Assessing the Latent Automated Program Repair Capabilities of Large Language Models using Round-Trip Translation

FERNANDO VALLECILLOS RUIZ, Simula Research Laboratory, Norway
ANASTASIIA GRISHINA, Simula Research Laboratory, Norway
MAX HORT, Simula Research Laboratory, Norway
LEON MOONEN*, Simula Research Laboratory, Norway

Research shows that errors in natural language can be corrected by translating texts to another language and back using language models. We explore to what extent this *latent* correction capability extends to Automated Program Repair (APR) by investigating Round-Trip Translation (RTT): translating code from one programming language into another programming or natural language and back, using Large Language Models (LLMs). We hypothesize that RTT restores patterns most commonly seen in the LLM's training corpora through *regression toward the mean*, replacing infrequent bugs with more frequent, *natural*, bug-free code. To test this hypothesis, we employ nine LLMs and four common APR benchmarks in Java, and perform a detailed quantitative and qualitative analysis of RTT-generated patches. We find that RTT through English generates plausible patches for 100 of 164 bugs with GPT-4 on the HumanEval-Java benchmark, and 97 are found to be correct in our manual assessment. Moreover, RTT uniquely generates plausible patches for 46 bugs that were missed by LLMs specifically fine-tuned for APR. While this demonstrates the viability of RTT for APR, we also observe limitations, such as a lower overall bug fix rate than the state-of-the-art and diluting the original coding style. We analyze the impact of these limitations and discuss the potential of using RTT as a complementary component in APR frameworks.

CCS Concepts: • **Software and its engineering → Correctness**; **Automatic programming**; **Software testing and debugging**.

Additional Key Words and Phrases: automated program repair, large language model, machine translation

1 INTRODUCTION

As software becomes ubiquitous and more people engage in software engineering (SE) tasks, the need to ensure its reliability and integrity increases. In the meantime, code maintenance and refactoring are tedious tasks that take a significant amount of developers' time and impede their progress in building new features [9, 44]. Automated program repair (APR) aims to fix errors in source code with minimal human involvement, thus reducing code maintenance needs and releasing resources for creative code writing. With the advent of language models trained on source code, learning-based methods that use generative and translation models to fix bugs have started to compete with traditional heuristic and constraint-based approaches for APR [34, 43].

Large Language Models (LLMs) trained on vast amounts of natural language and source code have pushed many fields away from traditional techniques and common heuristics, including the field of software engineering.

* Corresponding author.

Authors' Contact Information: Fernando Vallecillos Ruiz, fernando@simula.no, Simula Research Laboratory, Oslo, Norway; Anastasiia Grishina, anastasiia@simula.no, Simula Research Laboratory, Oslo, Norway; Max Hort, maxh@simula.no, Simula Research Laboratory, Oslo, Norway; Leon Moonen, leon.moonen@computer.org, Simula Research Laboratory, Oslo, Norway.

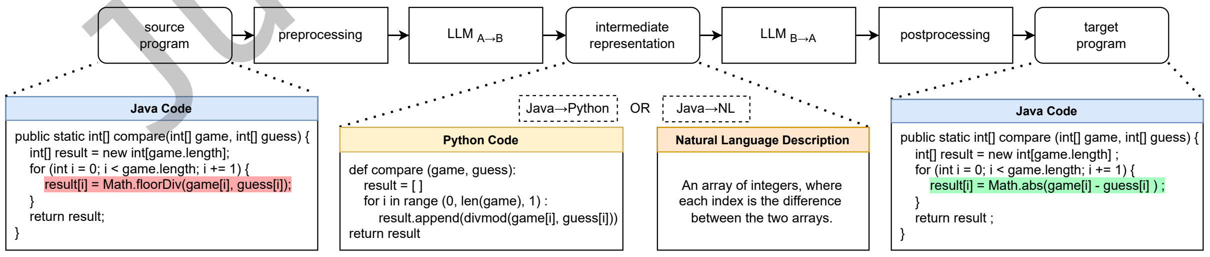
ACM 1557-7392/2025/10-ART
https://doi.org/10.1145/3771922

Nowadays, LLMs are able to generate code, create documentation, and locate bugs [13]. They have also been empirically proven to possess latent capabilities. In other words, they are capable of solving tasks that they were not specifically pre-trained or fine-tuned on [11].

Inspired by the use of translation to find and correct errors in natural language [16], we explore to what extent LLMs trained for understanding and generating code have the latent capability to find and correct errors in code. Specifically, we investigate an LLM's capability to debug code through a process known as *round-trip translation* (RTT). RTT was previously used to evaluate machine translation of natural languages, and more recently, to evaluate the correctness of code generation [7]. Unlike more traditional LLM-based APR approaches that center on one-shot bug-to-fix translations, RTT consists of two steps: the translation of buggy code into an intermediate language, and then the translation back to its original language. Thereby, RTT may offer a novel approach to repairing coding errors based on the latent error correcting capabilities of LLMs, an area previously overlooked by other learning-based APR techniques.

Our hypothesis is that RTT may be capable of fixing bugs as a result of a *regression toward the mean* phenomenon exhibited by generative language models trained on vast code corpora. Studies show that frequent code patterns in such large code corpora are bug-free [54]. Thus, as a result of training LLMs on these corpora, RTT will regress toward the same mean of bug-free code, in other words, demonstrate the latent capability of producing code without errors. To empirically investigate this hypothesis and assess the viability of RTT for APR, we conduct the first comprehensive study on RTT with LLMs for APR. Our experiments use nine LLMs, including three GPT versions, and four APR benchmarks. The models vary in size, ranging from 140M parameters to recent OpenAI models with undisclosed sizes that have been estimated in the trillion-parameter range. Moreover, they vary in original training objectives, spanning from code and docstring infilling to code summarization, translation, and generation. The benchmarks contain code with different context size and bug complexity, ranging from student assignments in QuixBugs [37] and HumanEval-Java [25] to real-world projects in Defects4J v1.2 and v2.0 [27]. Note that HumanEval-Java was not available during training of most the LLMs used in this study, mitigating bias from data leakage on the results obtained with this benchmark.

In this paper, we utilize an RTT pipeline, shown in Figure 1, to systematically explore the latent program repair capabilities of LLMs trained on coding tasks other than APR. We use LLMs initially trained or fine-tuned for code translation, summarization, and generation, to translate buggy code from one language to another and back, investigating whether this process implicitly fixes the bugs. We assess how effectively this approach eliminates bugs under various conditions. The pipeline uses either a programming language (PL) or a natural language (NL) as intermediate representation. Moreover, our RTT pipeline uses the LLMs in a zero-shot fashion: unlike other neural APR methods, with RTT we do not need to fine-tune models on the bug repair task, but apply them *off-the-shelf*, as provided by the model authors. Our investigation aims to understand the characteristics of



```
public static int[] compare(int[] game, int[] guess) {
    int[] result = new int[game.length];
    for (int i = 0; i < game.length; i += 1) {
        result[i] = Math.floorDiv(game[i], guess[i]);
    }
    return result;
}
```

```
def compare (game, guess):
    result = [ ]
    for i in range (0, len(game), 1) :
        result.append(divmod(game[i], guess[i]))
return result
```

An array of integers, where each index is the difference between the two arrays.

```
public static int[] compare (int[] game, int[] guess) {
    int[] result = new int[game.length] ;
    for (int i = 0; i < game.length; i += 1) {
        result[i] = Math.abs(game[i] - guess[i] ) ;
    }
    return result ;
}
```

Fig. 1. High-level overview of the RTT process with concrete examples taken from our empirical evaluation. The red highlight on the left indicates the buggy line, the green highlight on the right is the repaired line.

RTT-based repair, including its potential strengths and weaknesses compared to direct bug-fixing approaches. A key aspect of our analysis involves determining if RTT can address bugs that are missed by other methods, which could inform its potential role within the broader APR landscape. Our findings suggest the potential of RTT as a complementary tool in a collaborative LLM-based APR framework, where different models and their pipelines generate and update code.

Contributions: The main contributions of this work are as follows:

- ★ We present the first systematic empirical evaluation of RTT with LLMs for APR, an area previously overlooked by other APR approaches;
- ★ We asess RTTs effectiveness across nine language models (six open-source models and three API-based), four common APR benchmarks with various context sizes and bug types, and 10 different seeds;
- ★ We compare the effectiveness of using another programming language versus using natural language as the intermediate representation in the RTT pipeline;
- ★ We analyze the trade-offs between LLM size, temperature, and repair performance;
- ★ Our findings include that RTT generates plausible patches for 100 of 164 bugs in the HumanEval-Java benchmark, and 97 were found to be correct in our manual assessment. Moreover, over all four benchmarks, RTT generates plausible patches for 46 bugs that were not fixed by other methods, even those fine-tuned on the APR task. Note that the evaluation of patches for QuixBugs and Defects4J only considered patch plausibility, which has inherent limitations, such as ignoring potential overfitting to test code;
- ★ We provide an in-depth qualitative analysis of RTT-generated patches for QuixBugs and HumanEval-Java, identifying common repair patterns, characteristics, and underlying challenges associated with the approach;
- ★ We conclude by discussing the benefits, limitations, and potential role of RTT in the APR landscape, based on our empirical findings;
- ★ We release the code for RTT and results obtained to ensure replication and verification of our work, including the manual assessment of patch correctness on over 5,000 RTT-generated patches for HumanEval-Java.[1]

2 BACKGROUND AND RELATED WORK

2.1 Neural Machine Translation

Traditional methods for translating text from one language into another are increasingly replaced by Neural Machine Translation (NMT), where neural networks are used to predict a sequence of translated words [62, 76] and sequence-to-sequence methods, in general [63]. Sequence-to-sequence models enable NMT to automatically learn complex mappings between different languages, efficiently capturing context and offering more accurate translations in comparison to their predecessors.

To generate translated sentences, autoregressive NMT methods use the whole source sentence and the initial part, or *prefix*, of the target sentence. Assuming $x = \{x_1, ..., x_n\}$ is the source sentence split into n tokens, $y = \{y_1, ..., y_m\}$ is the target sentence and $y_{<i} = \{y_1, ..., y_{i-1} | \ i \leq m\}$ is the beginning of the target sequence generated up to token $i - 1$, we can formulate the generation as a conditional probability $P(y|x)$:

$$P(y|x) = \prod_{i=1}^{m} P(y_i|x, y_{<i}). \tag{1}$$

Notably, NMT methods are applied to translate between both natural and programming languages. The predominant model architecture is encoder-decoder or decoder-only transformer [66]. A widely used approach to create an NMT model for PL-PL translation is to fine-tune a model pre-trained on code and, possibly, natural

[1] The replication package is available for download from Zenodo: https://doi.org/10.5281/zenodo.10500593.

language on a dataset with pairs of code snippets in the source and target PLs [2, 19, 68]. Similarly, GPT models by OpenAI are sequence-to-sequence models trained to create an output text or code sequence (response) from an input sequence (prompt), the latest models benefiting from Reinforcement Learning with Human Feedback (RLHF) and other training advancements as well as large amounts of pre-training data [10, 46, 52]. Overall, the incorporation of deep learning-based NLP techniques left a distinct mark on the field of data-driven software engineering methods by allowing it to leverage statistical properties such as code naturalness.

2.2 Software Naturalness

Software naturalness is the property of source code to exhibit patterns and follow conventions that are statistically similar to other forms of human expression, such as natural language [6, 23]. Due to the repetitive nature of coding patterns and the natural incline of software developers to create simple blocks of code for readability and maintainability, programming languages have learnable statistical properties [23]. This means that NLP techniques that rely on learning patterns in input sequences, such as NMT, can be applied to source code. Neural Program Translation (NPT) applies NMT to understand the underlying logic and semantics of the source code and generates functionally equivalent programs in the target language [56]. Ray et al. [54] observed bugs to be deviations that manifest as unnatural *noise* which increases entropy in otherwise predictable and repetitive natural code. This observation has been used to address various tasks, such as APR [65], vulnerability identification [12], and patch ranking [28, 31].

2.3 Language Models for Automated Program Repair

Interpreting APR as a translation from buggy to fixed sequences further stimulated the use of language models [15, 26]. Early models used RNN and LSTM architectures that cannot handle long-range dependencies and scale poorly [79]. Transformers [66] addressed these challenges and are now the prevalent choice. Transformers for APR and other SE tasks have evolved by incorporating new representations [22], new loss functions [24], and increasing their size [71]. This enables them to understand more complex syntactical structures [45, 78], and make human-competitive repairs [18].

Recent LLM-based approaches to code repair include Neural Machine Translation (NMT) [15], cloze-style repair [72], instruction and chat-based methods, and frameworks with iterative LLM calls [14, 58]. NMT methods are trained to repair code by translating from buggy lines to fixed lines, i.e., by learning repairing edits. In the cloze-style repair approach, buggy lines are masked to allow a model to infer correct code directly given surrounding code tokens as context. Recent work has leveraged the use of zero-shot LLMs to successfully perform cloze-style APR on numerous benchmarks [25, 71, 72]. Instruction-based LLMs and chat-based LLMs are adept at following the queries specified by the user. In the context of APR, developers may describe the bug or the intended behaviour of the code snippet. The LLM would suggest precise fixes or generate corrected code snippets directly [73, 74]. Chat-based LLMs add a layer of accessibility to APR which developers are able to engage with. The iterative process of refining queries based on feedback allows a deeper understanding of the problem and more customized solutions [14, 60].

Frameworks with iterative debugging use self-reflection abilities of language models to provide feedback to past partial solutions and debug the code further. Meanwhile, combining different LLMs and approaches to one framework via autonomous agents has been a recent tendency [75]. Therefore, the code repair task, similar to other automated decision-making tasks, is prone to be solved with a collection of tools rather than one model. In this line of thought, round-trip translation is a novel approach to repairing code by translating it to an intermediate language and then back. While in cloze-style and NMT settings, a patch is generated in one single attempt, and in iterative debugging [21], a patch is evolved by a number of calls to the LLM, RTT attempts to repair the code in exactly two steps. Moreover, NMT and cloze-style techniques differ from the RTT approach proposed in this paper in that they are fine-tuned on, or prompted to perform, the APR task, whereas the proposed RTT approach uses LLMs without any fine-tuning or prompting for APR.

2.4 Round-Trip Translation

RTT involves translating a text from its original language to an intermediate language and then translating it back to the original language. Our use of RTT was inspired by the practical observation that, for our secondary languages, we would check or correct errors using RTT through publicly available NMT tools. The value of this practice was confirmed in a study by Hermet and Désilets [16]. Other uses of RTT in NMT include improving translation results [42], and testing the accuracy of a translation model [80]. In the context of APR research, RTT has previously been used for data augmentation [3, 59] and evaluating code correctness [7].

3 REPAIR THROUGH ROUND-TRIP TRANSLATION

Our study evaluates the effectiveness of APR based on round-trip translation (RTT) using nine recent LLMs. A high-level overview of the RTT approach was presented earlier in Figure 1, illustrated with concrete examples that are taken from our experiments. Specifically, the process uses LLMs for two subsequent translations: first, to translate buggy code from its source language to an intermediate language (which can be either another programming language or a natural language), and second, to translate the intermediate representation back to the original source language.

3.1 Motivation to Use Round-Trip Translation for Program Repair

We hypothesize that RTT is capable of repairing bugs as a result of the *regression toward the mean* or homogenization phenomenon exhibited by generative language models. The reasoning is as follows: The LLMs employed in RTT are trained on vast real-world code corpora. Such models can treat code as natural language due to the naturalness hypothesis [23]. They generate or summarize code by iteratively selecting the sequence of the most probable tokens, or the most probable sub-sequences of tokens, for example, using beam search [63]. The probability is estimated by the language model based on its weights, and adapted during the model training to return the most frequently occurring tokens in similar contexts. Ray et al. [54] have shown that frequent code patterns in large real-word code corpora are bug-free. Thus, as a result of training LLMs on these corpora, they have a tendency to generate code that is also bug-free. This process in which LLMs return the most probable tokens during generation can be viewed as regression toward the mean, where noisy samples are replaced by samples closer to the mean. Therefore, each translation step in RTT homogenizes the source fragment toward a less noisy variant that is closer to the expected most probable code, a patch candidate. In the context of code, bugs have been shown to act as a form of noise that is less natural than the mean [54], so they should be reduced or eliminated over the course of round-trip translation, thereby making APR a latent capability of LLMs trained on code.

3.2 Formulation of Round-Trip Translation

Formally, our approach can be described as follows. Let $x = \{x_1, ..., x_n\}$ be a buggy code snippet split into n tokens and $\tilde{x} = \{\tilde{x}_1, ..., \tilde{x}_m\}$ its round-trip translated version with m tokens, i.e., a candidate patch. We use LLMs as neural machine translation models: $LLM_{A\to B}(\cdot)$ from language A to B, where $A \neq B$, and $LLM_{B\to A}(\cdot)$. The round-trip translation of a code snippet x is a two-legged translation. The first leg, *forward translation*, produces a sequence in language B, and the second one, *backward translation*, generates code in language A from the sequence in language B. The whole process can be expressed as:

$$\tilde{x} = LLM_{B\to A}(LLM_{A\to B}(x)). \tag{2}$$

The total probability of the candidate patch generated by RTT, can be expressed with an intermediate representation r of x as follows:

$$P(\tilde{x}) = P(\tilde{x}|r) \cdot P(r). \tag{3}$$

Probabilities $P(r)$ and $P(\tilde{x}|r)$ can be approximated with available LLMs according to Eq. 1. Therefore, we use two legs of translation to approximate the candidate patch $\tilde{x}$ in Eq. 3:

$$P(\tilde{x}) = P(r) \cdot P(\tilde{x}|r) \approx \prod_{i=1}^{k} P_{LLM_{A\to B}}(r_i|x, r_{j<i}) \cdot \prod_{i=1}^{m} P_{LLM_{B\to A}}(\tilde{x}_i|r, \tilde{x}_{j<i}).$$

In this work, we use different intermediate languages B with the goal of encouraging a diverse range of representations, namely natural language (English) and programming languages.

We also formalize the notion used to investigate if RTT can indeed repair bugs. We denote a benchmark with N buggy code snippets as $\{x^i\}_{i=1}^{N}$, and let $Plausible(x) \rightarrow \{0;1\}$ be a function that returns 1 if code snippet x passes all test cases [79]. Then, to evaluate if RTT can indeed repair bugs, we check if a collection of snippets after round-trip translation has a higher overall plausibility than the original collection, expressed by the following equation:

$$\sum_{i=1}^{N} Plausible(\tilde{x}^i) > \sum_{i=1}^{N} Plausible(x^i). \tag{4}$$

The practical implementation and evaluation of RTT is discussed in more detail in Section 4.3.

4 EXPERIMENT DESIGN

The following five research questions guide our evaluation of RTT's performance on the APR task:

RQ1: How well does RTT perform repairs with a programming language as intermediate representation?
RQ2: How well does RTT perform repairs with natural language (specifically English) as intermediate representation?
RQ3: How sensitive is RTT repair performance to variation of the LLM's temperature hyperparameter?
RQ4: What quantitative trends can be observed in the patches generated by RTT?
RQ5: What qualitative trends can be observed in the patches generated by RTT?

To address these research questions, we use nine LLMs and four APR benchmarks discussed in detail below. The selection of these models and benchmarks was guided by ensuring a diverse and thorough evaluation of RTT for APR.

4.1 Models

We use nine distinct transformer-based language models for our evaluation. Their sizes, architectures, and characteristics of training datasets are shown in Table 1. We select the models based on two main requirements: *(i)* they are trained on large code corpora and perform well on code-related tasks; *(ii)* they can perform both legs of a round-trip translation, through an NL or another PL. None of the model variants we use were originally trained or fine-tuned for code repair, and we use them *as-provided*, without additional fine-tuning or training. Although the models' original goal was not code repair, we consider the *outputs* of the backward (second) translation leg as *candidate patches* in our experiments. Observe that one can choose to use different models in each leg of the translation, removing the need for our second requirement. However, in the context of this paper, we use the same model in each leg. This decision is discussed further in subsection 4.9.

PLBART [2] (Programming Language Bidirectional and Auto-Regressive Transformer) is a large language model designed for code-related tasks. Based on the BART [35] model, at the time of model release it excelled in understanding and generating code across different programming languages. PLBART follows the same architecture as its predecessor, a sequence-to-sequence transformer architecture with 6 encoder-decoder layers. Only one size was released initially: PLBART-base (140M). However, a bigger version, PLBART-large (400M), was distributed later in the same year.

PLBART is trained on a dataset in Java, Python, and natural language (English). As a result, the model has learned the patterns, semantics, and syntax of programming languages and grasped the foundations of code understanding and generation. This results in a base model that is versatile and able to achieve decent results throughout a large range of tasks. At the time of its publication, PLBART achieved state-of-the-art results in text-to-code, code-to-text, and code-translation tasks. In addition, the authors have fine-tuned the model on a series of downstream tasks and released the checkpoints. Furthermore, they also studied and released models fine-tuned on multi-tasking corpora and multi-language which can increase the versatility and robustness of the model even further [4]. At the time of the writing of this work, the authors have released a total of 53 fine-tuned models. Of those, we use the *base* models fine-tuned on translation between Java and C# and the *base* models fine-tuned on code summarization (Java → NL) and code generation (NL → Java).

CodeT5 [68] is a large language model designed for code-related tasks. It is based on the T5 model (Text-to-Text Transfer Transformer) [53] and is further tuned for code understanding. Like its predecessor, CodeT5 has a transformer-based architecture. Introduced by Google researchers, T5 was designed as a series of multiple encoder-decoder layers that capture contextual information and relationships in the code. To this end, CodeT5 was trained with a new identifier-aware denoising objective that aims to improve the model's code comprehension. Initially, two sizes were released: CodeT5-small (60M) and CodeT5-base (220M). A year after this release, and additional was delivered: CodeT5-large (770M) [33]. At the time of the writing of this work, a new family of models has been released CodeT5+ [67] that includes five different sizes (220M, 770M, 2B, 6B, 16B).

Table 1. Overview of language models used for RTT.

Model	Size	Base Model	Architecture	Data Source
PLBART	base (140M)	BART	encoder-decoder	StackOverflow BigQuery
CodeT5	base (220M)	T5	encoder-decoder	CodeSearchNet BigQuery
TransCoder	~440M	T5	encoder-decoder	Google BigQuery
SantaCoder	1.1B	GPT-2	decoder	The Stack (v1.1)
InCoder	1.3B 6.7B	Mixture of Experts	decoder	StackOverflow GitHub/GitLab
StarCoderBase	15.5B	GPT-2	decoder	The Stack (v1.2)
GPT-3.5-Turbo	Not Disclosed	GPT-3	decoder	Public Data
GPT-4	Not Disclosed	GPT-4	decoder	Public Data
GPT-4o-mini	Not Disclosed	GPT-4o	decoder	Public Data

CodeT5 follows a two-step training process. The first step results in a base model which accumulates most of the code understanding but does not excel in any task specifically. Fine-tuning the model toward specific code-related tasks such as summarization, translation, and generation (among others) has resulted in the model achieving its best results. The authors have further trained and released a series of fine-tuned checkpoints that cover most of the tasks in the CodexGLUE benchmark [40]. In some of these tasks, the authors have also fine-tuned the model to cover multiple programming languages, resulting in a more general knowledge model while maintaining most of its fine-tuned advantage [4]. We use the *base* size models fine-tuned on the same types of tasks as PLBART.

TransCoder [56] is a large language model specifically created with the sole purpose of source code translation between different programming languages, namely C++, Java, and Python. Researchers at Meta AI focused on creating a transcompiler or source-to-source translator. Rule-based transcompilers often require larger resources and perform worse than the neural approaches. However, the lack of data poses a major challenge in the code-translation domain. TransCoder uses an unsupervised approach to overcome this obstacle and relies simply on monolingual source code.

Researchers trained the model with the three principles of unsupervised machine translation (initialization, language modeling, back-translation) as indicated by Lample et al. [32]. They transformed these principles into three forms of training: cross-lingual masked language model training, denoising auto-encoding, and back-translation. These techniques can take place with monolingual data and, in the case of back-translation, in parallel input corpora, i.e., the corpora of aligned examples in two languages. This resulted in a model that outperformed every other baseline such as *j2py*[2] at the time by a significant margin, as well as an easily generalizable model for other programming languages. This makes it a promising candidate for an RTT pipeline, where we aim to reduce noise or bugs in two steps, from the original buggy example to the translated intermediate representation and back to obtain the final candidate patch.

SantaCoder [5] is an open-source large language model designed for tasks related to code. The model was developed by the BigCode project, a scientific collaboration whose goal is the creation of ethical and responsible large language models for code. Currently, it supports three programming languages: Java, JavaScript, and Python. Its structure is a decoder-only transformer and holds approximately 1.1B parameters.

SantaCoder features multi-query attention and was trained on the Stack dataset (v1.1) [30] through the fill-in-the-middle objective. Despite its relatively small size, the authors claim that the model achieves comparable to improved performance when juxtaposed with previous multilingual models such as InCoder-6.7B. The authors showed that SantaCoder outperformed these models in infilling and left-to-right generations for some benchmarks. Since SantaCoder is released under an OpenRAIL license, the support of this model also reinforces the development of ethical and responsible models for code. We use SantaCoder for RTT with NL since the model learned to operate with docstrings during pre-training.

InCoder [19] is a large language model able to perform program synthesis as well as code editing in 28 programming languages. The researchers' goal was to design a model that is able to infill arbitrary regions of code in different settings such as comment generation, type inference, and variable-renaming. InCoder differs from its competitors by offering left-to-right generation, masking, and infilling in a single model. Therefore, it offers a comprehensive solution for code manipulation and generation.

The model was trained with a causal masking objective, which allows the extraction of context from both sides of the infilled region [1]. The training corpus consisted of code files and repositories from GitHub and GitLab, as well as a corpus extracted from StackOverflow. The structure of the model follows the one described by Artetxe et al. [8]. InCoder can also perform zero-shot tasks while still achieving comparable performance to similar left-to-right code synthesis models. This result indicates that the additional editing and infilling capabilities of the model do not hinder its program synthesis capabilities. The authors released two different sizes of the model,

[2]https://github.com/natural/java2python

1.3B and 6.7B parameters, showing that although trained with the same data, the increased size also increased its performance. These models offered a powerful tool with the ability of code infilling using bidirectional context that expanded on previous left-to-right generational models. We use InCoder in a similar fashion to SantaCoder.
StarCoder [36] follows the same structure as SantaCoder. The main difference is that StarCoder is a 15.5B-parameter model. StarCoder is an open-source large language model trained on source code and natural language. Being also developed by the BigCode project, its creation contributes to the development of ethical and responsible language models for code. Furthermore, this model was trained on over 80 programming languages, Jupyter notebooks, and a large amount of Git communications such as commits or issues.

Similar to its predecessor, StarCoder also features multi-query attention and was trained on a bigger version of the Stack dataset (v1.2) [30] through the fill-in-the-middle objective. The authors claim that StarCoder outperforms every other multi-lingual open large language model for code and matches the OpenAI *code-cushman-001* model. They released two versions of the model, StarCoderBase and StarCoder, which is a further fine-tuned version in Python. The model is also able to handle a context of up to 8 thousand tokens. Longer contexts allow for new and improved applications of the model such as code autocompletion, modifications through instructions, or summarization of code. StarCoder is also released under an OpenRAIL license, supporting the goals of the BigCode project. Two versions are released: StarCoderBase and StarCoder (fine-tuned on Python), of which we use StarCoderBase in the infilling mode for experiments with NL as intermediate.
GPT-3.5 [10], **GPT-4** [46], and **GPT-4o-mini** [47] are models from OpenAI's series of decoder-only transformer-based large language models. Unlike the previously mentioned models, they are not designed solely for code-related tasks. The extensive training data, which includes natural language and code, provides the necessary knowledge to perform a wide range of tasks including code-related ones. Despite the similarity of GPT-4 to its predecessor, GPT-3, the significant increase in parameters leads to the model outperforming in code-related and NLP tasks.

These models perform well in tasks such as code translation, summarization, and generation, even though they were not trained or fine-tuned for code-related tasks. The ability to handle large contexts and their general purpose natural and programming language understanding allow the models to grasp the syntax and semantics of the code in an indirect way. These aspects also make the models more versatile for any tasks that may require natural language and source code, such as code summarization or generation. The models are also the only completely closed models in this section. Therefore, any interaction with them must be done through the OpenAI API provided.[3] However, the availability of API endpoints has also facilitated real-world application of the models in many fields such as support, sales, content creation, and programming among others. We have chosen to use the GPT models only with NL as intermediate to limit the costs.

4.2 Benchmarks

We have chosen four diverse APR benchmarks, following Jiang et al. [25]. In this choice, only single-hunk bugs are included. We follow their example because it enables direct comparison between the RTT approach and results of previous work on the NMT-style APR, in which buggy code is directly translated to patches. Furthermore, this decision only modifies the original size of two benchmarks: Defects4J v1.2 and Defects4J v2.0. The other two benchmarks already contain only single-hunk bugs. Single-hunk bugs are frequently chosen in software engineering benchmarks since their scope allows for both, quantitative metric evaluation and feasible manual analysis of patch quality. All the benchmarks are in Java and contain buggy and fixed code, as well as tests to check the test pass rate of the candidate patches generated by RTT. Note that we use the concepts *problem*, *bug*, and *code example* interchangeably, because we use buggy code examples with single-hunk bugs only.

[3] See https://platform.openai.com/docs/guides/gpt. Specifically, we use the *gpt-3.5-turbo-1106*, *gpt-4-0613*, and *gpt-4o-mini-2024-07-18* models.

QuixBugs [37] is a program repair benchmark with 40 buggy common algorithmic programs such as *bitcount* and *bucketsort* and their fixed versions. These problems were collected in the Quixey Challenge, in which programmers were provided with a buggy implementation of a classic algorithm that would need to be fixed in under a minute. Given that the problems were designed as a programming challenge for humans, it is a valuable testbed for program repair techniques given their diversity and realism.

Manual translation from Python to Java was done carefully after the challenge. Given the differences between the programming languages, many revisions took place. Special attention was put into maintaining the one-line confinement of all bugs despite Java's verbosity. Although the initial version of QuixBugs was not entirely suitable for automatic program repair in Java, recent work has thoroughly reviewed and improved the dataset by providing correct versions and integrating their findings in a newer version that is used to evaluate automatic repair techniques [77]. This update greatly enhances the applicability of the benchmark and allows for a more comprehensive evaluation of APR techniques. Currently, the benchmark consists of 40 programs in Java and Python with one single-hunk bug in each.

The original classification of the programs was performed based on defect types, providing a basic foundation and guide such as *incorrect variable* or *incorrect comparison operator*. However, posterior classifications have been done ad-hoc to further refine the understanding of the bugs [77]. This makes QuixBugs a valuable resource to evaluate and compare automatic repair techniques. Although it consists of a relatively small number of programs, the diverse set of bugs enables comparative studies against realistic problems. Furthermore, the coverage of multiple languages allows the testing of a wider range of tools and more diverse and extensive analysis.

HumanEval-Java [25] is introduced to address the potential issue that some of the APR benchmarks may have been used as source for training data for some models. HumanEval-Java is derived from HumanEval [13] and, unlike its predecessor, this benchmark is tailored for automatic program repair. Although the original dataset was written in Python, the authors manually translated every problem to Java along with their test suites. Furthermore, bugs were injected into the Java programs, creating the final dataset composed of 164 single-hunk Java bugs. Single-hunk bug datasets are very desirable in the field of automatic program repair since they allow users to try different techniques and models even with small context sizes while facilitating benchmarking and maintaining real-world relevance. The authors aimed to introduce a range of different defects, from simple incorrect operator usages to more complex logical bugs that may require the removal or editing of multiple lines to fix. This manual translation and bug injection ensures that no model has been trained on this benchmark before, making it an unbiased and fair dataset in Java to compare multiple models.

Defects4J v1.2 is one of the first releases of the Defects4J benchmark [27]. In its turn, Defects4J is one of the most recognizable benchmarks in the field of APR. It has built a bridge between the research and real-world problems people faced in their work. The benchmark consists of a collection of reproducible bugs from open-source Java projects. Furthermore, test suites are provided which significantly facilitates the use of iterative tools for APR. These bugs have a wide range of complexity and are related to multiple domains, offering users a comprehensive benchmark for new tools and techniques. Over time, Defects4J has been updated resulting in different versions with deprecated bugs and additional projects.

Defects4J v1.2 provides 395 bugs (4 are deprecated) from six open-source Java projects (Chart, Closure, Lang, Math, Mockito, and Time) along with different properties of the defects, the buggy and fixed version, dependencies, and triggering tests. Each defect has a different test suite provided to automate the new test. Researchers in academia and industry have leveraged this benchmark to study different iterative techniques such as genetic algorithms and machine learning approaches. The availability of unit tests along with the detailed metadata allow thorough comparative studies among techniques, bug types, project sizes, etc. Moreover, it gave generalizability, effectiveness, and efficiency measures for new APR tools. The deeper analysis of this release [61] along with its novelty resulted in this benchmark becoming one of the essentials in the automatic program repair field.

Defects4J v2.0 is the latest stable version of the benchmark. It includes additional 438 bugs from nine open-source Java projects. Every bug comes with relevant metadata, a fixed version, and a test suite. This new version provides an even more thorough and diverse collection of defects to test APR techniques. Since this benchmark follows the same structure as the previous versions, researchers can easily expand their experiment and confirm or disprove their former results. The inclusion of new defects and projects only enhances the reliability of the benchmark for evaluating the mentioned metrics. The inclusion of more projects is especially essential to measure the transferability of the techniques across different bug types and environments.

4.3 Implementation of Round-Trip Translation

The RTT pipeline comprises four main steps: preprocessing of input buggy code, generation of translations, postprocessing of RTT-generated outputs, and their validation. We refer to *input* source code as *buggy code*, *buggy examples* or simply *bugs*, while *outputs* of the RTT pipeline correspond to *candidate patches* in APR terminology. The first three steps are shown in Figure 1. We add the fourth evaluation step in the current section. The result is a versatile and parallelizable pipeline able to generate and validate RTT patches with diverse models and test against different benchmarks. Our pipeline extends the framework of Jiang et al. [25].
Step 1: Preprocessing and Prompting. We follow the common practice and extract solely the buggy function as is also done by Jiang et al. [25]. To conform with language models requirements, we add prefixes, suffixes, masks and/or general style changes, such as removing newline characters, before tokenization. We insert a Javadoc header that serves as a prompt for the infilling models (SantaCoder, StarCoderBase, InCoder). The extraction of code and preprocessing is automated and requires no human intervention. Section 4.5 contains the exact prompts used for the models.
Step 2: Round-trip Translation. In the round-trip translation step, we generate two translations for each buggy example using the same type of LLM for both RTT legs: from preprocessed buggy code to an intermediate language and from the intermediate language back to the original language. We generate five different translations per leg in the round-trip translation using LLMs with non-zero temperature to ensure the diversity of the intermediate representations and final candidate patches. Therefore, for each buggy example in a benchmark, we obtain five translations in the intermediate language and 25 final candidate patches, i.e., five from each intermediate translation. Temperature and other model-specific hyperparameters are described in Section 4.4.
Step 3: Postprocessing. We also perform minor postprocessing of the RTT-generated candidate patches to ensure that function signatures are as expected by the test suites. Therefore, we extract code if both code and text are generated and remove extra tokens, which also increases the readability of patches. This process is also automated and does not require human intervention. Section 4.6 contains more details on postprocessing with an NL and PLs as intermediate translations.
Step 4: Evaluation of RTT Results. The final step evaluates the postprocessed RTT results against the test suites provided by the benchmarks. We calculate additional metrics for each candidate patch to evaluate the performance of the models and measure the effectiveness of RTT for APR.

4.4 Hyperparameters for Language Models

We use the recommendation of the authors of the models when choosing the hyperparameters, unless specified otherwise in the current section. All the final hyperparameter values are reported in Table 2. In detail, we set the number of beams to 10 and the temperature to 1 for PLBART,[4] CodeT5,[5] and TransCoder.[6] For SantaCoder,[7]

[4] https://github.com/wasiahmad/PLBART
[5] https://github.com/salesforce/CodeT5
[6] https://github.com/facebookresearch/TransCoder
[7] https://huggingface.co/bigcode/santacoder

Table 2. Hyperparameters used in RTT.

Model	Number of beams	Temperature first leg	Temperature second leg	Top-p
PLBART	10	1	1	-
CodeT5	10	1	1	-
TransCoder	10	1	1	-
SantaCoder	1	0.3	0.4	0.95
InCoder	1	0.3	0.4	0.95
StarCoderBase	1	0.3	0.4	0.95
GPT-3.5	1	1.0	1.0	0.95
GPT-4o-mini	1	0.6	0.6	0.95
GPT-4	1	0.3	0.2	0.95

StarCoder,[8] and InCoder,[9,10] we follow the hyperparameter setup reported in their public demos with a number of beams of 1 and Top-P nucleus sampling of 0.95. Although this choice is reflected in the respective publications [5, 19, 36], they chose a temperature of 0.2 when generating a single output and a temperature of 0.8 when generating up to 100 outputs. We found that a temperature of 0.2 led to insufficient variation. To account for the increased number of generated outputs, we modify the recommended temperature to 0.3 for the first leg (code-to-text) and 0.4 for the second leg (text-to-code).

For the GPT models, we start following the advice backed by OpenAI [11] that for code-generation tasks, LLMs should use a lower temperature when generating structured code compared to natural language generation and therefore choosing 0.3 and 0.2 for each leg of the translation. We perform a temperature sweep for GPT-3.5 and GPT-4o-mini (further discussed in Section 5.3, arriving at different best-performing temperatures for said models). We decide to use top-p close to one to increase diversity but follow OpenAI guidelines to not drastically modify temperature and top-p simultaneously. Moreover, we experimented with different other hyperparameters. For example, we tried different tags in the Javadoc header and values of *repetition_penalty*, *length_penalty*, *no_repeat_ngram_size*. Albeit we did not perform a systematic study, the results from other prompts and the use of these hyperparameters did not show any promising improvement in our use case. Therefore, we have used the parameters as reported in Table 2 and the prompts described in Section 4.5.

4.5 Prompt Choice

The prompts for the instruction and cloze-style models differ due to the presence of the system message in the GPT models and the special infilling tokens for the cloze-style models.

4.5.1 GPT models.

We do not use conversation memory in-built for the three GPT models and run only one forward and one backward translation step. The **system message** is as follows:

```
You are an expert programmer in all programming languages.
```

The **user prompt** for PL → NL summarization (forward translation) has been chosen in the following way:

```
Create a Javadoc for the Java function delimited by triple backquotes. Do not return generate the
method again, return only the Javadoc. Java function: ```{buggy code}```.
```

[8] https://huggingface.co/bigcode/starcoderbase
[9] https://huggingface.co/facebook/incoder-1B
[10] https://huggingface.co/facebook/incoder-6B
[11] https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api/172683

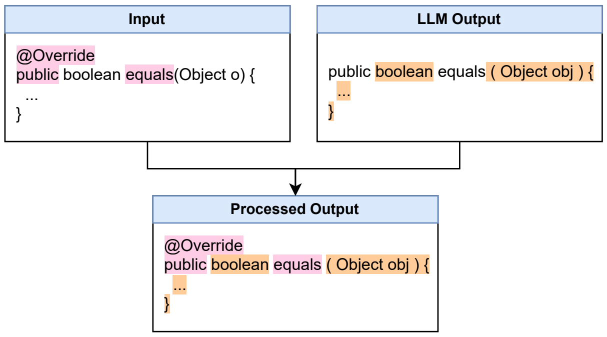

Input
@Override
public boolean equals(Object o) {
...
}
LLM Output
public boolean equals (Object obj) {
...
}
Processed Output
@Override
public boolean equals (Object obj) {
...
}

Fig. 2. Post-processing step to overwrite scope and method name in the output of the LLM.

For NL → PL code generation (backward translation), the **user prompt** is as follows:

```
Given the signature of a Java function and its Javadoc delimited by triple backquotes, generate
the body of the function. Do not generate any additional methods nor repeat the Javadoc nor give
any explanations. Return only the completed function without any comments.```{NL description as a
comment and function signature}```.
```

4.5.2 Open-source Models. For the rest of the models, we follow the default settings with the exception of the following modifications. We ban the word *TODO* when generating code with StarCoder and SantaCoder, since a considerable amount of generated candidate patches were left blank after the use of the word. In addition, the tag *@description* is inserted in the Javadoc header to prompt the models to generate natural language summaries after the description tag.

The resulting prompt for the PL → NL step (forward translation) has the following form:

```
/* @description <INFILL>
*/
{buggy code}.
```

We prepend the NL summary with the header $< |file\ ext = .java| >$ when using InCoder for the code generation (second leg) to improve the overall results of the model by giving context. The prompt for NL → PL code generation (backward translation) is, therefore:

```
<| file ext=.java |>
/* @description {NL description}
*/
{function signature}.
```

4.6 Ensuring Testability of Candidate Patches

When generating a candidate patch with RTT, we take two post-processing approaches according to the model used. For models such as PLBART, CodeT5, and TransCoder, we observe that they do not retain essential contextual information such as method names, scopes, or annotation (e.g., `@Override`). To address this limitation, we automatically overwrite the signature of the generated candidate patch with the appropriate ones from the buggy code. This ensures that the candidate patch is tested regardless of small errors such as a change of the function name. This process is shown in Figure 2.

In contrast, when using the rest of the models, we provide the function signature known from the original buggy example alongside the NL description generated in the first RTT leg (forward translation step). Since function signatures are provided as context, and LLMs use conditional generation setup, it is likely that function logic will be implemented with the variables from the function signatures in the code generated by LLMs. As a result, candidate patches from these models can be tested directly without requiring additional modifications.

Finally, we set up the RTT pipeline so that we skip a buggy example if its original code or the generated translation does not fit in the context window of the model and mark such cases in the final results. However, out of the four benchmarks and eight models from our evaluation setup, it happens only for the *Jsoup 15* bug and the StarCoderBase model that we are forced to skip a buggy example due to a lack of computing resources.

4.7 Evaluation Metrics

We compute a total of seven common APR metrics for each candidate patch generated by the RTT pipeline to evaluate the performance of RTT with different models and assess the effectiveness of RTT for APR [79]. We report the following metrics to *Weights & Biases*,[12] an online tool to analyze the models and their results:

- *Compilability* $\in \{0, 1\}$: ability of the candidate patch to be compiled successfully.
- *Plausibility* $\in \{0, 1\}$: ability of the candidate patch to pass all test cases of the corresponding benchmark.
- *Test pass rate* $\in [0, 100]$: percentage of tests passed by the candidate patch.
- *Exact Match* $\in \{0, 1\}$: binary metric to check if the candidate patch exactly matches the target solution.
- *BiLingual Evaluation Understudy (BLEU)* $\in [0, 1]$ [49]: evaluates by comparing the n-grams against target solution.
- *CodeBLEU* $\in [0, 1]$ [55]: extension of BLEU designed for source code.
  The score is a weighted sum of four components: *(a)* BLEU, measuring n-gram overlap; *(b)* Weighted N-Gram Match, assigning higher importance to programming-related keywords; *(c)* Syntactic Abstract Syntax Tree (AST) Match, comparing syntactic similarity through ASTs; and *(d)* Semantic Data-Flow Match, comparing semantic similarity through data-flow graphs.

4.8 Manual Assessment of Correctness

We perform manual assessment of correctness in addition to the automated test suites on the HumanEval-Java dataset. We decide to perform this assessment only on one dataset for three reasons: (a) HumanEval-Java has been created with the goal of reducing data leakage into the models; (b) given the number of runs, number of models, and number of outputs per model, every new benchmark manually assessed quickly increases the amount of patches to a prohibitive amount; (c) Defects4J contains problems that require deep knowledge about the corresponding project or a significant amount of time to verify. For example, a patch containing a different method being called does not always indicate an incorrect patch. In such cases, the reviewer would need to manually inspect the additional methods, compare their logic against the ground truth, and determine if the program semantics are preserved. The overriding of methods and different scopes makes the task of finding the executed code non-trivial.

In the manual assessment, we compare the generated patch against the reference solution provided in the dataset. Each reviewer examined a subset of problems focusing only on plausible patches. For each problem and each run, we reviewed the patches in the order they were generated until we found a correct patch. If the reviewer was unsure about the equivalence of the patch to the solution, it was collectively discussed until a consensus was agreed.

[12] https://wandb.ai

Table 3. Average number of unique problems with at least one plausible patch ± standard deviation over 10 runs, generated with a PL as intermediate. The best results on each dataset are highlighted in **bold**.

Model	Defects4J v1.2 (130 bugs)	Defects4J v2.0 (89 bugs)	QuixBugs (40 bugs)	Human Eval-Java (164 bugs)
PLBART (C#)	1.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
CodeT5 (C#)	1.0 ± 0.0	1.2 ± 0.4	0.0 ± 0.0	1.0 ± 0.0
TransCoder (C++)	1.0 ± 0.0	**3.0 ± 0.0**	0.0 ± 0.0	3.0 ± 0.0
TransCoder (Python)	**3.0 ± 0.0**	1.0 ± 0.0	**0.1 ± 0.3**	**5.0 ± 0.0**

4.9 Further Implementation Details

Model Choice: Forward and backward translations are performed by the same language model. We consciously make this design choice for consistency and simplicity. Using the same model ensures comparable performance across both legs of the translation. We acknowledge that employing different models may offer advantages, but such configurations introduce additional complexity that can quickly increase the number of needed experiments to perform a thorough analysis. This consideration is further discussed in Section 7.
Addressing Model Stochasticity: To account for randomness in LLMs with non-zero temperatures, we run each experiment that uses open source models with 10 different seeds and refer to these as *10 runs.* This helps mitigate the impact of randomness and results in a more accurate representation of RTT capabilities. We perform only one run for each of the experiments with OpenAI's models, because at the moment of writing these models do not allow setting a seed.[13]
Hardware: We run the patch generation on 3 NVIDIA V100 GPUs or 2 NVIDIA A100 GPUs, depending on model needs. We run the test suites for patch validation on a 32-Core AMD EPYC 7601 CPU with 2TB RAM.

5 RESULTS AND DISCUSSION

5.1 Round-Trip Translation through PL

We first investigate the capabilities of LLMs to fix bugs via RTT using another programming language as the intermediate. For this purpose, we use the LLMs which are able to translate Java code into another PL: PLBART (Java ↔ C#), CodeT5 (Java ↔ C#), and TransCoder (Java ↔ C++, Java ↔ Python). Table 3 summarises the bug fixing performance of RTT along with the intermediate language used in RTT. We set plausibility to 1 if at least one of the 10 generated candidate patches passes all the tests for a given buggy code sample, sum up plausibility over buggy code examples in a dataset, and then take an average over 10 runs with different seeds. We refer to the number of unique problems with at least one plausible patch as *plausibility rate.*

For RTT through PL, we observe that the average plausibility rate is low, with at most five bugs repaired on average for the HumanEval-Java dataset with 164 buggy code examples and at most three code examples repaired on average for the remaining three datasets. PLBART only fixes a single bug across the four datasets with RTT. CodeT5 provides at most two plausible patches for any of the datasets. The best performance with RTT through PL is achieved by TransCoder with Python as intermediate PL on three out of four datasets. Moreover, TransCoder is the only model that provided a plausible patch for QuixBugs. We observe that larger models fix more buggy examples than smaller ones, which is aligned with the general tendency of larger models to perform better on downstream tasks [69].

[13] This has the added benefit of limiting overall experiment costs, especially for GPT-4. The approximate cost for the single run using GPT-4 was ~140USD.

The models tend to repeat the candidate patches for a given buggy example over the 10 runs regardless of the non-zero temperature and different random seeds. This trend, in addition to the low plausibility rates and standard deviation obtained, can indicate a potential rigidity in the conceptual mapping between languages, which may limit the model to literal translation, preventing efficient use of context to filter out noise, or bugs. In other words, code-to-code NMT models with similar target and source languages keep the same tokens and logical bugs. This is supported by the fact that TransCoder with Python as intermediate performs the best on three out of four datasets. Python and Java are less alike than C# or C++ and Java, which motivates bigger changes when translating.

Answer to RQ1 (RTT through PL): The use of PL as an intermediate language in our approach, while yielding a very low number of plausible patches, has shed light on a few key points: *(a)* the intermediate translation should differ enough from the buggy code; *(b)* larger models produce better RTT results on APR through PL.

5.2 Round-Trip Translation through NL

We continue our experiments with RTT that uses a natural language (English) as intermediate representation. We report the number of buggy code examples with plausible patches in Table 4. We include average and standard deviation of the plausibility rate over 10 runs with different seeds, as well as the exact values observed in the union of all runs (Any Run) and their intersection (Every Run). Note that the models in Table 4 are ordered by size.

A strong correlation is observed between the model size (excluding models with undisclosed size) and the number of compilable patches (Person's $r = 0.60$). We also observe an even stronger correlation between the model size and the plausibility rate (Person's $r = 0.77$). The only outlier is SantaCoder, which also performs comparably to larger models on other tasks in related work [5]. The growth of the average plausibility rate from SantaCoder (1.1B) to StarCoderBase (15.5B) is less pronounced. This result can be affected by a larger proportion of Java code within SantaCoder training data compared to StarCoderBase training set. In addition, the majority of models with more than 1B parameters fix at least one bug with RTT through NL that is not repaired by RTT through NL with other models. Note that for a fixed model, bug repair performance differs based on the dataset. Thus, the proportion of the buggy input examples with plausible patches is lower for more complex datasets (Defects4J variants) and larger for simpler tasks (QuixBugs and HumanEval-Java).

The low standard deviation of the plausibility rate indicates that most models tend to repair a similar number of bugs in every run. However, the number of fixed bugs on *Any Run* is two to three times higher on average than the number of repaired bugs in *Every Run*. For example, StarCoderBase fixes 16 Defects4J v1.2 buggy input examples in the aggregation of 10 runs and only 6 same bugs in every run. This indicates that RTT is capable of repairing diverse bugs.

The aggregated metrics over 10 runs bring additional perspectives, such as a comparison of unique bugs fixed by RTT and those in related work. Although the number of repaired bugs tends to vary between the models, Figure 3 indicates that in our approach, most models tend to solve at least one unique problem. Therefore, the kind of problems solved by RTT are not only size-dependent, but also model-dependent.

5.2.1 *Comparison with previous work.*

To position RTT in the broader APR landscape, we compare the results to those reported in prior studies. For easier comparison we include Table 5 which compiles the results from models of varying sizes, base and fine-tuned for APR, as well as some DL-based APR techniques from Jiang et al. [25]. Additionally, Table 6 compares the results of our RTT approach with the same fine-tuned models to find the intersection of problems solved. We made this comparison with problems that have at least one plausible patch based on the results reported in their repository since the authors have not released their manual assessment. The

Table 4. Number of unique problems with at least one plausible patch over 10 runs, with NL as intermediate. GPT-3.5 and GPT-4 are run once, and reported in the last two rows of “Any Run”. The best results in each group are shown in **bold**.

	Model	Model size	Defects4J v1.2 (130 bugs)	Defects4J v2.0 (89 bugs)	QuixBugs (40 bugs)	Human Eval-Java (164 bugs)
Avg ± STD	PLBART	140M	1.0 ± 0.0	1.0 ± 0.0	2.0 ± 0.0	3.0 ± 0.0
	CodeT5	220M	2.0 ± 0.0	2.0 ± 0.0	1.0 ± 0.0	4.0 ± 0.0
	SantaCoder	1.1B	7.5 ± 1.3	5.3 ± 1.7	12.1 ± 1.2	31.4 ± 2.0
	InCoder	1.3B	4.0 ± 0.9	3.3 ± 1.5	4.6 ± 0.9	16.5 ± 1.4
	InCoder	6.7B	4.9 ± 1.0	4.5 ± 0.9	7.8 ± 1.3	26.8 ± 2.4
	StarCoderBase	15.5B	**9.2 ± 1.5**	**6.3 ± 1.4**	**19.8 ± 1.5**	**36.9 ± 7.2**
Any Run	PLBART	140M	1	1	2	3
	CodeT5	220M	2	2	1	4
	SantaCoder	1.1B	13	**12**	21	46
	InCoder	1.3B	6	7	8	30
	InCoder	6.7B	10	8	14	41
	StarCoderBase	15.5B	**16**	10	**24**	**52**
	GPT-3.5	UNK	14	2	30	84
	GPT-4o-mini	UNK	10	5	**33**	90
	GPT-4	UNK	**17**	**7**	27	**100**
Every Run	PLBART	140M	1	1	2	3
	CodeT5	220M	2	2	1	4
	SantaCoder	1.1B	2	2	5	**20**
	InCoder	1.3B	2	0	1	8
	InCoder	6.7B	2	0	3	15
	StarCoderBase	15.5B	**6**	**3**	**16**	16

majority of models used in RTT repair at least one buggy example not repaired by the same models fine-tuned for NMT-type of APR methods.

We also compare our top-performing model for HumanEval-Java , GPT-4, against the 10 LLMs studied in the work of Jiang et al. [25], fine-tuned and non-finetuned for the APR task, in Figure 4. Not only is GPT-4 able to generate more plausible patches than any of the tested models by Jiang et al. [25], but 30 out of the 100 bugs are only fixed by RTT through NL and not repaired by any of the tested models without RTT (see Figure 4). This comparison highlights that RTT generates plausible patches for the bugs that common APR approaches have not fixed and emphasizes the added value of RTT in the APR landscape.

Studies show that generating more candidate patches can result in higher repair performance on the datasets used in our evaluation [71, 74]. The authors generate 200 patches per model, compared to 10 patches in our case, which may be the reason why they fix all bugs in QuixBugs with GPT-3.5 [74] or get 26 and 29 plausible patches with InCoder 1.3B and 6.7B, respectively [71]. However, we could not find enough details or replication packages for those studies and do not include them in Table 6. The latter work also shows that plausible patches have, on average, lower entropy than non-plausible ones. In our experiments, we obtain an almost uniform distribution of the number of plausible patches depending on their position, i.e., over how far on the list of candidate patches

Table 5. Number of unique problems with at least one plausible patch from previous work (Jiang et al. [25]).

	Model	Model size	Defects4J v1.2 (130 bugs)	Defects4J v2.0 (89 bugs)	QuixBugs (40 bugs)	Human Eval-Java (164 bugs)
Base Models	PLBART	140M	**25**	**25**	13	40
	CodeT5	220M	3	7	0	5
	InCoder	1.3B	13	19	**18**	40
	CodeGen	2B	14	6	17	50
	InCoder	6.7B	20	20	**18**	**59**
Fine-Tuned Models	PLBART	140M	33	24	15	36
	CodeT5	220M	33	25	17	54
	InCoder	1.3B	43	**38**	20	64
	CodeGen	2B	38	36	20	53
	InCoder	6.7B	**56**	**38**	**24**	**70**
DL-based APR	CURE	N/A	6	6	5	18
	Reward	N/A	20	8	7	**22**
	Recoder	N/A	**24**	11	6	11
	KNOD	N/A	20	**13**	**10**	18

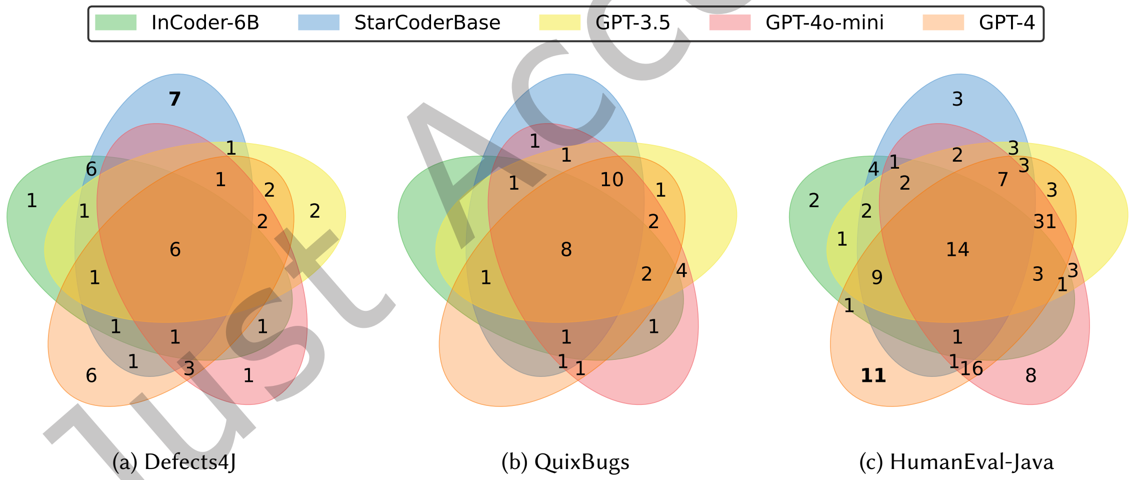


Fig. 3. Number of unique bugs fixed in various datasets by RTT through NL with a fixed language model. The largest number for each dataset is highlighted in **bold**.

the plausible ones occur. This suggests that results can be improved by generating more candidate patches with RTT, at higher resource usage.

5.2.2 HumanEval-Java Manual Correctness Assessment. We conducted a manual assessment of the correctness of RTT-generated patches for the HumanEval-Java benchmark. Table 7 reports the number of manually verified

Table 6. Number of unique problems with at least one plausible patch, shown as P / O / N, with P in previous work, O in our work using RTT through NL (*Any Run*), and N only in our work, not in previous.

Model	Defects4J v1.2 (130 bugs)	Defects4J v2.0 (89 bugs)	QuixBugs (40 bugs)	Human Eval-Java (164 bugs)
PLBART	33 / 1 / 0	24 / 1 / 1	15 / 2 / 1	36 / 3 / 0
CodeT5	33 / 2 / 1	25 / 2 / 1	17 / 1 / 1	54 / 4 / 0
InCoder (1.3B)	43 / 6 / 3	38 / 7 / 2	20 / 8 / 4	64 / 30 / 11
InCoder (6.7B)	56 / 10 / 3	38 / 8 / 1	24 / 14 / 4	70 / 41 / 16

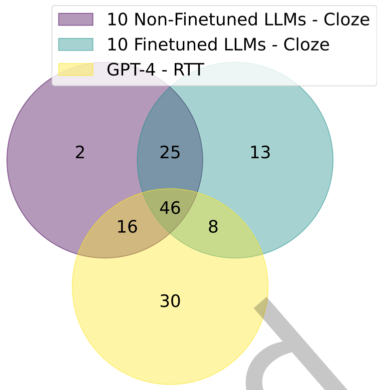


Fig. 4. Comparison of GPT-4 on the number of HumanEval-Java problems with plausible patches.

correct fixes for each model in our approach, as well as in previous work [25]. GPT-4 maintains its position as the top performer for the benchmark. On the other hand, the model with most number of overfitting patches is GPT-3.5 with a total of 17. Although this model tied with GPT-4o-mini in the number of plausible patches, the manual analysis points out the superiority of the later model. For the rest of the models, we see a slight decrease in the average and standard deviation of problems solved. We further appreciate a slightly more substantial decrease in the number of problems solved in Any Run.

To promote transparency and reproducibility, our manual assessment of over 5,000 patches is released alongside our replication package. We hope to enable other researchers to examine and/or extend our work.

Answer to RQ2 (RTT through NL): We observe that *(a)* the trend that larger models obtain better results for bug fixing with RTT; *(b)* although standard deviation of plausibility rate is low, the number of bugs repaired in the union of runs is 2-3 times higher on average than the rate for every run, which highlights that RTT is capable of repairing diverse bugs; *(c)* RTT through NL is able to repair bugs not repaired by the same models fine-tuned for APR.

5.3 Temperature Sensitivity Analysis

Temperature plays an important role in controlling the diversity of outputs from large language models. While lower temperatures increase determinism (although do not enforce it [48]), higher temperatures promote creative and varied responses. In our main experiments, we choose temperatures recommended by the authors (Section 4.4) for simplicity. However, we further analyze its effect on the RTT approach by performing a sweep on the models with the best results, GPT-3.5 and GPT-4o-mini, with six temperatures ranging from $T = 0.0$ (highly deterministic) to $T = 1.0$ (highly stochastic) to assess the effects of temperature in our proposed technique. Although GPT-4 achieves the second best results, it is significantly more expensive than any other model and therefore not considered as a candidate for this experiment. We restricted this sweep to two models since running a wide-range hyperparameter exploration for all models would be prohibitively expensive in terms of computation. Furthermore, narrowing this sweep helps reduce the computational and energy footprint of our experiments, aligning with sustainable research practices.

Table 7. Comparison of number of unique problems with at least one correct patch (manually checked) of RTT and previous work. The best results in each group are shown in **bold**.

	Model	Model size	Human Eval-Java (164 bugs)	
			Avg ± STD	Any Run
RTT Approach	PLBART	140M	3.0 ± 0.0	3
	CodeT5	220M	4.0 ± 0.0	4
	SantaCoder	1.1B	29.4 ± 1.4	45
	InCoder	1.3B	16.2 ± 1.4	27
	InCoder	6.7B	26.0 ± 2.7	39
	StarCoderBase	15.5B	**35.8 ± 7.3**	**51**
	GPT-3.5	UNK	–	80
	GPT-4o-mini	UNK	–	88
	GPT-4	UNK	–	**97**

	Model	Model size	Human Eval-Java (164 bugs)
Base Models	PLBART	140M	39
	CodeT5	220M	5
	CodeGen	2B	49
	InCoder	1.3B	40
	InCoder	6.7B	**59**
Fine-Tuned Models	PLBART	140M	41
	CodeT5	220M	54
	CodeGen	2B	53
	InCoder	1.3B	64
	InCoder	6.7B	**70**
DL-based APR	CURE	N/A	18
	Reward	N/A	**22**
	Recoder	N/A	11
	KNOD	N/A	18

Table 8 summarizes the number of plausible patches produced by these two models under five temperatures. For GPT-3.5, the performance generally improves with higher temperatures, reaching its peak at $T = 1.0$ with 130 total plausible patches. On the other hand, GPT-4o-mini reaches its best results (138 total) at $T = 0.6$, indicating it benefits from some stochasticity. This difference indicates that the optimal temperature can differ considerably across model versions or architectures. Interestingly, for GPT-3.5, there is a monotonic improvement in the total number of repaired bugs as we increase temperature. However, $T = 1.0$ achieves the best results only in half of the benchmarks (Defects4J v1.2 and HumanEval-Java), underscoring that there is no best temperature in every scenario. In a similar trend, GPT-4o-mini shows an optimal temperature around $T = 0.6$, but $T = 0.4$ still achieves a better performance on the QuixBugs benchmark.

These observations tie back to the potential rigidity discussed in Section 5.1, where code-to-code machine translation with closely related languages repeatedly produced the same candidate patches. By raising the temperature, we introduce more noise into the generation process, which can help the model deviate from literal translations and therefore avoid translating the buggy logic. In other words, while low-temperature decoding preserves the same structure and tokens (potentially retaining the bug), higher-temperature decoding results in more diverse translations. This diversity is more likely to break away from any literal mapping that preserves the same error.

Answer to RQ3 (Temperature Senstitivity): We note that (a) temperature affects RTT repair performance, and tuning it can result in meaningful gains; (b) model-specific experimentation is crucial, as we see GPT-3.5 and GPT-4o-mini exhibit distinct optimal temperature settings.

Table 8. Number of unique problems with at least one plausible patch for each benchmark at different temperatures (T). The best temperature for each model is shown in **bold**.

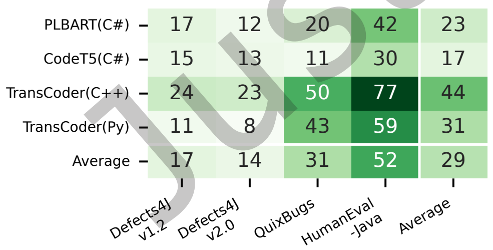
Benchmark	GPT-3.5						GPT-4o-mini					
	T=0.0	T=0.2	T=0.4	T=0.6	T=0.8	**T=1.0**	T=0.0	T=0.2	T=0.4	**T=0.6**	T=0.8	T=1.0
Defects4J v1.2	10	9	9	11	13	**14**	6	9	9	10	**11**	**11**
Defects4J v2.0	2	4	**5**	3	**5**	2	3	3	4	**5**	3	2
QuixBugs	26	30	30	29	**31**	30	28	32	**34**	33	31	33
HE-Java	39	58	68	79	74	**84**	57	77	78	**90**	84	**90**
Total	77	101	112	122	123	**130**	94	121	125	**138**	129	136

5.4 Quantitative Analysis of Generated Candidate Patches

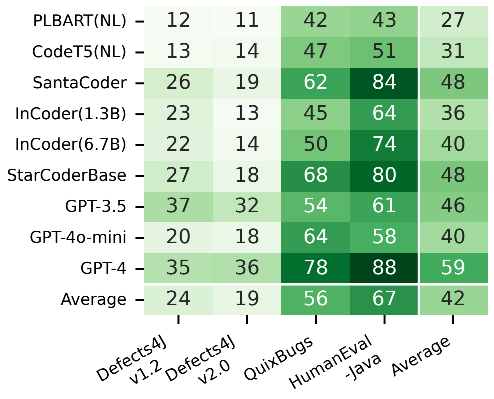
Through a close inspection of candidate patches generated, we are able to gain more insights into the quality of RTT-generated patches. To do this, we investigate compilability, test pass rates, CodeBLEU and other patch candidates' characteristics.

5.4.1 Compilability. We explore the average ratio of compilable patches out of all candidate patches generated over 10 runs and present the results in Figures 5a and 5b for RTT through PL and NL, correspondingly. In general, RTT through PL generates less compilable patches than RTT through NL. TransCoder helps RTT through PL generate a high number of compilable patches, which we associate with its rigorous denoising pre-training objectives.

For RTT through NL, the trend of obtaining better compilability ratios with larger models holds. The previously discussed pattern, where SantaCoder obtains better results than InCoder and slightly worse than StarCoderBase, is observed here, too. Overall, compilability on the datasets with challenging contexts (Defects4J's) is lower than on simpler tasks. Although compilability rates vary across the models and datasets, the majority (80-96%) of

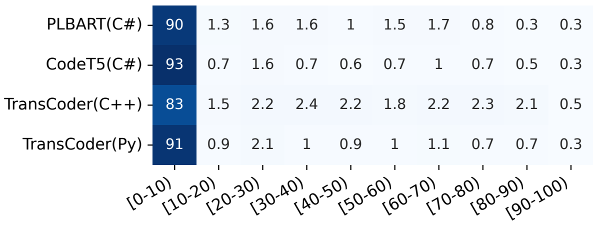


(a) with PL as intermediate language

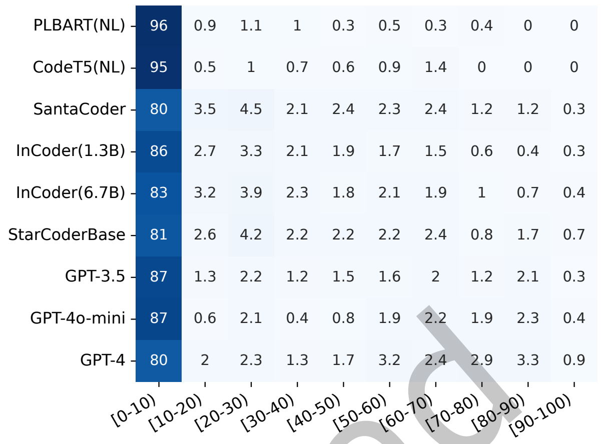


(b) with NL as intermediate language.

Fig. 5. Percentage of compilable candidate patches generated in 10 runs, where applicable, and at 10 attempts for each buggy example.



(a) with PL as intermediate.



(b) with NL as intermediate.

Fig. 6. Percentage of candidate patches in the different test pass rate ranges. We explore what ratio of candidate patches generated by a specific models for any of the datasets pass from A% (incl.) to B% (excl.) tests and report percentage over all generated candidate patches. For example, 96% of candidate patches generated with PLBART with NL as intermediate pass between 0% (incl.) and 10% (excl.) of tests.

candidate patches generated by RTT have low test pass rate, with only 0 to 10% actually passing the test-suite (further discussed in Section 5.4.2).

The trends observed for plausibility rates are also present for the ratio of compilable candidate patches. Similarly to plausibility rates, compilability percentage is higher for experiments with NL than for RTT through PL. On average, the percentage of compilable patches out of all generated candidate patches ranges from 8% (with TransCoder through Python, see Figure 5a) to 77% (with TransCoder through C++) for RTT through PL and from 11% (with PLBART, see Figure 5b) to 88% (with GPT-4) for RTT through NL. Moreover, RTT generates a higher proportion of compilable candidate patches on average for datasets with simpler tasks (QuixBugs, HumanEval-Java) than for datasets with more complex contexts and bugs (Defects4J variants). GPT-4, StarCoderBase and SantaCoder are the leading models in terms of the average compilability for the RTT through NL . However, in the RTT through PL, the best average results are obtained with TransCoder (Java ↔ C++), unlike for plausibility rate which was the best for TransCoder (Java ↔ Python).

5.4.2 Test Pass Rate. To further explore the properties of RTT-generated candidate patches, we analyze the test pass rate of the non-plausible patches. We aim to explore whether compilable patches pass more or less tests, i.e., whether non-plausible patches need a lot of further updates or not.

The test pass rate results are presented in Figures 6a and 6b for RTT through PL and NL, respectively. We take a union of all non-plausible candidate patches over four benchmarks and all runs for each model and report the percentage of candidate patches in these unions that fall into test pass rate ranges from 0–10%, 10–20% and so on, including the beginning and excluding the end of the intervals. The vast majority of such a union of RTT-generated candidate patches pass 0 to 10% of test cases both for PL and NL as intermediate. This result indicates that non-plausible patches require substantial fixing updates to repair the bugs in the chosen benchmarks. One avenue for future work is to experiment with more iterations in the RTT (further discussed in Section 7), or update the model prompts or descriptions with bug summaries or results of not passing test cases, similarly to Grishina et al. [21].

5.4.3 Characteristics of RTT-generated candidate patches. To reiterate, we generate five intermediate translations in the first RTT leg, for example, in English language, and two final translations from each of the intermediates. We denote the first five intermediate translations as A, B, C, D, and E. We enumerate backward translations obtained from each of the two forward translations as A1–A2, B1–B2, ..., E1–E2. The ratio of compilable patches out of a union over all runs and datasets of all generated patches with a fixed model is presented in Figure 7 for RTT through PL.

With RTT through PL, the number of compilable patches is decreasing from the first to the last candidate patch generated from a fixed intermediate translation (fixed letter). The percentage of compilable patches in the first position (A1, ..., E1) is always higher or equal to the compilability rate at the second position for CodeT5 (C#), and TransCoder (C++) and TransCoder (Python). The only exception is PLBART (C#) where the compilability rate at D2 is higher than at D1. The number of plausible patches obtained with RTT through PL is below five on average, as mentioned previously. Thus, the trend between plausibility and the position Ax,..., Ex is not observable from such low average plausibility rates.

For RTT through NL, we do not observe any trend in terms of how frequently first (A1, B1, ..., E1) or second (A2, ..., E2) patches are plausible or compilable. Remarkably, RTT through NL and RTT through PL with TransCoder (Python) have higher plausibility rates than RTT with other models and through other PLs, as shown previously. This observation points back at the discussion of rigidity of code-to-code translation models in the RTT setting: They keep same variable names, other tokens and logical bugs in place. By contrast, NL models and code-to-code models with a PL that differs enough from the original PL show better results in RTT for bug fixing. They abstract and change the input buggy code enough to obtain a different representation that can in the next step lead to a bug fix. The uniform estimated distribution of compilable and plausible patches over the position at which they are generated also supports the argument that sampling more candidate patches from LLMs in the RTT pipeline can improve the bug fixing scores.

5.4.4 CodeBLEU. We calculate average CodeBLEU values for input buggy examples and candidate patches generated by RTT with each fixed model over all runs and datasets and show the frequency of observed values in Figure 8. The metric values are scaled to $[0; 100]$, with highest values being the best. The majority of buggy examples have high CodeBLEU scores, which indicates that target bug fixes are very similar to original buggy code. High CodeBLEU for the majority of buggy examples is also explained by the type of bugs: We only consider single-hunk bugs.

In contrast, the majority of candidate patches obtained with RTT through PL have CodeBLEU between 40 and 60, with the outlier value of ca. 26 frequently observed among candidate patches for Defects4J variants. The most frequently observed average CodeBLEU values for RTT through NL are between 20 and 50, with a similar outlier value ca. 26 and an additional outlier of zero CodeBLEU noticeable for a number of candidate patches for Defects4J versions. The frequency of higher CodeBLEU values increases with larger model sizes. CodeBLEU values are considerably lower than 100 for the vast majority of RTT-generated candidate patches. However, one can observe the trend of regressing towards the mean type of candidate patches with similar CodeBLEU calculated between targets and RTT-generated patches.

Furthermore, although larger models shift the distribution of generated patches to higher CodeBLEU scores, they still remain far below the scores of the input distribution. This indicates that substantially bigger models, although increasing plausibility rates, still produce patches significantly different from the proposed solutions. We further expand on this observation by noting a weak positive correlation between CodeBLEU and two metrics, patch compilability (Person's $r = 0.22$) and patch plausibility (Person's $r = 0.15$), suggesting that CodeBLEU alone is a weak predictor for RTT-generated patches.

5.4.5 Other common APR metrics. Exact Match and BLEU are frequently used to check whether candidate patches resemble the ground truth in benchmarks [40]. Since RTT is aimed at finding functionally correct patches, not

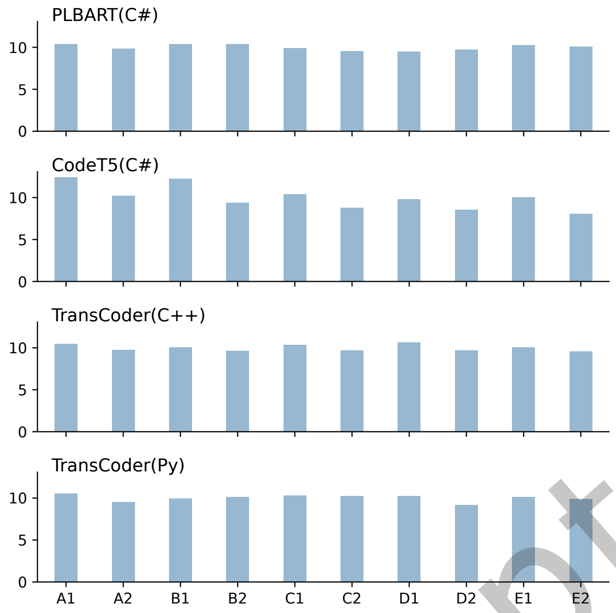


Fig. 7. Percentage of compilable candidate patches in the 10 positions with PL as intermediate. The percentage is calculated over all patches in four benchmark datasets generated with RTT using a fixed model

stylistically equivalent ones, we have found these metrics non-descriptive for RTT. Especially when using NL as intermediate, RTT can freely deviate from the original buggy code's style, evidenced by the average BLEU and CodeBLEU scores ± std over the patches that pass all tests being less than 40.1 ± 0.09 and 63.7 ± 6.7, respectively.

5.4.6 Limitations of RTT for APR. Studies show that the original style of writing texts can be diluted by generative language models [20, 51]. Furthermore, LLMs have been known to generate code containing security flaws [50, 64]. These flaws may compromise the integrity of the application. Therefore, it is recommended that developers thoroughly audit and review any generated code. Our experiments show that bugs in code can be corrected via RTT, but we also see that the original styling of the code is not always retained. Such a restyle leads to challenges with code maintainability, which can reduce the willingness of developers to adopt the approach. This issue will be less of a challenge in projects that enforce a uniform coding style through automated tools. Moreover, the impact will be lower if RTT is applied in smaller contexts, for example, in highly modular projects with localized bugs, where restyling will have limited impact on maintainability.

Answer to RQ4 (Quantitative Analysis of RTT patches): We observe that *(a)* RTT-generated patches, even if compilable, rarely pass most of the tests, therefore needing significant refinement; *(b)* RTT can change the code considerably, reducing usefulness of metrics for ground-truth matching, such as BLEU and CodeBLEU; *(c)* Because RTT can dilute the coding style, it is best used in circumstances where rephrasing does not impact maintainability.

5.5 Qualitative Analysis of Generated Candidate Patches

We manually analyze patches generated by RTT to find out more about the quality of solutions, how they differ among patches, and what the common flaws are. To make the analysis concise, we have chosen to go through

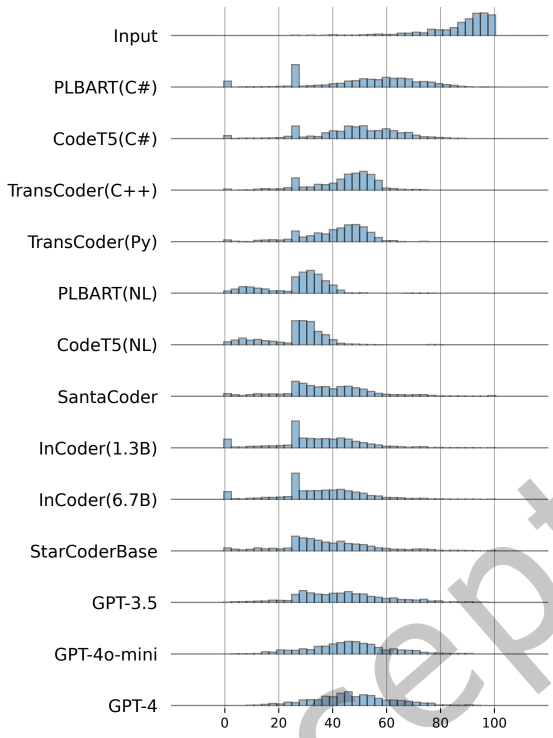


Fig. 8. Histogram on the CodeBLEU scores of candidate patches generated with RTT through PL or NL (NL is default, if not mentioned). The distribution is calculated over all patches generated for four APR benchmarks.

HumanEval-Java patches, starting from the problems solved in this work and not solved by other APR approaches in the scope of our comparison (see Section 5.2.1). This section also includes error and success analysis of the problems in QuixBugs and HumanEval-Java, looking into the reasons for patches to not compile, and the ones that compiled or passed all the tests more frequently than for other problems in the dataset.

Unique problems solved by RTT: In HumanEval-Java, some problems were solved uniquely by RTT and not solved by models reported by Jiang et al. [25]. Several problems have RTT patches with varying efficiency and general logic of algorithms. For example, two factorization problems were solved only by models using RTT and not their fine-tuned or base counterparts. The first problem consists of factorizing a number into the list of its prime factors (FACTORIZE, problem 25), and the second one aims to find the largest of them (LARGEST_PRIME_FACTOR, problem 59). Solutions for FACTORIZE resulted from RTT differ in the way they specify the limit for looping over the numbers: until and including the input number n, e.g., SantaCoder's solution in Listing 1a, or until its square root, e.g., target code from the dataset and Incoder-6B solution. An implementation that stops the loop at the square root of the input number is more optimal because it involves fewer iterations. In case the iterations stop at the square root, the additional check ($if\ (n > 1)$) for including the number itself is needed, for example, for a prime number 11 as input. For reference, the target solution checks for ($n > 1$) while Incoder-6B — for n != 1 after the loop.

The LARGEST_PRIME_FACTOR problem is formatted as a class with two methods: an auxiliary method that checks whether a number is prime and the main method that loops over factors of an input number and chooses the largest prime factor (see Listing 2a). This setup is challenging for RTT because the auxiliary method may get

```java
public static List<Integer> factorize(int n){
    List<Integer> result = new ArrayList<Integer>();

    int i = 2;
    while (i <= (int)(Math.sqrt(n) + 1)){
        if (n %
            result.add(i);
            n = n / i;
        } else{
            i += 1;
        }
    }
    // bug: the following if-statement was missing
    if (n > 1){
        result.add(n);
    }
    return result;
}
```

(a) Target code, reproduced by Incoder-6B up to renaming of the variables and a small change of if-condition.

```java
public static List < Integer > factorize(int n) {
  List < Integer > list = new ArrayList < > ();
  if (n < 0) {
    return list;
  }
  for (int i = 2; i <= n; i++) {
    while (n %
      list.add(i);
      n /= i;
    }
  }
  return list;
}
```

(b) SantaCoder (NL) patch

Listing 1. Problem: FACTORIZE (HumanEval-Java).

lost in the two-step translation. However, some of the generated solutions implicitly ensured that the resulting output was prime. For example, the RTT patch by StarCoderBase loops over numbers from $i = 2$ to the input number n and modifies it with $n = n/i$ (see Listing 2c). This process ensures that all the non-prime factors are excluded and the resulting largest factor is prime. A similar solution was generated by RTT with other models, too, ending the loop at n or $Math.sqrt(n)$ again. Interestingly, StarCoderBase does use the check for $is\_prime()$ in some of its solutions as shown in Listing 2d. While the input to RTT contains this auxiliary method in the context (see Listing 2b), the intermediate translation does not mention it. So, the generated output either "hallucinated" (as LLMs sometimes do when they import non-existing libraries or methods [17, 38]) or copied the method from a similar code snippet the model has "seen" before. This unexpected usage of $is\_prime()$ highlights the data leakage issue from either the benchmark in Python or some textbooks the models might have been trained on.
Differences in Implementation Styles: In line with the efficiency discussion, Listing 3 refers to the calculation of the greatest common divisor between two numbers. The proposed solution includes classic recursive calls, while the plausible generated patch shows the iterative version of this recursion. In this case, the target code is slightly more inefficient due to the recursive calls. On the other hand, Listing 4 shows the classic BIT_COUNT problem. The goal is to count how many bits are activated in a given natural number. There exist not only different approaches for this problem, but their efficiency depends on the context in which they will be used. The proposed solution follows the Brian Kernighan's algorithm, while the plausible patch generated by several models fosters the readability of the approach. Although it is true that the target code may be considered slightly more efficient, the difference is negligible in most scenarios.

Similar to checking if a number is prime when searching for the largest prime factor, in the problem EVEN_ODD_
PALINDROME (problem 107), the method $is\_palindrome()$ present in the context is sometimes used in the RTT-generated patch and other patches perform this check as a one-liner in the generated code in the if-statement, such as: $if\ \ (s.equals(new\ StringBuilder(s).reverse().toString()))$. Both types of checks happen in several models arbitrarily, so we could not draw any conclusion on correlations with a model size or type. Noteworthy, in a similar fashion that methods from the input context are not always used in the generated patches (and the presence of those is usually not reflected in the intermediate translation), built-in Java classes are not always

```
public static boolean is_prime(int k) {
    if (k < 2)
        return false;
    for (int i = 2; i < k; i += 1){
        if (k %
            return false;
    }
    return true;
}

public static int largest_prime_factor(int n) {
    int largest = 1;
    for (int j = 2; j < n; j += 1) {
        // buggy line:
        // if (is_prime(j)){
        if (n %
            largest = j;
        }
    }
    return largest;
}
```

(a) Target code. Method *is_prime* serves as context, while method *largest_prime_factor* is the main function.

```
INPUT
<fim_prefix>

/**
@description <fim_suffix>
*/
public static int largest_prime_factor(int n) {
    int largest = 1;
    for (int j = 2; j < n; j += 1) {
        if (is_prime(j)){
            largest = n %
                }
    }
    return largest;
}
<fim_middle>

OUTPUT: INTERMEDIATE TRANSLATION
/**
@description This method returns
the largest prime factor of the number passed in.
*/
```

(b) Input for RTT with StarCoderBase and its intermediate translation.

```
public static int largest_prime_factor(int n) {
    int largest_prime = 2;
    int i = 2;
    while (i <= n) {
        if (n %
            n = n / i;
            if (i > largest_prime) {
                largest_prime = i;
            }
        } else {
            i++;
        }
    }
    return largest_prime;
}
```

(c) StarCoderBase patch (NL). *is_prime* method from the input context is not used.

```
public static int largest_prime_factor(int n) {
  int largest = 1;
  for (int i = 2; i <= n; i++) {
    if (n %
      if (is_prime(i)) {
        largest = i;
      }
    }
  }
  return largest;
}
```

(d) StarCoderBase patch (NL) with *is_prime* method.

Listing 2. Problem: LARGEST_PRIME_FACTOR (HumanEval-Java).

used for standard manipulations. For example, a sorting algorithm can be either reused from a standard Java library or implemented in the patch.

Common Logical Problems: Several problems in HumanEval-Java have method names that have a misleading effect on the outputs of RTT. In this way, CORRECT_PARENTHESIS (problem 61) has led to one output checking for several types of parenthesis, although the original problem description talks only about the round parentheses "()." However, we use only the code as input for the first RTT leg that summarizes the code. Therefore, we sometimes miss assumptions about the input only present in the original description of the program, for example, those stating that the input will always be composed of round parentheses. An excessive check for several types of parentheses happens in patches generated for other problems, too. This is reflected in the intermediate translation that covered multiple bracket types, even though the input code only had checks for round parentheses.

```
public static int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a%
    }
}
```

(a) Target Code

```
public static int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a %
        a = temp;
    }
    return a;
}
```

(b) StarCoderBase patch

Listing 3. Problem: GCD (QuixBugs).

```
public static int bitcount(int n) {
    int count = 0;
    while (n != 0) {
        n = (n & (n - 1));
        count++;
    }
    return count;
}
```

(a) Target Code

```
public static int bitcount(int n) {
    int count = 0;
    while (n != 0) {
        count += n & 1;
        n >>>= 1;
    }
    return count;
}
```

(b) StarCoderBase patch

Listing 4. Problem: BitCount (QuixBugs)

The misleading method names are some of the common reasons for low plausibility rates. In these cases, an intermediate translation takes the problem far from the input code, and the resulting patch can even have a different number of input arguments, which makes testing on the same dataset's tests impossible. A similar misleading effect occurred for the solution of SMALLEST_CHANGE (problem 73, Listing 5), whereby the model created an intermediate translation that did not describe the intended problem well. So, the solution is generated for a different problem based on variable names (coins) and extra variables set up for the solution, but surprisingly, it is correct for the initial problem, too. In essence, the only useful check in Listing 5d is that $arr[i]\ != \ arr[j]$ which, in this implementation, is realized as $arr[i] < arr[j]$. Once the condition is true, the counter should be incremented by 1, which is done by $coins\ += \ 1$. Thus, the variable min is not useful for obtaining the final answer at all. Moreover, it only makes sense to run the loop as long as $i < j$, which should be an additional condition in the while loop to make the implementation more optimal.

Frequent Failure and Success Cases: Problems with string manipulation usually have high compilability and plausibility rates, even though solution approaches vary. The variation occurs both in the Java class used by a patch, e.g., StringBuilder, StringBuffer, or just String, and in the logic for solving the problem. Listing 6 displays the concatenation problem in the HumanEval-Java benchmark. The purpose of the given function is to concatenate strings given an array. The plausible patch generated by PLBART is similar to the proposed solution.

Continuing the discussion on tasks in HumanEval-Java with string manipulation, in MATCH_PARENS (problem 119), the bug consisted in the wrong if-statement or, otherwise, wrong place to increment and decrement the counter val of unmatched opening parentheses. So, the patch should have either swapped the increment and decrement or changed the if-statement $if\ \ (s.charAt(i)\ \ ==\ \ '(')$ to $if\ \ (s.charAt(i)\ \ ==\ \ ')')$. One of those changes was present in the correctly generated patches. However, a solution created by RTT with SantaCoder used an integer array that served as a stack (see Listing 7). In this solution, opening parentheses are put into a stack that is implemented as an array of integers. Index i loops over the input string. The variable $open$ updates the index of the latest open and unmatched parenthesis occurring in the input, i.e., at the top of the stack. When

```
public static int smallest_change(int[] arr) {
    int result = 0;
    // buggy line:
    // for (int i = 0; i < arr.length; i += 1) {
    for (int i = 0; i < arr.length / 2; i += 1) {
        if (arr[i] != arr[arr.length - i - 1])
            result += 1;
    }

    return result;

}
```

(a) Target code

```
/**
@description <|mask:0|>
*/
public static int smallest_change(int[] arr) {
    int result = 0;
    for (int i = 0; i < arr.length; i += 1) {
        if (arr[i] != arr[arr.length - i])
                result += 1;
    }
    return result;
}
<|mask:0|>
```

(b) Input for RTT with Incoder-6B

```
/**
@description Find the smallest number of coins
    needed to make change
    for a given amount of money.

@param: arr - an array of integers representing coins

@return: the smallest number of coins
    needed to make change
    for the given amount of money

@exception: ArrayIndexOutOfBoundsException
    - if array is empty
*/
```

(c) Incoder-6B intermediate translation (NL)

```
public static int smallest_change(int[] arr) {
     int min = Integer.MAX_VALUE;
     int coins = 0;
     int n = arr.length;
     int i = 0;
     int j = n-1;
     while(i < n && j >= 0) {
         if(arr[i] < arr[j]) {
             min = Math.min(min, arr[i] + arr[j]);
             coins += 1;
         }
         i++;
         j--;
     }
     return coins;
}
```

(d) Incoder-6B patch

Listing 5. Problem: SMALLEST_CHANGE (HumanEval-Java).

```
public static String concatenate(String[] strings){
    String result = "";
    for (String string : strings)
        result += string;
    return result;
}
```

(a) Target code

```
public static String concatenate (String... strings){
    StringBuilder builder = new StringBuilder();
    for (String s : strings){
        builder.append(s);
    }
    return builder.toString();
}
```

(b) Plausible patch by PLBART (NL)

Listing 6. Problem: CONCATENATE (HumanEval-Java).

a closing parenthesis occurs in the input string, an element is "popped" from the stack by decrementing *open*. The solution can be simplified by manipulating just the *open* variable (as it happens to *val* in the target code) and thus have $O(1)$ space instead of $O(n)$ with an extra *stack* array of size *n*. A surprising fact is that the solution with a stack does not use the standard *java.util.Stack* class.

One source for common errors is the challenge of dealing with linked lists and nodes. Thus, patches for problems that require node processing usually fail to compile because of the wrong node handling, such as calling the *next*() method instead of *getSuccessor*(). Some general observations are that smaller models, specifically both Incoders, sometimes output repetitive tokens or very similar lines of code, such as an if-condition that compares a variable to all numbers from 1 to 50 one by one. This can happen when a model's capacity to train is

```java
public static boolean check(String s) {
    int val = 0;
    for(int i = 0; i < s.length(); i++) {

        // buggy lines with swapped val++ and val--
        // if (s.charAt(i) == '(') val--;
        //     else val++;
        //     if(val < 0) return false;

        if(s.charAt(i) == '(') val++;
        else val--;
        if(val < 0) return false;
    }
    return val == 0;
}
```

(a) Target code

```java
public static boolean check(String s) {
  int[] stack = new int[s.length()];
  int open = 0;
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '(') {
      stack[++open] = i;
    } else if (s.charAt(i) == ')') {
      if (open == 0) {
        return false;
      }
      stack[--open] = i;
    }
  }
  return open == 0;
}
```

(b) SantaCoder patch (NL)

Listing 7. Problem: MATCH_PARENS (HumanEval-Java).

not fully used, and performance can benefit from more training. Incoder models also more frequently generate several alternatives within one generation, which looks more like a batch generation, but instead of having several outputs in a batch (or a list), all of them are appended to one generated string.

Some of the most frequently solved problems (with True Positive Rate=1 for the most generated outputs and the majority of models) are common programming or mathematical tasks for which there should be plenty of training data available. Such problems are, for example, GCD and BITCOUNT in QuixBugs, and Fibonacci sequence, string concatenation, and again GCD calculation in HumanEval-Java. Many output patches are similar to the target solutions presented in the datasets up to renaming the variables, changing the order of conditional statements, or other operations that do not affect the logic of the solutions. These observations confirm that even though some of the datasets possibly leak to the training set of the models, the models do not reproduce this code line by line.

Answer to RQ5 (Qualitative analysis of RTT patches): We find that *(a)* While many HumanEval-Java patches are very similar to target code, they have different variable names, and there exist other solutions that do not resemble target code at all, which confirms that even if the models were exposed to the benchmarks, they did not memorize them to the point of copy-pasting the answers; *(b)* Using the code from the surrounding problem context is challenging for RTT patches, and some simple checks made in the surrounding contextual methods are sometimes replaced by one-line checks in the patch body; *(c)* Time and space complexity vary across the generated patches, given the fact that the prompts did not urge models to optimize for those; *(d)* In RTT, method names can sometimes be misleading and result in a shift to a different problem in the intermediate translation; *(e)* RTT as an APR technique results in an indirect fix to a problem leading to more creative solutions as opposed to direct line-specific fixes of fine-tuned models.

5.6 Impact and Potential Usage of RTT in Code Repair Frameworks

The design and development of a single tool or methodology to address the entire spectrum of software bugs can be challenging [41]. For this reason, the evolution of APR has emphasized the value of ensemble approaches [28]. In ensemble frameworks, diverse tools and methodologies contribute their unique strengths in identifying and fixing software bugs. These ensemble approaches underscore the importance of the diversity of tools and align with realistic software maintenance scenarios, where a combination of techniques is employed to achieve the best results.

In the context of APR, agent-based approaches can be used to exemplify this ensemble approach. They are expected to become a new standard for decision-making support and coding in the future. In such systems, one agent can play the role of a router and other agents suggest alternative solutions for a task under consideration, together composing an ensemble setting for generating a solution [57]. In this case, it is beneficial to the final repair target if each individual agent proposes good-quality solutions that differ from the solutions suggested by other agents.

RTT can be viewed as a complementary approach for a repair agent in an APR agent-based framework, where some agents can apply self-reflection or self-debugging, others apply NMT approaches to predict repairing code edits, a cloze-style repair agent fills in the masked buggy lines with a suggestion for correct lines, and an RTT agent generates an alternative repair proposal by translating buggy code to another language and back. Because RTT repairs 46 unique problems not solved by models that employ cloze-style repair, its added value in a multi-agent code repair environment can boost the performance of the framework. Future work can explore the avenue of creating multi-agent APR frameworks based on language models trained for code with tools such as LangChain,[14] LlamaIndex[15] or AutoGPT.[16]

6 THREATS TO VALIDITY

This section discusses four types of threats to validity for this study, structured cf. Wohlin et al. [70, Sec. 6.7 & 6.8].

Internal Validity: To support the validity of our results, we applied RTT with two intermediate representations to four benchmarks and tested nine models. As the benchmarks are publicly available, there is a risk that they were used during training, also referred to as *data leakage*. This threat can be mitigated by using models that remove the benchmarks from their training data. Here, we use HumanEval-Java, which was constructed after the training of any of the models used in this work (except GPT-4o-mini). For the other three datasets, we find an exact match in only 0.03% of the generated candidate patches, which reinforces the validity of the data and results by ensuring that even if the models were trained on the used benchmarks, they failed to copy-paste patches.

Construct Validity: To evaluate the RTT performance, we apply widely used APR metrics, including compilability and plausibility rates. For metrics that depend on test suites, low-quality or easy-to-pass tests could positively bias the evaluation. We mitigate this risk by employing four widely used APR benchmarks with different bug types. However, most of our evaluation uses plausibility as a proxy for correctness, which comes with inherent limitations, such as ignoring potential overfitting of patches to the test code. To address this threat, we manually assess the correctness of over 5,000 patches generated for HumanEval-Java. This helps to balance the difficulty of problems covered by the benchmark with the resources required to evaluate multiple types of patches for each problem and each LLM. Furthermore, HumanEval-Java was created to minimize the effect of possible data leakage, while QuixBugs and Defects4J were already available online. Finally, evaluating the correctness of a large number of patches for Defects4J would require an unfeasible amount of resources and would be prone to evaluator (human) errors, since many of the bugs require in-depth knowledge of the projects in Defects4J.

External Validity: Threats to the external validity concern the generalizability of our approach. We have validated the approach on a representative sample of APR benchmarks, but have not extended the results to language pairs that were not covered. However, we focused our evaluation on single-hunk bugs, therefore, effectiveness may not transfer to more complex multi-hunk or multi-file bugs. These multi-hunk bugs may represent a more realistic scenario that requires richer context and evaluation of cross-hunk consistency. Moreover, we applied

[14]https://www.langchain.com/

[15]https://www.llamaindex.ai/

[16]https://github.com/Significant-Gravitas/AutoGPT

the approach using only nine transformer-based models. Extending the evaluation requires more computational resources and language models that comply with the RTT requirements in Section 4.1.
Conclusion Validity: For our experiments, we used off-the-shelf language models that are publicly available and can be used without retraining. The four benchmarks are also publicly available and widely used in APR research. To support open science and enable replication and verification of our work, a replication package is available.[1]

7 DIRECTIONS FOR FURTHER RESEARCH

This study opens up several avenues for future research, such as expansion of key components of the current study by adding new models, benchmarks, or metrics. Our replication package can be used as a base to expand the study to new models, intermediate languages, and datasets. Interesting directions for future work include:
Model Variation: Investigating models with the same architecture but varied size, e.g. CodeT5+ [67], may shed light on whether some capabilities show only after a specific size threshold [29, 69]. This exploration can result in some conclusions on the scalability of RTT and how model complexity may impact precision and quality of repairs. The emergence of unique repair capabilities at certain model size thresholds can reveal insights on the optimization of APR tools' performance. Another straightforward variation that can influence repair performance is usage of different models in each RTT leg. For example, choosing the newest best performing model for summarization and the best performing one for code generation has a potential on improving RTT via natural language.
Cross-entropy Analysis: A more granular analysis of cross-entropy along with increased sampling from models can provide insights into the naturalness hypothesis and its effects on the approach [71]. Studying how similar the probability distribution of repaired code match natural, bug-free code can help researchers assess the effectiveness of APR methods. This could result in the development of new loss functions that could capture nuances of successful code repair.
Extend Intermediate Representations: Researching the properties of other intermediate representations, not only other programming languages, but also pseudocode or combinations of PL and NL. This exploration can identify what intermediate representations are the most effective for each programming language or bug type. As a result, the use of specifically chosen intermediate can provide with customized solutions that take advantage of the strengths of the various coding languages.
Error Analysis: Classifying the bugs from the benchmark into groups that share similar bug properties could provide trends about which type of errors are fixed with some specific intermediate representations. This information could also be used to generate prompts for the problems adapted to failing tests or compilation errors, similarly to Liventsev et al. [39]. Furthermore, knowing which leg of the round-trip can provide insights for experiments that combine multiple models. Gaining a deeper understanding of the characteristics and limitations of unique patches, such as model rigidity or optimal number of outputs, can help develop more sophisticated translation and repair techniques.
Multiple Round Trips: Since the process relies on the regression toward the mean, repeated RTT may increase the bug-fixing behavior. This investigation can lead to a more nuanced understanding into the mean of the code learned by language models. Researching if repeated translations can incrementally refine the repairs may lead to more accurate and robust results, enhancing the bug-fixing capabilities of RTT-based APR approaches.
Granularity of RTT: One can restrict RTT from translating the whole function to applying changes to certain masked areas of the code. However, RTT's bug fixing capabilities rely on context to accurately filter out bugs. Consequently, future research can investigate if an optimal granularity exists that balances accurate modifications with the amount of context needed. Moreover, studying the effect of increasing the context could allow RTT to

be applied to more realistic multi-hunk and multi-file bugs where a rich context is available but complexity is also increased.

8 CONCLUSION

In this work, we explore the latent capability of LLMs for automated program repair illuminated by the round-trip translation through an intermediate representation. Our experiments confirm the viability of RTT for APR, but also show potential pitfalls and limitations. Although RTT does not outperform state-of-the-art NMT and cloze-style fine-tuned models for APR, we find that RTT, without any fine-tuning costs, is able to repair 46 unique bugs that were not fixed by the same LLMs after fine-tuning them for NMT and cloze-type APR. In more detail, we find that RTT through NL (English) as an intermediate translation generated plausible patches for 100 of 164 bugs with GPT-4 on the HumanEval-Java benchmark, and 97 were determined to be correct in our manual assessment. When using PL as intermediary, we obtain two main insights: First, the intermediate representation should be sufficiently distinct from the original language to ensure effective rectification. Secondly, larger LLMs consistently provide better results than their smaller counterparts.

Through close examination, we uncover several properties and characteristics of RTT-generated patches. The inherent nature of the approach to introduce substantial changes to the code makes it inefficient to use common ground-truth matching metrics, such as BLEU, to assess efficacy. Our qualitative analysis of the patches generated for QuixBugs and HumanEval-Java highlights the complex relationship between RTT and the naturalness of code. While RTT leverages naturalness and the regression toward the mean to remove bugs, this process may also dilute the original code author's style, remove comments, and substantially reformat the code. Therefore, RTT is better suited in circumstances where such rephrasing does not impact maintainability. Together with the ability to provide fixes for bugs that other approaches do not solve, this makes RTT a technique that is particularly useful in collaborative scenarios where multiple methods are used to address a task, for example, in multi-agent or ensemble frameworks. Not only can RTT expand the repertoire of available tools, it also introduces a novel dimension to patch generation by applying a non-traditional approach. Moreover, this expansion comes at a low cost because the approach does not require any fine-tuning of LLMS for the APR task.

Finally, we have sketched a number of promising directions for future research, including experimentation with a wider range of models, a more granular analysis of cross-entropy in code, investigating properties of other intermediate representations, a qualitative analysis of errors, experimenting with multiple round-trips, and restricting RTT to certain 'masked' areas of the code. We believe that new insights into any of these areas will help further the application of RTT to make software engineers be even better at their jobs.

ACKNOWLEDGMENTS

This work is supported by the Research Council of Norway through the secureIT project (IKTPLUSS #288787), and by the European Union the Horizon Europe Marie Skłodowska-Curie Actions (#101151798). The empirical evaluation presented in this article made use of the Experimental Infrastructure for Exploration of Exascale Computing (eX3), financially supported by the Research Council of Norway under contract #270053. In addition, we acknowledge Sigma2, Norway for awarding this project access to the LUMI supercomputer, owned by the EuroHPC Joint Undertaking, hosted by CSC (Finland) and the LUMI consortium through the Research Council of Norway. This research builds on work that has been submitted in partial fulfillment of the requirements for the Master's degree of the first author at KTH, Sweden.

REFERENCES

[1] Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, and Luke Zettlemoyer. 2022. CM3: A Causal Masked Multimodal Model of the Internet. https://doi.org/10.48550/arXiv.2201.07520 arXiv:2201.07520 [cs]

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

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages. In *17th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 1528–1542.

[4] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual Training for Software Engineering. In *44th International Conference on Software Engineering (ICSE)*. ACM, 1443–1455. https://doi.org/10.1145/3510003.3510049

[5] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: Don't Reach for the Stars! arXiv:2301.03988 [cs]

[6] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4 (July 2018), 81:1–81:37. https://doi.org/10.1145/3212695

[7] Miltiadis Allamanis, Sheena Panthaplackel, and Pengcheng Yin. 2024. Unsupervised Evaluation of Code LLMs with Round-Trip Correctness. arXiv:2402.08699 [cs]

[8] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. 2022. Efficient Large Scale Language Modeling with Mixtures of Experts. https://doi.org/10.48550/arXiv.2112.10684 arXiv:2112.10684 [cs]

[9] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where Is the Bug and How Is It Fixed? An Experiment with Practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/3106237.3106255

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. Curran Associates, Inc., 1877–1901.

[11] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. https://doi.org/10.48550/arXiv.2303.12712 arXiv:2303.12712 [cs]

[12] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. https://doi.org/10.48550/arXiv.2006.12641 arXiv:2006.12641 [cs]

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374

[14] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. https://doi.org/10.48550/arXiv.2304.05128 arXiv:2304.05128 [cs]

[15] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 47 (2021), 1943–1959. https://doi.org/10.1109/tse.2019.2940179

[16] Alain Désilets and Matthieu Hermet. 2009. Using Automatic Roundtrip Translation to Repair General Errors in Second Language Writing. In *Machine Translation Summit XII*. 9.

[17] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Mitigating LLM Hallucinations in Code Generation Tasks via Iterative Grounding. https://doi.org/10.48550/arXiv.2401.01701 arXiv:2401.01701 [cs]

[18] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1469–1481. https://doi.org/10.1109/

icse48619.2023.00128

[19] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *11th International Conference on Learning Representations (ICLR)*. 1–26.

[20] Gianluca Grimaldi and Bruno Ehrler. 2023. AI et al.: Machines Are about to Change Scientific Publishing Forever. *ACS Energy Letters* 8, 1 (2023), 878–880. https://doi.org/10.1021/acsenergylett.2c02828

[21] Anastasiia Grishina, Https://Orcid.Org/0000-0003-3139-0200, View Profile, Vadim Liventsev, Https://Orcid.Org/0000-0002-6670-6909, View Profile, Aki Härmä, Https://Orcid.Org/0000-0002-2966-3305, View Profile, Leon Moonen, Https://Orcid.Org/0000-0002-1761-6771, and View Profile. 2025. Fully Autonomous Programming Using Iterative Multi-Agent Debugging with Large Language Models. *ACM Transactions on Evolutionary Learning and Optimization* 5, 1 (March 2025), 1–37. https://doi.org/10.1145/3719351

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. https://doi.org/10.48550/arXiv.2009.08366 arXiv:2009.08366 [cs]

[23] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[24] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive Code Representation Learning. In *Conference on Empirical Methods in Natural Language Processing: EMNLP 2021*. Association for Computational Linguistics, 5954–5971. https://doi.org/10.18653/v1/2021.emnlp-main.482

[25] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442. https://doi.org/10.1109/icse48619.2023.00125

[26] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. https://doi.org/10.1109/icse43902.2021.00107 arXiv:2103.00073 [cs]

[27] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *23th International Symposium on Software Testing and Analysis (ISSTA) (ISSTA 2014)*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[28] Sungmin Kang and Shin Yoo. 2022. Language Models Can Prioritize Patches for Practical Program Patching. In *3rd International Workshop on Automated Program Repair @ ICSE 2022 (APR '22)*. ACM, 8–15. https://doi.org/10.1145/3524459.3527343

[29] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. https://doi.org/10.48550/arXiv.2001.08361 arXiv:2001.08361 [cs, stat]

[30] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and Harm de Vries. 2023. The Stack: 3 TB of Permissively Licensed Source Code. *Transactions on Machine Learning Research* (2023).

[31] Sophia D. Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. In *Deep Learning for Code Workshop @ ICLR 2022*. 1–11.

[32] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Phrase-Based & Neural Unsupervised Machine Translation. In *Conference on Empirical Methods in Natural Language Processing: EMNLP 2018*. Association for Computational Linguistics, 5039–5049. https://doi.org/10.18653/v1/d18-1549

[33] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs]

[34] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[35] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[36] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: May the Source Be with You! arXiv:2305.06161 [cs]

[37] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Companion of the SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, Vancouver BC Canada, 55–56. https://doi.org/10.1145/3135932.3135941

[38] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. https://doi.org/10.48550/arXiv.2404.00971 arXiv:2404.00971 [cs]

[39] Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully Autonomous Programming with Large Language Models. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 1146–1155. https://doi.org/10.1145/3583131.3590481

[40] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Neural Information Processing Systems Track on Datasets and Benchmarks*. 1–16.

[41] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 101–114. https://doi.org/10.1145/3395363.3397369

[42] Sneha Mehta, Bahareh Azarnoush, Boris Chen, Avneesh Saluja, Vinith Misra, Ballav Bihani, and Ritwik Kumar. 2020. Simplify-Then-Translate: Automatic Preprocessing for Black-Box Machine Translation. https://doi.org/10.48550/arXiv.2005.11197 arXiv:2005.11197 [cs]

[43] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan. 2018), 1–24. https://doi.org/10.1145/3105906

[44] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2015. The Design Space of Bug Fixes and How Developers Navigate It. *IEEE Transactions on Software Engineering* 41, 1 (Jan. 2015), 65–81. https://doi.org/10.1109/tse.2014.2357438

[45] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *31st International Joint Conference on Artificial Intelligence*. 5546–5555. https://doi.org/10.24963/ijcai.2022/775

[46] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs]

[47] OpenAI, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, A. J. Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, Alex Renzin, Alex Tachard Passos, Alexander Kirillov, Alexi Christakis, Alexis Conneau, Ali Kamali, Allan Jabri, Allison Moyer, Allison Tam, Amadou Crookes, Amin Tootoochian, Amin Tootoonchian, Ananya Kumar, Andrea Vallone, Andrej Karpathy, Andrew Braunstein, Andrew Cann, Andrew Codispoti, Andrew Galu, Andrew Kondrich, Andrew Tulloch, Andrey Mishchenko, Angela Baek, Angela Jiang, Antoine Pelisse, Antonia Woodford, Anuj Gosalia, Arka Dhar, Ashley Pantuliano, Avi Nayak, Avital Oliver, Barret Zoph, Behrooz Ghorbani, Ben Leimberger, Ben Rossen, Ben Sokolowsky, Ben Wang, Benjamin Zweig, Beth Hoover, Blake Samic, Bob McGrew, Bobby Spero, Bogo Giertler, Bowen Cheng, Brad Lightcap, Brandon Walkin, Brendan Quinn, Brian Guarraci, Brian Hsu, Bright Kellogg, Brydon Eastman, Camillo Lugaresi, Carroll Wainwright, Cary Bassin, Cary Hudson, Casey Chu, Chad Nelson, Chak Li, Chan Jun Shern, Channing Conger, Charlotte Barette, Chelsea Voss, Chen Ding, Cheng Lu, Chong Zhang, Chris Beaumont, Chris Hallacy, Chris Koch, Christian Gibson, Christina Kim, Christine Choi, Christine McLeavey, Christopher Hesse, Claudia Fischer, Clemens Winter, Coley Czarnecki, Colin Jarvis, Colin Wei, Constantin Koumouzelis, Dane Sherburn, Daniel Kappler, Daniel Levin, Daniel Levy, David Carr, David Farhi, David Mely, David Robinson, David Sasaki, Denny Jin, Dev Valladares, Dimitris Tsipras, Doug Li, Duc Phong Nguyen, Duncan Findlay, Edede Oiwoh, Edmund Wong, Ehsan Asdar, Elizabeth Proehl, Elizabeth Yang, Eric Antonow, Eric Kramer, Eric Peterson, Eric Sigler, Eric Wallace, Eugene Brevdo, Evan Mays, Farzad Khorasani, Felipe Petroski Such, Filippo Raso, Francis Zhang, Fred von Lohmann, Freddie Sulit, Gabriel Goh, Gene Oden, Geoff Salmon, Giulio Starace, Greg Brockman, Hadi Salman, Haiming Bao, Haitang Hu, Hannah Wong, Haoyu Wang, Heather Schmidt, Heather Whitney, Heewoo Jun, Hendrik Kirchner, Henrique Ponde de Oliveira Pinto, Hongyu Ren, Huiwen Chang, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian O'Connell, Ian Osband, Ian Silber, Ian Sohl, Ibrahim Okuyucu, Ikai Lan, Ilya Kostrikov, Ilya Sutskever, Ingmar Kanitscheider, Ishaan Gulrajani, Jacob Coxon, Jacob Menick, Jakub Pachocki, James Aung, James Betker, James Crooks, James Lennon, Jamie Kiros, Jan Leike, Jane Park, Jason Kwon, Jason Phang, Jason Teplitz, Jason Wei, Jason Wolfe, Jay Chen, Jeff Harris, Jenia Varavva, Jessica Gan Lee, Jessica Shieh, Ji Lin, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joanne Jang, Joaquin Quinonero Candela, Joe Beutler, Joe Landers, Joel Parish, Johannes Heidecke, John Schulman, Jonathan Lachman, Jonathan McKay, Jonathan Uesato, Jonathan Ward, Jong Wook Kim, Joost Huizinga, Jordan Sitkin, Jos Kraaijeveld, Josh Gross, Josh Kaplan, Josh Snyder, Joshua Achiam, Joy Jiao, Joyce Lee, Juntang Zhuang, Justyn Harriman, Kai Fricke, Kai Hayashi, Karan Singhal, Katy Shi, Kavin Karthik, Kayla Wood, Kendra Rimbach, Kenny Hsu, Kenny Nguyen, Keren Gu-Lemberg, Kevin Button, Kevin Liu, Kiel Howe, Krithika Muthukumar, Kyle Luther, Lama Ahmad, Larry Kai, Lauren Itow, Lauren Workman, Leher Pathak, Leo Chen, Li Jing, Lia Guy, Liam Fedus, Liang Zhou, Lien Mamitsuka, Lilian Weng, Lindsay McCallum, Lindsey Held, Long Ouyang, Louis Feuvrier, Lu Zhang, Lukas Kondraciuk, Lukasz Kaiser, Luke Hewitt, Luke Metz, Lyric Doshi, Mada Aflak, Maddie Simens, Madelaine Boyd, Madeleine Thompson, Marat Dukhan, Mark Chen, Mark Gray, Mark Hudnall, Marvin Zhang, Marwan Aljubeh, Mateusz Litwin, Matthew Zeng, Max Johnson, Maya Shetty, Mayank Gupta, Meghan Shah, Mehmet Yatbaz, Meng Jia Yang, Mengchao Zhong, Mia Glaese, Mianna Chen, Michael Janner, Michael Lampe, Michael Petrov, Michael Wu, Michele Wang, Michelle Fradin, Michelle Pokrass, Miguel Castro, Miguel Oom Temudo de Castro, Mikhail Pavlov, Miles Brundage, Miles Wang, Minal Khan, Mira Murati, Mo Bavarian,

Molly Lin, Murat Yesildal, Nacho Soto, Natalia Gimelshein, Natalie Cone, Natalie Staudacher, Natalie Summers, Natan LaFontaine, Neil Chowdhury, Nick Ryder, Nick Stathas, Nick Turley, Nik Tezak, Niko Felix, Nithanth Kudige, Nitish Keskar, Noah Deutsch, Noel Bundick, Nora Puckett, Ofir Nachum, Ola Okelola, Oleg Boiko, Oleg Murk, Oliver Jaffe, Olivia Watkins, Olivier Godement, Owen Campbell-Moore, Patrick Chao, Paul McMillan, Pavel Belov, Peng Su, Peter Bak, Peter Bakkum, Peter Deng, Peter Dolan, Peter Hoeschele, Peter Welinder, Phil Tillet, Philip Pronin, Philippe Tillet, Prafulla Dhariwal, Qiming Yuan, Rachel Dias, Rachel Lim, Rahul Arora, Rajan Troll, Randall Lin, Rapha Gontijo Lopes, Raul Puri, Reah Miyara, Reimar Leike, Renaud Gaubert, Reza Zamani, Ricky Wang, Rob Donnelly, Rob Honsby, Rocky Smith, Rohan Sahai, Rohit Ramchandani, Romain Huet, Rory Carmichael, Rowan Zellers, Roy Chen, Ruby Chen, Ruslan Nigmatullin, Ryan Cheu, Saachi Jain, Sam Altman, Sam Schoenholz, Sam Toizer, Samuel Miserendino, Sandhini Agarwal, Sara Culver, Scott Ethersmith, Scott Gray, Sean Grove, Sean Metzger, Shamez Hermani, Shantanu Jain, Shengjia Zhao, Sherwin Wu, Shino Jomoto, Shirong Wu, Shuaiqi, Xia, Sonia Phene, Spencer Papay, Srinivas Narayanan, Steve Coffey, Steve Lee, Stewart Hall, Suchir Balaji, Tal Broda, Tal Stramer, Tao Xu, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Cunninghman, Thomas Degry, Thomas Dimson, Thomas Raoux, Thomas Shadwell, Tianhao Zheng, Todd Underwood, Todor Markov, Toki Sherbakov, Tom Rubin, Tom Stasi, Tomer Kaftan, Tristan Heywood, Troy Peterson, Tyce Walters, Tyna Eloundou, Valerie Qi, Veit Moeller, Vinnie Monaco, Vishal Kuo, Vlad Fomenko, Wayne Chang, Weiyi Zheng, Wenda Zhou, Wesam Manassra, Will Sheu, Wojciech Zaremba, Yash Patil, Yilei Qian, Yongjik Kim, Youlong Cheng, Yu Zhang, Yuchen He, Yuchen Zhang, Yujia Jin, Yunxing Dai, and Yury Malkov. 2024. GPT-4o System Card. https://doi.org/10.48550/arXiv.2410.21276 arXiv:2410.21276 [cs]

[48] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (Feb. 2025), 1–28. https://doi.org/10.1145/3697010

[49] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A Method for Automatic Evaluation of Machine Translation. In *40th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 311–318. https://doi.org/10.3115/1073083.1073135

[50] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/sp46214.2022.9833571

[51] Dino Pedreschi, Luca Pappalardo, Ricardo Baeza-Yates, Albert-Laszlo Barabasi, Frank Dignum, Virginia Dignum, Tina Eliassi-Rad, Fosca Giannotti, Janos Kertesz, Alistair Knott, Yannis Ioannidis, Paul Lukowicz, Andrea Passarella, Alex Sandy Pentland, John Shawe-Taylor, and Alessandro Vespignani. 2023. Social AI and the Challenges of the Human-AI Ecosystem. https://doi.org/10.48550/arXiv.2306.13723 arXiv:2306.13723 [cs]

[52] Alec Radford, Jeff Wu, Rewon Child, D. Luan, Dario Amodei, and I. Sutskever. 2019. Language Models Are Unsupervised Multitask Learners.

[53] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

[54] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *38th International Conference on Software Engineering (ICSE)*. ACM, 428–439. https://doi.org/10.1145/2884781.2884848

[55] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: A Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297

[56] Baptiste Roziere, Marie-Anne Lachaux, Guillaume Lample, and Lowik Chanussot. 2020. Unsupervised Translation of Programming Languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Vol. 33. 20601–20611. https://doi.org/10.48550/arxiv.2006.03511

[57] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI Tasks with ChatGPT and Its Friends in Hugging Face. arXiv:2303.17580 [cs]

[58] Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: An Autonomous Agent with Dynamic Memory and Self-Reflection. https://doi.org/10.48550/arXiv.2303.11366 arXiv:2303.11366 [cs]

[59] André Silva, João F. Ferreira, He Ye, and Martin Monperrus. 2023. MUFIN: Improving Neural Repair Models with Back-Translation. https://doi.org/10.48550/arXiv.2304.02301 arXiv:2304.02301 [cs]

[60] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *IEEE/ACM International Workshop on Automated Program Repair (APR)*. 23–30. https://doi.org/10.1109/apr59189.2023.00012

[61] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 130–140. https://doi.org/10.1109/saner.2018.8330203 arXiv:1801.06393 [cs]

[62] Felix Stahlberg. 2020. Neural Machine Translation: A Review. *Journal of Artificial Intelligence Research* 69 (Oct. 2020), 343–418. https://doi.org/10.1613/jair.1.12007

[63] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 27. Curran Associates, Inc., 1–9.

[64] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Vasileios Mavroeidis. 2023. The FormAI Dataset: Generative AI in Software Security Through the Lens of Formal Verification. https://doi.org/10.48550/arXiv.2307.02192 arXiv:2307.02192 [cs]

[65] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (Sept. 2019), 19:1–19:29. https://doi.org/10.1145/3340544

[66] 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 *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 30. Curran, 1–11.

[67] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. https://doi.org/10.48550/arXiv.2305.07922 arXiv:2305.07922 [cs]

[68] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[69] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research* (June 2022), 1–30.

[70] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. 2000. *Experimentation in Software Engineering: An Introduction*. Number 6 in The Kluwer International Series in Software Engineering. Springer.

[71] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494. https://doi.org/10.1109/icse48619.2023.00129

[72] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2022)*. ACM, 959–971. https://doi.org/10.1145/3540250.3549101

[73] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. https://doi.org/10.48550/arXiv.2301.13246 arXiv:2301.13246 [cs]

[74] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT. https://doi.org/10.48550/arXiv.2304.00385 arXiv:2304.00385 [cs]

[75] Eran Yahav. 2023. Towards AI-Driven Software Development: Challenges and Lessons from the Field (Keynote). In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 1–1. https://doi.org/10.1145/3611643.3633451

[76] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. 2020. A Survey of Deep Learning Techniques for Neural Machine Translation. arXiv:2002.07526 [cs]

[77] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. *Journal of Systems and Software* 171 (Jan. 2021), 110825. https://doi.org/10.1016/j.jss.2020.110825 arXiv:1805.03454 [cs]

[78] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-Trained Models for Program Understanding and Generation. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 39–51. https://doi.org/10.1145/3533767.3534390

[79] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. https://doi.org/10.48550/arXiv.2301.03270 arXiv:2301.03270 [cs]

[80] Terry Yue Zhuo, Qiongkai Xu, Xuanli He, and Trevor Cohn. 2023. Rethinking Round-Trip Translation for Machine Translation Evaluation. In *Findings of the Association for Computational Linguistics: ACL 2023*. Association for Computational Linguistics, 319–337. https://doi.org/10.18653/v1/2023.findings-acl.22

Received 22 March 2024; revised 7 October 2025; accepted 9 October 2025