle new tokens that may appear in the after version; and leverages a novel NMT architecture called Transformer [62] to address the suboptimal performance of the RNN architecture used in the Tufano *et al.* approach.

**Overview.** Figure 2 provides an overview of our AutoTransform approach, which consists of two stages: training and inference. During the training stage, our AutoTransform performs two main steps. In Step ①, we perform subword tokenization on the original source code of the before and after versions of a changed method $(m_b, m_a)$ using BPE, which produces subword sequences $(sm_b, sm_a)$. In Step ②, we use these subword sequences $(sm_b, sm_a)$ to train a Transformer model. The inference stage is for transforming the before version $(m_b^t)$ of given methods (i.e., testing data) to the after version $(m_a^t)$. To do so, we perform subword-tokenization on the before version in Step ①b to produce a subword sequences $sm_b^t$. Then, in Step ③, we use the Transformer NMT model to generate a prediction for the source code of the after version $m_a^t$. Below, we provide details for each step in our approach.

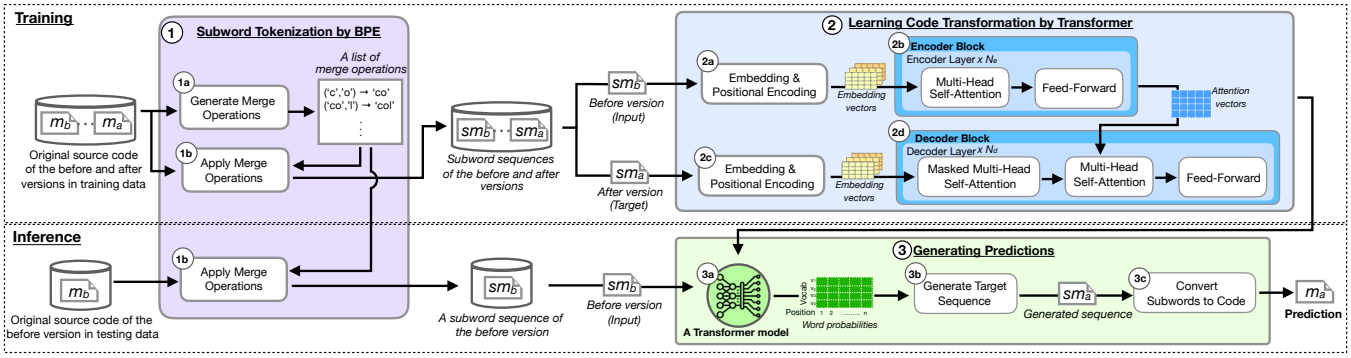


**Figure 2: An overview of our AutoTransform.**

## 3.1 Subword Tokenization

To reduce the vocabulary size and handle new tokens that may appear in the after version, we perform subword tokenization using Byte-Pair-Encoding (BPE) [52]. BPE is a tokenization approach that splits a word (i.e., a code token) into a sequence of frequently occurring subwords. For example, a code token `column` in Figure 1 will be split into a sequence of `col@@` and `umn`, where `@@` is a subword separator. When using BPE, the vocabulary will mainly contain subwords which are more frequently occurring than the whole code tokens. Prior studies also have shown that BPE can effectively address the large vocabulary size than other tokenization approaches (e.g., camel-case splitting) [31]. Moreover, BPE will enable our approach to address the new token by allowing the NMT model to generate a new code token based on a combination of subwords existing across all methods in the training data. For example, the code token `sidePanel` which is introduced in the after version (see Figure 1) can be constructed based on a combination of `side@@` and `Panel` if these two subwords exist in the training data.

The subword tokenization consists of two main steps (see Figure 2). Step (1a) is for generating merge operations which are used to determine how a code token should be split. To generate merge operations, BPE will first split all code tokens into characters sequences. Then, BPE generates a merge operation by identifying the most frequent symbol pair (e.g., the pair of two consecutive characters) that should be merged into a new symbol. For example, given that ('c', 'o') is the most frequent symbol pair in the corpus, the merge operation will be ('c', 'o') → 'co'. Then, BPE replaces all of the co-occurrence of ('c', 'o') with 'co' without removing 'c' or 'o' (which may still appear alone). After the previous merge operation is applied, BPE generates a new merge operation based on the frequency of current symbol pairs, e.g., ('co', 'l') → 'col'. This step is repeated until it reaches a given number of merge operations.

To ensure the consistency of subword tokenization between the before and after versions of a changed method, we use *joint* BPE to generate merge operations based on the union of code tokens from both before and after versions. Note that we generated merge operations based on the training data.

In Step (1b), we apply merge operations to split code tokens into subwords. To do so, BPE will first split all code tokens into sequences of characters. Then, the generated merge operations are applied to the before and after versions in the training data. Note that the lower the number of merge operations applied, the smaller the size of vocabulary is. We also apply the same list of merge operations to the before version in the testing data. Note that we did not split Java keywords since they are commonly used across changed methods.

## 3.2 Learning Code Transformation

To learn code transformation, we train a Transformer-based NMT model using the subword sequences ($sm_b$, $sm_a$) of the before and after versions. Unlike the RNN architecture, the Transformer architecture entirely relies on an attention mechanism without using the sequence-aligned RNNs or convolution networks [62], which allows the Transformer model to better pay attention to any set of tokens across arbitrarily long distances. Transformer uses a *self-attention* function to compute attention weights based on all tokens in a sequence, where attention weights indicate how each token is relevant to all other tokens in the sequence. This self-attention function enables the Transformer model to capture the contextual relationship between all tokens in the whole sequence instead of relying on the limited number of the final hidden states like the RNN architecture. In addition, instead of using a single self-attention, the Transformer architecture employs a multi-head self-attention, which calculates attention weights $h$ times (where $h$ is the number of heads) based on the different parts of input data.

The Transformer architecture consists of two major components: an Encoder block which encodes a subword sequence into a vector representation, and a Decoder block which decodes the representation into another subword sequence. The Transformer performs the following four main steps (see Figure 2). First, in Step (2a), given a subword sequence of the before version $sm_b = (s_1, ..., s_t)$, the Transformer embeds the tokens into vectors and uses positional encoding to add information about the token position of the sequence into the embedding vectors. Second, the embedding vectors are then fed into the encoder block (2b), which is composed of a stack of multiple Encoder layers (where $N_e$ is the number of Encoder layers). Each layer consists of two sub-layers: a multi-head self-attention and a fully-connected feed forward network (FFN) which computes attention weights and generates attention vectors. The attention vectors will be used to inform the Decoder block about which tokens that should be paid attention.

The subword sequence of the after version $sm_a = (u_1, ..., u_{t'})$ is used as an input of the Decoder block. Similarly, in Step (2c), the subword sequence of $sm_a$ is embeded into vectors with the position information. Then, the embedding vectors are fed into the Decoder block (2d) to generate an encoded representation of the target sequence. The Decoder block is also composed of a stack of multiple Decoder layers (where $N_d$ is the number of Decoder layers). Each Decoder layer also consists of multi-head self-attention and FFN sub-layers. However, before the embedding vectors are fed into the multi-head self-attention sub-layer, the masked multi-head self-attention layer is used to ensure that only previous tokens (i.e., $u_1, ..., u_{i-1}$) are used in decoding for a current token $u_i$. After that, the multi-head self-attention with subsequent FFN generates an encoded representation of the target sequence. The encoded representation are converted into word probabilities and the final output sequence using the linear and softmax functions. Finally, the loss function will compare this output sequence with the target sequence to estimate an error.

### 3.3 Generating Predictions

The inference stage aims to generate the after version for a given method based on the before version $m_b^t$ in the testing data. Starting from the before version $m_b^t$, we perform subword tokenization by applying the list of the merge operations generated in the training stage to produce a subword sequence ($sm_b^t$) for the before version in Step (1b). Then, in Step (3), we use our Transformer model to generate the target sequence, i.e., the after version $m_a^t$. In particular, in Step (3a), given the subword sequence $sm_b^t$, the Transformer model will estimate a probability of each subword in the vocabulary at each position in the output sequence. Then, based on the generated subword probabilities, in Step (3b), we use the beam search approach [53] to generate the target sequence.

Broadly speaking, beam search generates the target sequence by selecting the best subword for each position in the output sequence based on the generated subword probabilities. In particular, beam search selects a subword for a current position based on the selected subwords in the previous positions. The beam width $k$ is used to specify the number of previous positions that should be considered and the number of sequence candidates that will be generated. In other words, the selection of a subword for a position $i$ is based on the conditional probabilities of selected subwords in the positions $i-k$ to $i-1$. Finally, the beam search generates the best $k$ sequence candidates for the target sequence.

The sequence candidates generated in Step (3b) are the subword sequences. Hence, in Step (3c), we convert these subword sequences back to the code sequences. This step is simply performed by concatenating the subwords ending with `'@@'` with the subsequent subword. For example, the subwords `['col@@', 'umn']` are converted back to `column`.

## 4 CASE STUDY DESIGN

In this section, we present the motivation of our research questions, data preparation, and experimental setup for our case study.

### 4.1 Research Questions

To evaluate our AutoTransform, we formulate the following two research questions.

**(RQ1) Can AutoTransform transform code better than Tufano *et al.* approach?**

**Motivation.** In this work, we proposed our AutoTransform to address the limitations of the state-of-the-art approach [60]. In particular, the goal of our AutoTransform is to allow an NMT model to better transform code (1) when new tokens appear in the after version; and (2) when the code sequence become longer. Hence, we set out this RQ to evaluate our AutoTransform based on the two aforementioned aspects.

**(RQ2) What are the contributions of AutoTransform's components?**

**Motivation.** To address the limitations of the state-of-the-art approach [60], we used two different techniques in our AutoTransform, i.e., (1) BPE [52] to reduce the vocabulary size and handle new tokens appearing in the after version and (2) Transformer [62] to learn code transformation. In this RQ, we set out to empirically evaluate the contribution of each technique to the performance of our AutoTransform, compared against the techniques used in the Tufano *et al.* approach.

### 4.2 Datasets

In this work, we obtain the datasets from the work of Tufano *et al.* [5]. The datasets consist of 630,858 changed methods which were extracted from 58,728 reviews across three Gerrit code review repositories, namely Android [3], Google [6], and Ovirt [7]. For each changed method, the datasets consist of the source code of the before and after versions, i.e., $(m_b, m_a)$. For our experiment, we perform data preparation on these obtained datasets to classify and select the changed methods. Note that we did not use the filtered datasets that were used in the experiment of Tufano *et al.* [60], since their filtered datasets do not include the changed methods of which the after version contains a new token (which is considered in this work). Table 1 provides an overview of the studied datasets after the data preparation, which we describe in details below.

**Data Preparation.** We first classify the changed methods into two types of changed methods: (1) the changed methods of which the after version *does not* contain new identifiers/literals additionally from the before version (i.e., changed methods ***without*** new tokens) and (2) the changed methods of which the after version *contains* new identifiers/literals additionally from the before version (i.e., changed methods ***with*** new tokens). The changed methods *without* new tokens were used to fairly evaluate our AutoTransform against the Tufano *et al.* approach under the same condition as in the prior experiment [60], while the changed methods *with* new tokens were used to evaluate whether our AutoTransform can transform source code when a new token appears in the after version (i.e., evaluating the hypothesis discussed in Limitation 1).

For the changed methods *without* new tokens, we used the same selection approach as in the prior work [60]. More specifically, the changed methods *without* new tokens are those methods of which the after version $m_a$ contains only (1) Java keywords; (2) Top-300 frequent identifiers/literals [2]; and (3) identifiers and literals that are already available in the before version $m_b$. The remaining changed

**Table 2: Vocabulary size for the original, subword tokenized, and abstracted methods in the training datasets.**

| Dataset (Method size) | Change Type | Original | Subword Tokenized | | Abs |
|---|---|---|---|---|---|
| | | | BPE2K | BPE5K | |
| Android (Small) | w/o new tokens | 12,052 | 2,702 | 4,230 | 356 |
| | w/ new tokens | 43,795 | 7,448 | 9,247 | 408 |
| Google (Small) | w/o new tokens | 5,012 | 1,719 | 2,751 | 333 |
| | w/ new tokens | 13,737 | 3,417 | 4,884 | 383 |
| Ovirt (Small) | w/o new tokens | 9,772 | 1,992 | 3,575 | 306 |
| | w/o new tokens | 30,562 | 4,243 | 6,042 | 355 |
| Android (Medium) | w/o new tokens | 22,296 | 5,165 | 6,860 | 447 |
| | w/ new tokens | 76,264 | 15,585 | 17,874 | 496 |
| Google (Medium) | w/o new tokens | 9,340 | 2,831 | 4,046 | 371 |
| | w/ new tokens | 22,140 | 6,334 | 8,052 | 422 |
| Ovirt (Medium) | w/o new tokens | 17,680 | 3,231 | 4,958 | 353 |
| | w/o new tokens | 44,317 | 7,528 | 9,674 | 422 |

*The w/ and w/o new tokens change types are mutually exclusive sets.

methods of which the after version contains tokens not listed in the aforementioned token categories are classified as the changed methods *with* new tokens. Note that we exclude the changed methods that were completely added or deleted (i.e., $m_b = \emptyset$ or $m_a = \emptyset$) and the changed methods of which before and after versions appear the same (i.e., $m_b = m_a$) since NMT models would not be able to learn any code transformation patterns from these methods. In addition, we remove the duplicate changed methods (i.e., the methods whose $(m_b, m_b)$ are exactly same) to ensure that none of the duplicate methods in testing will appear in training.

After selecting and classifying the changed methods, we separate the datasets into two method sizes, i.e., small and medium to evaluate the hypothesis discussed in Limitation 2 (see Section 2). The **small** methods are those methods of which before and after versions have a sequence length no longer than 50 tokens. The **medium** methods are those methods of which before and after versions have a sequence length between 50-100 tokens. Similar to prior work [60], we disregard the changed methods that have a sequence longer than 100 tokens because large methods have a long tail distribution of sequence lengths with a high variance, which might be problematic when training an NMT model.

In total, we conduct an experiment based on 12 datasets, i.e., 3 repositories (Android, Google, Ovirt) × 2 method sizes (small, medium) × 2 change types (w/o and w/ new tokens). Each of the datasets is then partitioned into training (80%), validation (10%), and testing (10%).

## 4.3 Experimental Setup

This section provides setup details for our experiment.

**Source Code Pre-processing.** Before we perform subword tokenization (for our AutoTransform) and code abstraction (for Tufano *et al.* approach), we perform word-level tokenization to convert the formatted code into a sequence of code tokens. To do so, for each version of each changed method, we simply separate code lines, identifiers, literals, Java keywords, and Java reserved characters (e.g., `;(){}`) by a space. We do not convert code tokens to lower cases because the programming language (i.e., Java) of the studied datasets is case-sensitive. We also do not split code tokens with compound words (e.g., camel-case) since our AutoTransform already performs subword tokenization and such compound-word splitting is not performed in Tufano *et al.* approach.

**Table 3: Hyper-parameter settings for the Transformer and RNN models.**

| Hyper-Parameter | Transformer Model | RNN Model |
|---|---|---|
| #Encoder Layers ($N_e$) | {1, 2} | {1, 2} |
| #Decoder Layers ($N_d$) | {2, 4} | {2, 4} |
| Cell Types | n/a | {GRU, LSTM} |
| #Cells | n/a | {256, 512} |
| Embedding Size | n/a | {256, 512} |
| Attention Size | n/a | {256, 512} |
| #Attention Heads ($h$) | {8, 16} | n/a |
| Hidden Size ($hs$) | {256, 512} | n/a |
| Total #Settings | 8 | 10 |

For our AutoTransform, we use the implementation of Sennrich *et al.* [52] to perform subword tokenization using Byte-Pair Encoding (BPE) [1]. In this work, we experiment with two encoding sizes, i.e., the number of merge operations: 2,000 (*BPE2K*) and 5,000 (*BPE5K*), which substantially reduce the vocabulary size (at least approximately by 50%; see Table 2). For Tufano *et al.* approach, we use src2abs [9] to abstract tokens with reusable IDs (*Abs*) and generate a map $M$ for each changed method. Similar to prior work [60], we do not abstract Java keywords and the top-300 frequent identifier/literals [2]. To ensure that the identifier/literal appearing in both versions has the same ID, we use a pair mode of src2abs for the training and validation data, while a single mode is used for the before version in the testing data.

**NMT Models & Hyper-Parameter Settings.** To build a Transformer model in our AutoTransform, we use the implementation of the Tensor2Tensor library [10]. To build an RNN model in Tufano *et al.* approach, we use the implementation of the seq2seq library [8] which is also used in the prior work [60]. To ensure a fair comparison, we use similar combinations of hyper-parameters (where applicable) for both Transformer and RNN models (see Table 3). Therefore, we experiment with eight hyper-parameter settings for our Transformer model in AutoTransform. For the RNN models, we use ten hyper-parameter settings which are originally used in the experiment of the prior work [60].

When training the models for both approaches, we set the maximum number of epochs similar as in the prior work [60].[1] To avoid the overfitting of our models to the training data, we select the model checkpoint (i.e., the model that was trained until a particular number of epochs) that achieves the lowest loss value computed based on the validation data (not the testing data).

**Evaluation Measure.** To evaluate the performance of our AutoTransform and Tufano *et al.* approach, we measure the number of methods for which an approach achieves a *perfect prediction*, i.e., the generated after version *exactly* matches the ground-truth (i.e., the actual after version). Note that we convert the generated after version (i.e., the subword sequence of our AutoTransform, and

[1] Note that #epochs are calculated based on the size of training data, batch size and #train steps which are calculated differently for the Tensor2Tensor and seq2seq libraries. The details are provided in the Supplementary Materials [4].

**Table 4: Perfect predictions (#PP) of our AutoTransform and Tufano *et al.* approach approach for the small and medium changed method with and without new tokens in the after version. The percentage value in the parenthesis indicates the percentage improvement of our AutoTransform.**

| Dataset (Method Size) | Change Type | #Test | Beam width = 1 AutoTransform #PP | Beam width = 1 Tufano *et al.* #PP | Beam width = 5 AutoTransform #PP | Beam width = 5 Tufano *et al.* #PP | Beam width = 10 AutoTransform #PP | Beam width = 10 Tufano *et al.* #PP |
|---|---|---|---|---|---|---|---|---|
| Android | w/o new tokens | 443 | 84 | 53 | 125 | 83 | 130 | 107 |
| (Small) | w/ new tokens | 2,064 | 108 | 0 | 206 | 0 | 233 | 0 |
| Google | w/o new tokens | 228 | 11 | 14 | 22 | 36 | 29 | 42 |
| (Small) | w/ new tokens | 907 | 40 | 0 | 81 | 0 | 97 | 0 |
| Ovirt | w/o new tokens | 473 | 73 | 86 | 132 | 173 | 145 | 200 |
| (Small) | w/ new tokens | 2,328 | 352 | 0 | 618 | 0 | 715 | 0 |
| Android | w/o new tokens | 459 | 58 | 32 | 85 | 67 | 89 | 78 |
| (Medium) | w/ new tokens | 2,454 | 124 | 0 | 247 | 0 | 289 | 0 |
| Google | w/o new tokens | 283 | 16 | 9 | 28 | 18 | 33 | 22 |
| (Medium) | w/ new tokens | 1,162 | 18 | 0 | 46 | 0 | 63 | 0 |
| Ovirt | w/o new tokens | 622 | 111 | 18 | 179 | 49 | 199 | 62 |
| (Medium) | w/ new tokens | 3,327 | 415 | 0 | 833 | 0 | 992 | 0 |
| **Total** | w/o new tokens | 2,508 | 353 | 212 | 571 | 426 | 625 | 511 |
| | w/ new tokens | 12,242 | 1,060 | 0 | 2,031 | 0 | 2,389 | 0 |
| | Both | 14,750 | **1,413 (+567%)** | 212 | **2,602 (+511%)** | 426 | **3,014 (+490%)** | 511 |

the abstracted code sequence of the Tufano *et al.* approach) back to the code sequence before matching it with the ground-truth.

In our experiment, we use three different beam widths (i.e., $k = \{1, 5, 10\}$) when generating the after version. Thus, if one of the $k$ sequence candidates exactly matches the ground-truth, we consider that the NMT approach achieves a perfect prediction, i.e., the code is correctly transformed. We do not use other metrics, e.g., BLEU which measures the overlap (or similarity) between the generated and ground-truth sequences, since similarity cannot imply that the generated sequences are viable for code implementation. Ding *et al.* also argue that BLEU should not be used to evaluate the code transformation since the sequences that are similar (i.e., few code tokens are different between the two sequences) may have largely-different intentions or semantics [22].

# 5 RESULTS

## RQ1: Can AutoTransform transform code better than Tufano *et al.* approach?

**Approach.** To address our RQ1, we evaluate how well our AutoTransform can transform the source code of the before version $m_b$ to the after version $m_a$ of given methods (in the testing data), compare against the approach of Tufano *et al.* [60]. Therefore, we build an NMT model for each of the 12 datasets, i.e., small and medium methods with and without new tokens across three repositories (see Table 1). For this RQ, we use the maximum number of merge operations of 2,000 (i.e., BPE2K) in our AutoTransform. In total, we train 96 Transformer models for our AutoTransform (i.e., 12 datasets × 8 hyper-parameter settings); and 120 RNN models for Tufano *et al.* approach (i.e., 12 datasets × 10 hyper-parameter settings). Then, for each approach and for each dataset, we select the model with the best hyper-parameter setting that achieves the lowest loss value computed based on the validation data. Finally, we measure perfect predictions based on the testing data. To quantify the magnitude of improvement for our AutoTransform, we compute a percentage improvement of perfect predictions using a calculation of $\frac{(\#PP_{\text{our}} - \#PP_{\text{Tufano}}) \times 100\%}{\#PP_{\text{Tufano}}}$.

**Results.** Table 4 shows the results of perfect predictions of our AutoTransform and Tufano *et al.* approach of 14,750 changed methods across the 12 datasets. The results are based on three beam widths, $k = \{1, 5, 10\}$, where $k$ is the number of sequence candidates for a given method.

**When considering both change types (i.e., w/o and w/ new tokens), our AutoTransform achieves a perfect prediction for 1,413 methods which is 567% higher than the perfect predictions achieved by the Tufano *et al.* approach.** Table 4 shows that when a beam width is 1 and both change types are considered, our AutoTransform achieves a perfect prediction for 34 - 526 methods which accounted for 2% ($\frac{34}{1,445}$; for Google Medium) - 13% ($\frac{526}{3,949}$; for Ovirt Medium) of the methods in testing data. On the other hand, Tufano *et al.* approach achieves a perfect prediction for 9 - 86 methods which accounted for only 0.62% ($\frac{9}{1,445}$; for Google Medium) - 3% ($\frac{86}{2,801}$; for Ovirt Small). In total across the 12 datasets, our AutoTransform can correctly transform 1,413 methods, which is 567% higher than the perfect predictions achieved by Tufano *et al.* approach.

Even when we increase the beam width to 5 and 10, our AutoTransform can achieve higher perfect predictions, i.e., 5% ($\frac{74}{1,445}$ for Google Medium) - 28% ($\frac{1,102}{3,949}$ for Ovirt Medium) at beam width = 5 and 7% ($\frac{96}{1,445}$ for Google Medium) - 30% ($\frac{1,191}{3,949}$ for Ovirt Medium) at beam width = 10. The perfect predictions are improved by 511% for the beam width of 5 (and 490% for the beam width of 10) when compared to the perfect predictions achieved by Tufano *et al.* approach. These results indicate that the number of methods for which our AutoTransform can achieve a perfect prediction is substantially higher than Tufano *et al.* approach.

**For the changed methods *with* new tokens appearing in the after version, our AutoTransform achieve a perfect prediction for 18 - 415 methods.** At beam width is 1, Table 4 shows

that our AutoTransform achieves a perfect prediction for 18 - 415 methods which accounted for 2% ($\frac{18}{1,162}$) - 12% ($\frac{415}{3,327}$) of the changed methods *with* new tokens in the testing data. In total, our AutoTransform achieves a perfect prediction for 1,060 methods. Similarly, our AutoTransform achieves higher perfect predictions when the beam width is increased to 5 and 10, i.e., 4% ($\frac{46}{1,162}$) - 25% ($\frac{833}{3,327}$) at the beam width of 5; and 5% ($\frac{63}{1,162}$) - 30% ($\frac{992}{3,327}$) at the beam width of 10. On the other hand, Table 4 shows that Tufano *et al.* approach could not achieve a perfect prediction for any of the changed methods with new tokens appearing in the after version. This is because the IDs of the new tokens cannot be mapped back the actual identifiers/literals as they did not exist in the before version (see Limitation 1 in Section 2). These results highlight that our AutoTransform can transform source code even when new tokens appear in the after version.

**For the changed methods *without* new tokens in the after version, our AutoTransform achieves a perfect prediction for 11 - 111 methods, while Tufano *et al.* approach achieves a perfect prediction for 9 - 86 methods.** At the beam width of 1, Table 4 shows that our AutoTransform achieves a perfect prediction for 11 - 111 methods which accounted for 5% ($\frac{11}{228}$) - 18% ($\frac{111}{622}$) of the changed methods *without* new tokens in the testing data. On the other hand, Tufano *et al.* approach achieves a perfect prediction for 9 - 86 methods which accounted for 3% ($\frac{9}{283}$) - 18% ($\frac{86}{473}$). For the small methods, our AutoTransform achieves more perfect predictions in the Android dataset, but fewer in the Google and Ovirt datasets. We will further discuss this result in Section 6. Nevertheless, it is worth noting that for the medium methods, our AutoTransform achieves perfect predictions 78% ($\frac{16-9}{9}$) - 517% ($\frac{111-18}{18}$) higher than the perfect predictions achieved by Tufano *et al.* approach. The results are similar when the beam width is 5 and 10. These results suggest that when a sequence becomes longer, our AutoTransform transforms code better than the Tufano *et al.* approach, highlighting that our AutoTransform can address Limitation 2 discussed in Section 2.

## RQ2: What are the contributions of AutoTransform's components?

**Approach.** To address our RQ2, we examine perfect predictions when a component in our AutoTransform is varied. Specifically, we examine the percentage difference of perfect predictions when subword tokenization is changed to code abstraction (BPE → Abs); and when the NMT architecture is changed from Transformer to RNN (Transformer → RNN). We also investigate the case when the maximum number of merge operations is changed from 2,000 to 5,000 (BPE2K → BPE5K), i.e., the vocabulary size increases. Thus, we evaluate the perfect predictions of four additional combinations: BPE5K+Transformer, Abs+Transformer, BPE2K+RNN, BPE5K+RNN. We build an NMT model using each combination for each of the 12 datasets. In total, for RQ2, we further build 192 Transformer models (i.e., 2 Transformer-based combinations × 12 datasets × 8 hyper-parameter settings); and 240 RNN models (i.e., 2 RNN-based combinations × 12 datasets × 10 hyper-parameter settings). Similar to RQ1, we select the model with the best hyper-parameter setting based on the validation data; and measure perfect predictions based on the testing data.

**Results.** Figure 3 shows the perfect predictions of our AutoTransform (BPE2K+Transformer), Tufano *et al.* approach (Abs+RNN), and the four additional combinations.

**Using subword tokenization by BPE can increase perfect predictions at least by 284%, compared to the code abstraction with reusable IDs.** Figure 3 shows that at beam width of 1, the perfect predictions of our AutoTransform (BPE2K+Transformer) is 284% ($\frac{192-50}{50}$ for Android Small) - 2,290% ($\frac{526-22}{22}$ for Ovirt Medium) higher than Abs+Transformer. Considering the cases when the RNN architecture is used with BPE (i.e., BPE2K+RNN and BPE5K+RNN), the perfect predictions are also higher than Tufano *et al.* approach (Abs+RNN). For example, for the small methods, Figure 3 shows that BPE2K+RNN achieves perfect predictions 29% ($\frac{18-14}{14}$) - 323% ($\frac{364-86}{86}$) higher than Tufano *et al.* approach. Figure 3 also shows similar results when the beam width was increased to 5 and 10. These results indicate that regardless of the NMT architecture, subword tokenization by BPE largely contributes to the performance in transforming code of our AutoTransform.

When using a different number of merge operations (BPE2K → BPE5K), we find that perfect predictions were slightly different. For example, for the small methods (at beam width of 1), the percentage difference of perfect prediction between our AutoTransform and BPE5K+Transformer is 7% ($\frac{55-51}{51}$) - 19% ($\frac{192-156}{192}$). Figure 3 also shows that the results are similar for the RNN-based approaches (i.e., BPE2K+RNN and BPE5K+RNN). These results suggest that the number of merge operations has an impact (but relatively small) on the performance of our AutoTransform.

**Using the Transformer architecture can increase perfect predictions at least by 17%, compared to the RNN architecture.** Figure 3 shows that at the beam width of 1 and for small methods, our AutoTransform achieves perfect predictions 17% ($\frac{425-364}{364}$ for Ovirt) - 183% ($\frac{51-18}{18}$ for Google) higher than BPE2K+RNN. It is also worth noting that the percentage difference is much higher for the medium methods, i.e., 183% ($\frac{34-12}{12}$ for Google) - 507% ($\frac{182-30}{30}$ for Android). Figure 3 also shows a large difference of perfect predictions between our AutoTransform and BPE2K+RNN when the beam width is increased to 5 and 10. The results are also similar when comparing the perfect predictions between BPE5K+Transformer and BPE5K+RNN, i.e., the Transfomer models tend to achieve higher perfect predictions than the RNN models. These results suggest that the Transformer architecture also contributes to the performance of our AutoTransform in transforming code, especially for the methods with a relatively long sequence like medium methods.

## 6 DISCUSSION

In this section, we discuss our AutoTransform in several aspects including its advantage, performance, and practicality.

**Advantage:** *Why does our AutoTransform work for the changed methods with new tokens?* Table 4 shows that in total, our AutoTransform can correctly transform the methods that have new tokens appearing in the after version. We further analyze these methods to better understand the characteristics of the new code

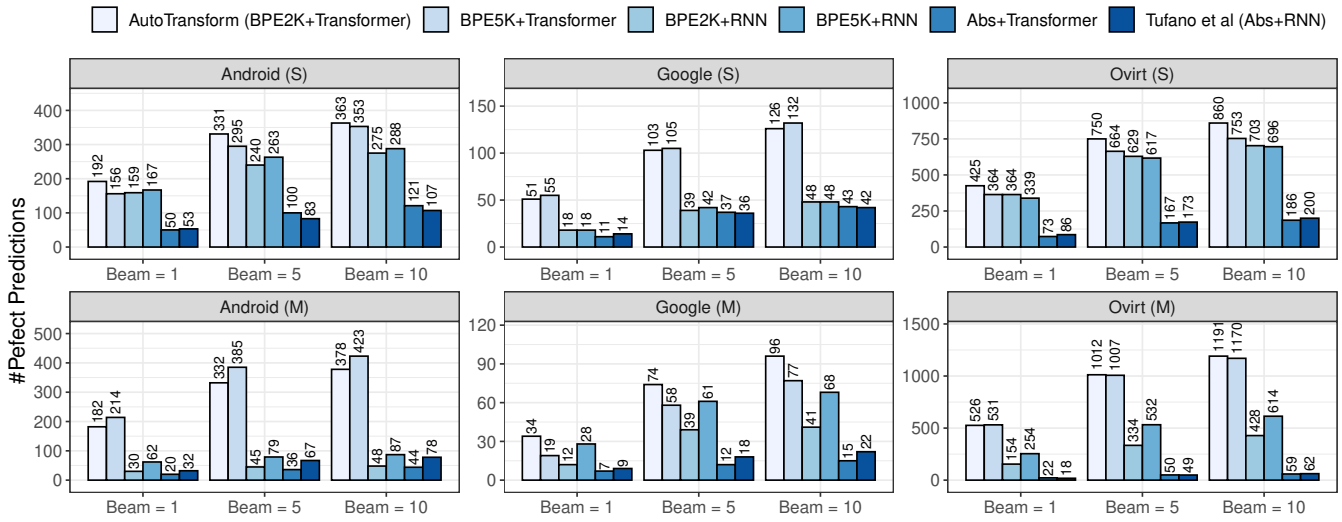


**Figure 3: The perfect prediction of our AutoTransform when a component is varied. The y-axis shows the total number of perfect predictions of changed methods *with* and *without* new tokens.**

tokens appearing in the after version. We find that 43% ($\frac{960}{1,689}$) of the new code tokens are the identifiers/literals that already exist in the training data (i.e., *known* code tokens), suggesting that our AutoTransform can reuse the code tokens existing across all methods that AutoTransform have learnt. On the other hand, the Tufano *et al.* approach cannot generate these new code tokens because their approach is restricted by the code abstraction with reusable IDs to use only the identifiers/literals that exist in the before version; or that are the top-300 frequent identifiers/literals.

Furthermore, the other 57% of the new code tokens appearing in the after version are new identifiers/literals that do not exist in the training data, suggesting that our AutoTransform can generate these new code tokens based on a combination of known subwords in the training data. We observe that these new code tokens are related to changing a Java package/library (e.g., `org.junit.Assert` → `org.hamcrest.MatcherAssert`), changing identifier (e.g., `getlog_type_name` → `getLogTypeName`, `d` → `dt.mID`), or even adding new statements (e.g., instantiating a new object).

**Limitation:** *Why does our AutoTransform achieve fewer perfect predictions than Tufano et al. approach for small methods without new tokens in the Google and Ovirt datasets?* Table 4 shows that, for the small methods without new tokens in the after version, our AutoTransform achieves fewer perfect predictions than the Tufano *et al.* approach in the Google and Ovirt datasets. To better understand the methods for which our AutoTransform cannot achieve a perfect prediction, we manually examine 9 of the 14 methods (Google) and 54 of the 86 methods (Ovirt) for which our AutoTransform cannot achieve a perfect prediction but the Tufano *et al.* approach could. We find that there are only 2 (out of 9) and 5 (out of 54) methods that are incorrectly predicted by our AutoTransform.

On the other hand, for the remaining 7 (out of 9) and 49 (out of 54) methods, we find that our AutoTransform almost achieves a perfect prediction, i.e., the generated sequence is very similar to the ground-truth with minor errors. Broadly speaking, we observe that for most of these methods, our approach transforms some code tokens that should remain unchanged in the after version. One possible reason is that these code tokens are a rare token and BPE splits them into many subwords, i.e., one rare token becomes a long subword sequence. Then, due to the large search space of subwords, our approach may unnecessarily generate a new combination of subwords. For example, we observe that `FixturesTool.DATA_CENTER` is split into 9 subwords (i.e., `FixturesTool.@@, DA@@, ..., TE@@, R`). Then, our approach inserts a subword `A@@`, resulting in a new token `FixturesTool.ADATA_CENTER`. Based on this observation, an approach to reduce the length of subword sequences (e.g., fine-tuning the number of merge operations) may improve the performance.

**Hyper-Parameter Sensitivity:** *How sensitive the hyper-parameter setting is in our AutoTransform?* Deep learning models are known for being sensitive to hyper-parameter settings. While we select the best hyper-parameter setting based on the validation data for our experiment, we are interested in examining the impact of hyper-parameter settings on the performance of our AutoTransform. Hence, we analyze the perfect predictions of our AutoTransform when each of the eight hyper-parameter settings in Table 3 is used. We find that a setting of ($N_e$ = 2, $N_d$ = 4, $h$ = 8, $hs$ = 512,) allows our AutoTransform achieves the highest perfect predictions for 7 out of 12 datasets, while a setting of ($N_e$ = 1, $N_d$ = 2, $h$ = 8, $hs$ = 512) allows our AutoTransform to achieve the highest perfect predictions for 4 out of 12 datasets. Nevertheless, we observe that the perfect predictions is decreased by only 1 - 3 percentage points when using the other hyper-parameter settings instead of the best setting.

**Performance:** *How long does our AutoTransform take to train and infer?* Model training and inference time can be one of the important factors when considering the adoption of our approach in practice. Hence, we measure the training and inference time for

the Transformer models in our AutoTransform. We find that the training time for each Transformer model in our AutoTransform used in RQ1 is ranging from 30 minutes to 2 hours depending on the size of the datasets and the number of epochs. The average inference time of AutoTransform per method is ranging from 15 to 60 milliseconds for generating one sequence candidate (i.e., Beam width = 1). Similarly, when generating 5 and 10 sequence candidates per method (i.e., Beam width = {5, 10}), the average inference time per input sequence is ranging from 42 to 200 milliseconds. Note that the training and inference time is based on a computer with an Nvidia GeForce RTX 3090 graphic card.

**Practicality:** *To what extent AutoTransform can support the modern code review process?* Our RQ1 has shown that AutoTransform can correctly transform 1,413 methods, which is substantially better than the prior work. This result highlights a great potential of AutoTransform to augment code authors by recommending the common revisions that occurred during the code reviews in the past to apply to the newly-written code without waiting for reviewers' feedback. Nevertheless, at this stage of the work, AutoTransform may be suitable for code changes of software components that tolerate false positives as AutoTransform will provide a recommendation for every changed method which is subject to produce many false positives compared to humans. Indeed, prior work [51] reported that developers may decide to not use a supporting tool if its false positive rate is above 25%. Furthermore, similar to the prior work [60], the applicability of AutoTransform is still limited to small and medium method sizes. Thus, to broaden the practicality of AutoTransform, future work should aim to develop an approach that is more selective to achieve higher accuracy (e.g., below 25% of false positive rate) for any method sizes (including large methods).

## 7 RELATED WORK

**Automated Code Review.** Code review is effective, but still human-intensive and time-consuming [11, 37, 50]. Thus, recent work leveraged machine learning techniques to support various activities throughout the code review process, for example, reviewer recommendation [13, 43, 46, 49, 56, 59], review task prioritization based on code change characteristics [23, 38, 58] and defect-proneness [30, 39, 44, 45, 65]. Several studies also proposed approaches to support reviewers when reading and examining code [14, 27, 55, 63, 64]. Although these approaches can reduce the manual effort of reviewers, code authors still need to manually modify the source code until it is approved by reviewers. Yet, few studies focus on developing an approach to automatically transform source code to help code authors reduce their effort in the code review context.

To the best of our knowledge, only two recent approaches [60, 61] are proposed to automatically transform the source code to the version that is approved by reviewers (i.e., the after version). However, both recent approaches use code abstraction (ABS+ RNN [60], ABS+Transformer [61]) to reduce the vocabulary size, where our results show that code abstraction hinders the NMT approaches in correctly transforming code if the after version has a new token (e.g., renamed variables). Thus, these recent approaches still have a limited usage to automatically transform code in the code review process. Different from prior work, we leverage BPE to address this limitation of prior work. Importantly, our RQ2 shows that using BPE (BPE+Transformer, BPE+RNN) achieves perfect predictions at least by 284% higher than using the code abstraction with reusable IDs (ABS+Transformer, ABS+RNN), highlighting the important contribution of this paper to the automated code transformation for code review.

**Neural Machine Translation (NMT) in Software Engineering.** NMT approaches have been developed to support various software engineering tasks, which can be categorized into four types of transformation: (1) Text→Text (e.g., language translation of code documentation [35], query reformulation [19]); (2) Text→Code (e.g., code search [24, 42]); (3) Code→Text (e.g., code summarization [25], commit message generation [29, 34]); and (4) Code→Code (e.g., automated program repair [20, 28, 33], programming language translation [48], code completion [54]). Although automated program repair (APR) approaches and our AutoTransform share a similar concept of using NMT for a Code→Code task, APR approaches [20, 28, 33] only aim to automatically transform buggy code to clean code for bug-fixing purposes, which may not be related to other types of code changes in code review. Different from APR, our AutoTransform aims to automatically transform source code that is changed during the code review process (e.g., refactoring) to improve readability, maintainability, and design quality [16, 41, 50, 57].

## 8 THREATS TO VALIDITY

**Construct Validity.** The source code granularity used in this work is at the method level. The results may be varied if the changed source code is extracted at a different granularity (e.g., changed lines). However, prior work pointed out that an NMT model requires code context and using only changed lines may lead the NMT model to suffer from the Out-Of-Vocabulary problem, even though BPE is used [22]. Hence, we believe that training an NMT model at a method level would provide a reasonable range of code context than changed lines.

We define the method size based on the number of tokens (i.e., 1 - 50 tokens for small and 51 - 100 tokens for medium). This size definition may not reflect the actual method sizes in practices, e.g., a method with 100 tokens may actually have few lines of code. The performance of AutoTransform may differ if other definitions of method size are used. Nevertheless, for the sake of fair comparison in our experiment, we opt to use the same definition of method size as used in the prior work [60]. Moreover, from the aspect of the NMT algorithm, transforming long sequences are not trivial as it requires higher memory and computation power [66].

**Internal Validity.** We experiment with only two settings of merge operations (i.e., BPE2K and BPE5K); and 8 hyper-parameter settings which are based on a combinations of four hyper-parameters. The results may be varied if the number of merge operations and the hyper-parameter settings of both approaches are optimized. However, finding an optimal setting can be very computationally expensive given a large search space of the number of merge operations and all available hyper-parameters. In addition, the goal of our work is not to find the best setting, but to fairly compare the performance of our approach with the prior approach based on similar settings as used in the prior work [60]. Nevertheless, our analyses in Sections 5 and 6 have shown that the number of merge

operations and hyper-parameter settings have a small impact on the performance of our approach.

**External Validity.** We evaluate our AutoTransform based on the changed methods that were extracted from three Gerrit code repositories. In addition, we only experimented with Java programs. Our results may not be generalized to other code review repositories or other programming languages. However, our approach is based on the techniques (i.e., BPE and Transformer) that are language-independent. In addition, we provided a replication package to facilitate future work to replicate our approach on different repositories and programming languages.

## 9 CONCLUSION

Prior work [60] proposed an NMT approach to automatically transform a given method from the before version (i.e., the version before the implementation of code changes) to the after version (i.e., the version that is reviewed and merged). Yet, its performance is still suboptiomal when the after version has new identifiers or literals, or when the sequence becomes longer. Hence, in this paper, we propose AutoTransform which leverages BPE to handle new tokens and a Transformer to handle long sequences.

Through an empirical evaluation based on 14,750 changed methods that are extracted from three Gerrit code review repositories, we find that (1) our AutoTransform can correctly transform 1,060 methods of which the after version has new tokens while the prior work can not correctly transform any of these methods; and (2) for the changed methods of which the after version do not have new tokens, our AutoTransform can correctly transform 353 methods, which is 67% higher than the prior work. Furthermore, our ablation study also shows that BPE and Transformer substantially contribute to the performance improvement of our AutoTransform, when compared to the components used in the prior work. These results highlight that our AutoTransform effectively address the limitations of the prior work, allowing the NMT approach to be applied to a wider range of changed methods (i.e., methods with new tokens and methods with long sequences). The proposed approach and the results of this paper contribute toward automated code transformation for code reviews, which could help developers to reduce their effort on modifying source code during the code review process.

## ACKOWLEDGEMENT

Patanamon Thongtanunam was supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE210101091). Chakkrit Tantithamthavorn was supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100941).

## REFERENCES

[1] [n.d.]. A library for subwod tokenization using Byte-Pair Encoding. https://github.com/rsennrich/subword-nmt.
[2] [n.d.]. A list of Top-300 frequent identifier/literals for each of the studied datasets. https://sites.google.com/view/learning-codechanges/data#h.p_r-R_Z4sKJC2L.
[3] [n.d.]. Android's Gerrit Code Review Repositories. https://android-review.googlesource.com/.
[4] [n.d.]. AutoTransform's Replication Package. https://github.com/awsm-research/AutoTransform-Replication.
[5] [n.d.]. Datasets of the paper titled "On Learning Meaningful Code Changes Via Neural Machine Translation". https://sites.google.com/view/learning-codechanges/data#h.p__6KdV38lN05N.
[6] [n.d.]. Google's Gerrit Code Review Repositories. https://gerrit-review.googlesource.com/.
[7] [n.d.]. Ovirt's Gerrit Code Review Repositories. https://gerrit.ovirt.org/.
[8] [n.d.]. Seq2Seq: A library for RNN-based NMT models. https://google.github.io/seq2seq/.
[9] [n.d.]. Src2Abs: A library for abstracting code with reusable IDs. https://github.com/micheletufano/src2abs.
[10] [n.d.]. Tensor2Tensor: A library for Transfomer-based NMT models. https://github.com/tensorflow/tensor2tensor.
[11] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of ICSE*. 712–721.
[12] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of ICLR*. 1–15.
[13] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of ICSE*. 931–940.
[14] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2019. Associating Working Memory Capacity and Code Change Ordering with Code Review Performance. *EMSE* 24, 4 (2019), 1762–1798.
[15] Gabriele Bavota and Barbara Russo. 2015. Four Eyes Are Better Than Two: On The Impact of Code Reviews on Software Quality. In *Proceedings of ICSME*. 81–90.
[16] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern Code Reviews in Open-Source Projects: Which Problems do They Fix?. In *Proceedings of MSR*. 202–211.
[17] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *Proceedings of MSR*. 146–156.
[18] Denny Britz, Anna Goldie, Minh Thang Luong, and Quoc V. Le. 2017. Massive Exploration of Neural Machine Translation Architectures. In *Proceedings of EMNLP*. 1442–1451.
[19] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. 2021. Automated Query Reformulation for Efficient Search based on Query Logs From Stack Overflow. In *Proceedings of ICSE*. 1273–1285.
[20] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-Sequence Learning for End-to-End Program Repair. *TSE* (2019).
[21] Czerwonka, Jacek and Greiler, Michaela and Tilford, Jack. 2015. Code Reviews Do Not Find Bugs How the Current Code Review Best Practice Slows Us Down. In *Proceedings of ICSE*. 27–28.
[22] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. 2020. Patching as Translation: The Data and the Metaphor. In *Proceedings of ASE*. 275–286.
[23] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. 2018. Early Prediction of Merged Code Changes to Prioritize Reviewing Tasks. *EMSE* 23, 6 (2018), 3346–3393.
[24] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of ICSE*. 933–944.
[25] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved Automatic Summarization of Subroutines via Attention to File Context. In *Proceedings of MSR*. 300–310.
[26] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proceedings of FSE*. 763–773.
[27] Yang Hong, Chakkrit Tantithamthavorn, and Patanamon Thongtanunam. 2022. Where Should I Look at? Recommending Lines that Reviewers Should Pay Attention To. In *Proceeding of SANER*.
[28] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of ICSE*. 1161–1173.
[29] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically Generating Commit Messages from Diffs using Neural Machine Translation. In *Proceedings of ASE)*. 135–146.
[30] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *TSE* 39, 6 (2013), 757–773.
[31] Rafael Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of ICSE)*. 1073–1085.
[32] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code Review Quality: How Developers See It. In *Proceedings of ICSE*. 1028–1038.
[33] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based Code Transformation Learning for Automated Program Repair. In *Proceedings of ICSE*. 602–614.
[34] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-Machine-Translation-based Commit Message Generation: How Far are We?. In *Proceedings of ASE*. 373–384.

[35] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021).
[36] Minh Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of EMNLP*. 1412–1421.
[37] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2017. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software* 35, 4 (2017), 34–42.
[38] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. 2019. Predicting Pull Request Completion Time: A Case Study on Large Scale Cloud Services. In *Proceedings of ESEC/FSE*. 874–882.
[39] Shane McIntosh and Yasutaka Kamei. 2017. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *TSE* (2017), 412–428.
[40] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *EMSE* 21, 5 (2016), 2146–2189.
[41] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do Code Review Practices Impact Design Quality?: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of SANER*. 171–180.
[42] Thanh Nguyen, Peter C Rigby, Anh Tuan Nguyen, Mark Karanfil, and Tien N Nguyen. 2016. T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation. In *Proceedings of FSE*. 1013–1017.
[43] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-based Peer Reviewers Recommendation in Modern Code Review. In *Proceedings of ICSME*. 367–377.
[44] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *Proceedings of MSR*. To Appear.
[45] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2022. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* (2022).
[46] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. 2016. CORRECT: Code Reviewer Recommendation in GitHub based on Cross-Project and Technology Experience. In *Proceedings of ICSE (Companion)*. 222–231.
[47] Peter C Rigby and Christian Bird. 2013. Convergent Contemporary Software Peer Review Practices. In *Proceedings of FSE*. 202–212.
[48] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages.. In *NeurIPS*.
[49] Shade Ruangwan, Patanamon Thongtanunam, Akinori Ihara, and Kenichi Matsumoto. 2018. The Impact of Human Factors on the Participation Decision of Reviewers in Modern Code Review. *EMSE* (2018), In press.
[50] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of ICSE (Companion)*. 181–190.
[51] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *Proceeding of the International Conference on Software Engineering (ICSE)*. 598–608.
[52] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of ACL*, Vol. 3. 1715–1725.
[53] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NIPS*. 3104–3112.
[54] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. In *Proceedings of ESEC/FSE*.
[55] Yida Tao and Sunghun Kim. 2015. Partitioning Composite Code Changes to Facilitate Code Review. In *Proceedings of MSR*. 180–190.
[56] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. 2014. Improving code review effectiveness through reviewer recommendations. In *Proceedings of CHASE*. 119–122.
[57] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2015. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *MSR*. 168–179.
[58] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2017. Review Participation in Modern Code Review. *EMSE* 22, 2 (2017), 768–817.
[59] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who Should Review My Code? A File Location-based Code-reviewer Recommendation Approach for Modern Code Review. In *Proceedings of SANER*. 141–150.
[60] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes Via Neural Machine Translation. In *Proceedings of ICSE*. 25–36.
[61] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanykz, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *Proceedings of ICSE*. To appear.
[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *NIPS*. 5999–6009.
[63] Dong Wang, Tao Xiao, Patanamon Thongtanunam, Raula Gaikovina Kula, and Kenichi Matsumoto. 2021. Understanding Shared Links and Their Intentions to Meet Information Needs in Modern Code Review. In *EMSE*. to appear.
[64] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. CORA: Decomposing and Describing Tangled Code Changes for Reviewer. In *Proceedings of ASE*. 1050–1061.
[65] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* (2020).
[66] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 17283–17297.