Template-Guided Program Repair in the Era of Large Language Models

Kai Huang
Technical University of Munich
Germany
huangkevinapr@outlook.com

Jian Zhang*
Nanyang Technological University
Singapore
jian_zhang@ntu.edu.sg

Xiangxin Meng
Beihang University
China
mengxx@buaa.edu.cn

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

***Abstract*—Recent advancements in automated program repair (APR) have been significantly driven by the application of Large Language Models (LLMs). In particular, the integration of LLMs with traditional template-based repair methods has demonstrated effective outcomes. Despite this, the synergy between the strengths of traditional methods and LLMs remains underexploited. This oversight originates from the indiscriminate use of templates and their insufficient coverage. Also, using small-scale LLMs within the zero-shot learning context proves to be suboptimal.**

To alleviate the limitations, we propose NTR (Neural Template Repair), a two-stage repair framework including template selection and patch generation, both of which are under the fine-tuning paradigm. In the template selection phase, we formulate it as a multiclass classification problem and fine-tune million-level LLMs for better selecting possible templates. During the patch generation phase, we leverage the chosen templates as probable directions (e.g., 'Mutate Conditional Expression') to guide the fine-tuning process of LLMs at the billion-level scale for precise patch creation. Moreover, we incorporate a unique template to signify the absence of a suitable template and employ a probability-based prioritization of templates, thereby optimizing patch generation. This framework not only effectively addresses template mismatch issues, but also enables the billion-level LLMs to explore the patch space more efficiently, despite the GPU memory constraints.

We evaluate NTR with different foundational models on Defects4J V1.2 and HumanEval-Java, the framework consistently demonstrates significant effectiveness. When utilizing StarCoder as the foundational model for patch generation, NTR fixes 128 and 129 bugs in Defects4J and HumanEval, outperforming the best baseline APR tool by 14 and 59 bugs. With the larger CodeLlama model, the fixed bugs rise to 139 and 136, respectively, exceeding the baseline by 25 and 66 bugs. Notably, the performance stems not only from the foundational models but also benefits greatly from our NTR framework. Specifically, NTR's implementation with StarCoder and CodeLlama leads to 22 and 23 additional fixes, which is beyond what the models achieve on their own. This emphasizes the success of our new perspective on utilizing templates to unlock the bug-fixing potential of LLMs.

***Index Terms*—Automated Program Repair, Large Language Models, Fine-Tuning, Repair Template**

I. Introduction

Automated Program Repair (APR) techniques have been under development for nearly two decades, spawning several methodological paths such as search-based [1], constraint-based [2], template-based [3], and learning-based [4] APR techniques. In this field, learning-based APR techniques have achieved considerable progress, as highlighted in recent studies [4].

* Corresponding author: Jian Zhang (jian_zhang@ntu.edu.sg).

Recently, the advent of Large Language Models (LLMs) presents new opportunities for the APR research. LLMs typically undergo an extensive pre-training phase, which enables them to acquire the rich domain knowledge for a wide range of downstream tasks. Inspired by this, researchers have started to explore the capabilities of LLMs for APR [5]–[13]. Unlike earlier learning-based APR efforts [14]–[19] that rely on traditional neural models, recent works [8]–[10], [13], [20] have adopted LLMs as foundational models, and achieved greatly improved results.

Among the LLM-based approaches, the incorporation of template has shown impressive performance as evident by AlphaRepair [9] and GAMMA [10], which can represent the state-of-the-art APR work. These tools adopt the zero-shot learning paradigm, which allows the model to directly predict the correct code for areas masked by predefined repair templates in the buggy code. However, existing template-based LLM solutions for APR still suffer from three primary drawbacks. **1) Indiscriminate template use**: Initiatives like AlphaRepair and GAMMA typically do not prioritize among templates during the selection process, often employing all available templates indiscriminately. Specifically, GAMMA utilizes an AST-based matching method to choose repair templates, which risks prioritizing incorrect templates over suitable ones. This can lead the model to miss opportunities of correct patch synthesis and exacerbate the patch overfitting problem. **2) Inadequate template coverage**: The effectiveness of template-based APR tools is inherently limited by their template scope [21]. For instance, GAMMA [10] utilizes a set of 13 repair templates derived from TBar [3] for masking code in the patch synthesis process. If a bug's expected repair behavior falls outside these predefined templates, these tools can struggle to generate the correct patch. **3) Limitations of small-scale LLMs under zero-shot paradigm**: Current approaches rely on zero-shot learning with smaller LLMs (e.g., CodeBERT-125M), while the potential of billion-level LLMs with templates under the fine-tuning paradigm remains untapped. In fact, previous studies have shown that simply fine-tuning and using larger LLMs can significantly improve repair capabilities [6], [12]. Unfortunately, the infilling code mask methods are largely confined to masked language modeling-based (MLM-based) LLMs, which restricts their applicability to decoder-only models without infilling capabilities, such as CodeLlama [22].

Therefore, in the era of LLMs, it is necessary to reconsider how templates can be more effectively integrated into large-scale LLMs to further enhance their repair capabilities. We present the following key insights to mitigate the above limitations. **Insight 1**: Inspired by the template-based approach in TRANSFER [23], training a model to rank templates has proven to be an effective strategy. Particularly in the context of LLMs, we can develop a template ranking model that leverages the code comprehension strengths of LLMs presents a promising avenue. **Insight 2**: Previous work [21] has shown that the Neural Machine Translation (NMT) workflow circumvents the template coverage issue by conceptualizing the repair task as a translation rather than an infilling task. In this workflow, we can design a special template that represents bug fixes beyond traditional template set, and adopt the NMT model to learn these complex repair behaviors (e.g., multi-hunk bugs). **Insight 3**: Recent studies [7], [12] suggest that applying NMT fine-tuning strategies can overcome the limitations associated with the zero-shot learning paradigm. Given that LLMs are not inherently tailored for repair tasks, we can impart bug-fixing knowledge of templates into LLMs during fine-tuning. Moreover, the flexibility of the NMT workflow allows it to adapt to various model architectures, without being restricted to MLM-based LLMs [7], [12].

Based on above insights, we propose a novel template-based framework, namely Neural Template Repair (NTR). NTR divides the repair task into two phases: **template selection** and **patch generation**. **1)** In the template selection phase, NTR formulates it as the multiclass classification problem [24] and fine-tunes lightweight LLMs as the template selection model to perform template ranking (**Insight 1**). It takes the buggy method as input and predicts the most appropriate repair templates for repairing it. **2)** In the patch generation phase, we adopt the NMT workflow and format the selected templates as guidance. We select an LLM with billions of parameters and fine-tune it under the NMT workflow (**Insight 3**). During fine-tuning, we introduce symbols to denote the buggy code, the template, and the fixed code. This enables LLMs to synthesize patches in an autoregressive manner, naturally following the logic of the selected templates. To address the template coverage issue, we set up a special template, $OtherTemplate$, to capture repair behaviors beyond the scope of existing templates (**Insight 2**). Furthermore, when performing inference, the prioritization of selected templates allows NTR to iteratively produce a broader array of candidate patches than approaches that do not leverage templates. This approach allows for better exploration of the patch space guided by repair templates and alleviates the limitations present in previous work. As an added benefit, the selection-then-generation framework empowers large-scale LLMs to expand the patch space even with small beam sizes. This adaptation is crucial because setting a large beam size, as done in traditional (non-pre-trained model based) work [16], [25], [26], becomes impractical due to the billions of parameters in these models and the limitations of GPU memory [7].

In summary, the main contributions of this paper are as follows:

- **Technique.** We introduce NTR, a novel framework that combines the strengths of both templates and large-scale LLMs, by guiding LLMs through a two-phased approach of template selection and patch generation. To the best of our knowledge, StarCoder-15B or CodeLlama-70B utilized in NTR, represents the largest LLM applied to fine-tuned APR research so far. Our work showcases, for the first time, a promising path toward integrating these large-scale LLMs with templates.
- **Extensive Study.** We performed extensive evaluations on Defects4J V1.2 and HumanEval-Java. The results demonstrate that NTR outperforms previous APR tools and further boosts the repair capabilities of LLMs. For instance, with StarCoder as the foundational model, NTR successfully repairs 128 and 129 bugs in Defects4J and HumanEval, respectively, surpassing the top baseline APR tool (ChatRepair) and LLM (InCoder) by 14 and 59 bugs. Similarly, utilizing the more substantial CodeLlama model, NTR fixes 139 and 136 bugs in Defects4J and HumanEval, respectively, outperforming the best baseline by 25 and 66 bugs. Notably, NTR's application with StarCoder fixed 22 more bugs than the foundational model, marking a 9.36% improvement. Similarly, its use of CodeLlama led to fixing 23 more bugs, achieving a 9.13% improvement over the foundational model.
- **Open Science.** To promote open science, we have released StarCoder-15B and CodeLlama-70B, two LLMs that have been fine-tuned with the NTR framework and basic NMT fine-tuning strategy. We encourage future researchers to leverage these foundational models to develop more powerful APR tools. Our source code and data are available at: https://sites.google.com/view/neuraltemplaterepair.

II. APPROACH

This section will describe how NTR combines template-based and learning-based technical paths to build a new program repair workflow. As shown in Figure 1, the workflow of NTR is divided into two main parts, i.e., model fine-tuning and model inference, which is in line with most of the supervised learning-based APR workflows [14]–[19], [21], [25]–[29]. In particular, NTR has two modules (phases) on design choice, namely **template selection** and **patch generation**. Briefly speaking, NTR first trains a template selection model to prioritize appropriate repair templates for the buggy code, and then trains a patch generation model to perform patch synthesis under the guidance of templates. The design of this two-phase repair framework for NTR is the main focus of this paper, which we will elaborate on in the following sections.

A. Model Fine-Tuning

This section describes how we train the template selection (or template ranking) model and the patch generation model.

1) Template Selection Model Training: Let's first review the workflow of template-based APR techniques. Typically, this approach employs a two-phase repair strategy that includes template extraction and patch generation. Earlier template-based works [3], [30]–[32] did not consider the prioritization of repair

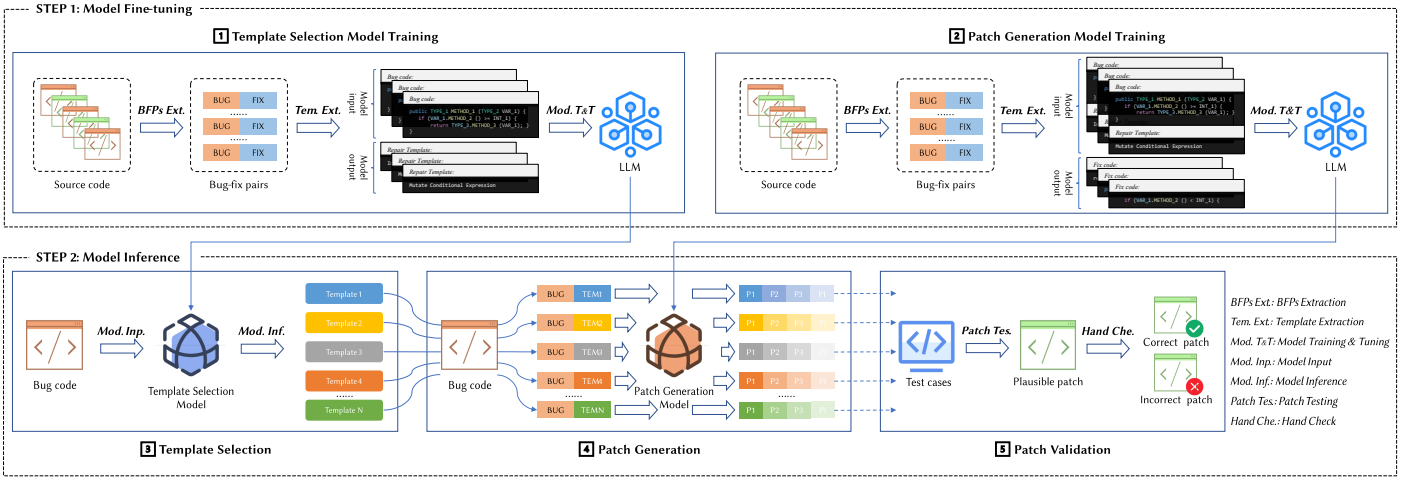

Fig. 1: The workflow for Neural Template Repair.

template usage. They used all repair templates sequentially to generate patches. However, the original order of templates can have a negative impact on the effectiveness of patch generation. In the worst case, such a practice could leave the most possible template at the end of the patch space, which would fail to generate the correct patch due to the limited budget of patch validation [23]. Subsequently, some works tried to optimize template selection by building probabilistic models [33]–[35] or simple MLP models [23]. These models facilitate the selection of repair templates that are more likely to be correct, offering a promising direction. Given that these neural models must learn from scratch using training data, their performance could be limited. Meanwhile, in the current era, LLMs have shown extraordinary capabilities in code understanding and generation tasks [24], [36]. Hence, a practical approach to enhancement involves leveraging the comprehension abilities of LLMs to aid in the selection of suitable repair templates.

Specifically, we adopt the NMT fine-tuning paradigm to train a specialized template selection model, denoted as $\mathcal{M}_{\text{template}}$. We outline the concrete steps as follows.

BFP Extraction. Before model training, we construct training samples by extracting data in the form of Bug-Fix Pairs (BFPs) [14] from a collection of bug-fix histories. Here, we used the Transfer dataset collected by Meng et al. [23] as the training corpus from which we extracted method-level BFPs. We chose the Transfer dataset because it has been successfully applied to template-based APR efforts [21], [23], which could be a robust foundation for deploying NTR's framework.

Let $\mathcal{D}$ denote a dataset, which comprises N pairs of bug code x_i and fix code y_i:

$$\mathcal{D} = \{(x_i, y_i) \mid x_i \text{ bug code}, y_i \text{ fix code}, 1 \leq i \leq N\} \quad (1)$$

Here, we have a specific instance: $\mathcal{D}_{\text{Transfer}}$ for the Transfer dataset. This dataset serve as the basis for extracting templates and training our template selection model.

TABLE I: 15 repair templates (No.1-15) from previous work and 1 special template (No.16) from NTR settings.

No.	Repair Templates	No.	Repair Templates
1	Insert Cast Checker	9	Mutate Class Instance Creation
2	Insert Range Checker	10	Mutate Integer Division Operation
3	Insert Null Pointer Checker	11	Mutate Operators
4	Insert Missed Statement	12	Mutate Return Statement
5	Mutate Conditional Expression	13	Mutate Variable
6	Mutate Data Type	14	Move Statement
7	Mutate Literal Expression	15	Remove Buggy Statement
8	Mutate Method Invocation Expression	16	Other Template

Template Extraction. After obtaining BFPs, it is necessary to deduce fix templates based on specific bug-fixing behaviors, that is, code changes. To achieve this, we perform template extraction based on state-of-the-art template-based APR works TBar [3], TRANSFER [23], and TENURE [21].

Specifically, on the $\mathcal{D}_{\text{Transfer}}$ dataset [23], we initially utilized the 15 repair templates as shown in Table I (more details please see TRANSFER [23]), provided by previous works [3], [23], represented as $\mathcal{T}_{\text{Transfer}}$, and conducted template extraction using the SC4FT [37] tool developed by Meng et al. [21], [23]. We selected these 15 repair templates because they encompass most of the bug repair behaviors, and prior research [3], [21], [23] has employed these templates to achieve advanced results.

However, due to the diverse nature of software bugs, creating a comprehensive set of templates that covers all possible bugs is unfeasible. Therefore, previous methods like GAMMA [10] relying solely on a predefined set of templates might not address a significant number of complex bugs, including multi-hunk bugs. To tackle the challenge, during the extraction process, we designate "*Other Template*" to label repair behaviors beyond the scope of the 15 repair templates mentioned above. In addition, this special template also contains multi-hunk bug fixing behavior. Thus it can be used to mitigate the template coverage problem [21] and allow the model to gain repair capability of multi-hunk bugs.

Model Training & Tuning. We denote the training data

as follows: $\mathcal{X}$ represents the set of bug code samples, and $\mathcal{T}$ represents the set of repair templates. Each training sample, expressed as (x_i, t_i), consists of a bug code x_i and its corresponding repair template t_i. The core objective of the template selection model, referred to as $\mathcal{M}_{\text{template}}$, is to predict the appropriate repair template t_i based on a given bug code x_i. This objective can be expressed as: $\mathcal{M}_{\text{template}}(x_i) = t_i$.

The fine-tuning objective aims to minimize the cross-entropy loss $\mathcal{L}_{\text{template}}$ for the template selection model $\mathcal{M}_{\text{template}}$, optimizing the model's parameters θ by reducing the discrepancy between the predicted templates $\hat{t}_i$ and the actual templates t_i for all bug code snippets x_i. This is formulated as:

$$\min_{\theta} \mathcal{L}_{\text{template}}(\theta) = -\sum_{i}\sum_{j} t_{ij} \log(\hat{t}_{ij}), \tag{2}$$

where $\hat{t}_{ij}$ denotes the model's predicted probability that label j is the correct classification for bug code x_i, and t_{ij} is a binary indicator of whether label j is the correct template for the given instance.

During the training of the template selection model, we use small-scale LLMs such as CodeT5 as the foundation model. The rationale for the choice of million-level CodeT5 that has an encoder instead of a billion-level LLM like decoder-only StarCoder is as follows. Template selection, fundamentally a task of program understanding, involves predicting the appropriate repair template for given buggy code. Findings from a prior study [24] indicate that GPT-like models (Decoder-only), including GPT-C [38] and CodeGPT [39], underperform in classification tasks such as this. Conversely, Encoder-Decoder models like CodeT5 [40] have demonstrated superior performance in classification challenges according to the same study [24]. Also, our preliminary experiments confirmed that StarCoder performs poorly on this type of task. Therefore, we use CodeT5 for demonstration purposes in this work.

The template selection task can be viewed as a multi-class classification problem, where each template serves as a class. Given the buggy code x_i and the model $\mathcal{M}_{\text{CodeT5}}$, the vector representation of the selection model be described as follows:

$$r_i = \mathcal{M}_{\text{CodeT5-[Enc]}}(x_i), \tag{3}$$

where [Enc] is the prefix in front of the input and the corresponding hidden state can represent the semantics of the whole buggy function based on the attention mechanism. A softmax function is then applied to these logits to derive a probability distribution over all repair templates: $\hat{p}(t|x_i) = \text{softmax}(r_i)$. Here, $\hat{p}(t|x_i)$ denotes the predicted probability distribution over repair templates t for the given buggy code x_i. The repair template with the highest probability is selected as the prediction: $\hat{t}_i = \arg\max_t \hat{p}(t|x_i)$.

It is true that a model can not always perfectly predict the most appropriate template. However, our model can also be used for template prioritization based on probabilities, allowing the inclusion of correct patch at the patch generation phase.

2) Patch Generation Model Training: When generating patches, existing template-based LLM approaches including AlphaRepair [9] and GAMMA [10] transform fix templates into masked donor code, and directly use million-level LLMs to fill the masks with zero-shot learning. This practice could suffer from the limitations of the models' small scale and misalignment between the code generation and program repair. Typically, the NMT fine-tuning paradigm of LLMs can address the problems. As a representative built on traditional neural model, TENURE [21] follows the NMT workflow while considering templates. Different from the above work, TENURE jointly generates a template along with the fixed code during training and inference. Theoretically, the template generation can be largely affected by the joint cross-entropy, since the fixed code is generally much longer than a template (see Table I). This one-step strategy potentially hampers the model's ability to accurately select the most appropriate template, let alone effectively prioritize among templates, thereby diminishing the benefits of template-based guidance. We will further analyze this problem in the ablation study of Section IV-B.

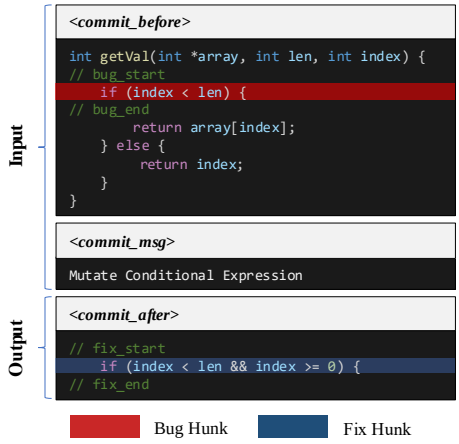

Fig. 2: The input and output of the patch generation model.

Therefore, NTR implements a two-phase repair strategy that distinctively coordinates template selection and patch generation by optimizing them independently. In the patch generation phase, we specifically incorporate the chosen repair templates alongside the buggy code as inputs, while designating the fixed code solely as the output. This dual-input method makes the patch synthesis process aligns more closely with the guidance offered by the selected templates.

Figure 2 illustrates the path generation process. In the fine-tuning phase, we format the buggy code, the template and the fixed code as: *<commit_before> bug_method <commit_msg> repair_template <commit_after> fix_code*. This approach is inspired by the pre-training methodologies of existing LLMs such as StarCoder, which learn the commit behaviors from open-source projects in this manner. Since bug fixing often involves commits in practice, the format is naturally suited for fine-tuning as it closely matches the structure used during pre-training.

Model Training & Tuning. Based on both repair templates and the buggy code, the data $\mathcal{D}$ extracted from STEP 1&2 (BFP&Template Extraction) are restructured to align with the

requirements of the patch generation task. Let $\mathcal{X} \in \mathcal{D}$ represent the set of buggy code samples, $\mathcal{T}$ denote the set of repair templates, and $\mathcal{Y} \in \mathcal{D}$ signify the set of fix code samples. Each training sample, denoted as (x_i, t_i, y_i), consists of a buggy code x_i, its corresponding repair template t_i. The objective of the patch generation model, referred to as $\mathcal{M}_{\text{patch}}$, is to autoregressively synthesize the appropriate fix code $\hat{y_i}$ given x_i and t_i. This can be represented as: $\hat{y_i} = \mathcal{M}_{\text{patch}}(x_i, t_i)$.

During fine-tuning, the objective is to optimize the model's parameters Θ by maximizing the log-likelihood of the autoregressive generation process across all training samples. This involves calculating the likelihood of each token y_{ij} in the predicted fix code $\hat{y}_i$, based on the actual fix code y_i, up to the current token, alongside the bug code x_i, and the selected repair template t_i. The aggregated log-likelihood function, representing the sum of log probabilities for correctly predicting each token in the sequence, is given by:

$$\min_{\Theta} \mathcal{L}(\Theta) = -\sum_{i} \sum_{j} \log(\hat{p}(y_{ij}|y_{i,<j}, x_i, t_i)) \tag{4}$$

Here, $\hat{p}(y_{ij}|y_{i,<j}, x_i, t_i)$ denotes the model's predicted probability of token y_{ij} given the preceding tokens $y_{i,<j}$ in the actual fix code, under the condition of the bug code x_i and the repair template t_i. The goal of fine-tuning is to maximize this function, thereby enhancing the model's ability to accurately predict the next token in the sequence.

We choose StarCoder-15B/CodeLlama-70B as the foundation model for fine-tuning. There are two main reasons for this decision. Fundamentally, program repair is inherently a code generation task, and based on prior studies [6], [12], larger LLMs tend to exhibit more powerful repair capabilities. In fact, it is revealed that LLMs show a clear pattern called the emergent capability [41]. That is, the performance is near-random until a certain critical threshold of scale is reached (e.g., 10B), after which performance increases to substantially above random. Hence, we select billion-level LLMs for the patch generation task. This enable it to better deal with the unseen and complex bugs in real-world projects.

Through template-guided training, NTR's patch generation model acquires repair behaviors aligned with the diverse directions outlined in the templates. Importantly, this training approach enables conditional generation, allowing the large-scale LLMs to extend beyond the confines of explicitly specified templates. This is particularly true for the special template "Other Template", which broadens the LLMs' ability to learn repair behaviors beyond those predefined templates.

B. Model Inference

After the fine-tuning process is complete, we obtain a template selection model and a patch generation model. It is worth noting that during inference, traditional APR methods can generate hundreds of patches, constituting a vast patch space, by setting a large beam size in the beam search algorithm [6], [7], [12]. Theoretically, we could also adopt a large beam size for our generation model, relying solely on the predicted template from the selection model to maintain competitiveness. However, this strategy encounters two significant challenges. Firstly, the accuracy of the predicted template is not guaranteed, which could mislead the generation model into producing invalid patches. Secondly, the constraint of GPU memory poses a substantial hurdle, especially for large-scale LLMs such as CodeLlama-70B, potentially leading to out-of-memory (OOM) errors and thus a constrained patch space. These considerations drive NTR to adopt a dual-model approach, implementing template prioritization and iterative patch generation during the inference phase to effectively navigate these challenges.

1) Template Prioritization: To mitigate the risk of inaccurate template selection, we implement template prioritization, ordering the templates according to the model's output probabilities.

The inputs to the template selection model include the buggy code (x) and the candidate repair templates ($\mathcal{T}$), which are derived from the model's output. The model is adapted to compute the probabilities for each repair template as $\hat{p}(t_j|x) = \mathcal{M}_{\text{template}}(x, t_j)$. In this expression, $\hat{p}(t_j|x)$ denotes the likelihood of choosing repair template t_j based on the buggy code x. This phase ends up with a sorted candidate template space as $\mathcal{T}_{\text{sorted}}$. This space is derived by ordering the templates according to their predicted probabilities from the template selection model:

$$\mathcal{T}_{\text{sorted}} = \text{sort}\left(\hat{p}(t_j|x) : t_j \in \mathcal{T}, \text{desc}\right) \tag{5}$$

Here, $\text{sort}(\cdot, \text{desc})$ represents the sorting operation in descending order of the probabilities $\hat{p}(t_j|x)$, which are the chances of selecting each repair template t_j given the buggy code x. The output, $\mathcal{T}_{\text{sorted}}$, is subsequently fed into the patch generation model in the next step.

2) Iterative Patch Generation: Given the sorted template space $\mathcal{T}_{\text{sorted}}$, we iteratively concatenate each repair template t_{ij} from this space with the buggy code x_i to form model inputs. These concatenations are then fed into the patch generation model $\mathcal{M}_{\text{patch}}$, following the NMT paradigm, to predict the corresponding repair patches. This strategy of iteratively combining different repair templates t_{ij} with the bug code x_i can effectively explore the patch space. The underlying idea is that multiple solutions may exist for fixing a bug. Therefore, NTR can synthesize correct patches using different repair templates, leveraging the inherent advantage of template-based APR techniques [3], [23]. Moreover, the patch generation model's flexibility with the NMT workflow enables it to overcome template coverage limitations, potentially synthesizing the correct patch even when repair template prediction fails or corresponding repair templates are lacking, which combines the strengths of learning-based APR techniques [21].

For each of the k repair templates t_{ij} where $1 \leq j \leq k \leq |\mathcal{T}_{\text{sorted}}|$, our goal is to generate m patches, represented as:

$$(y_{ij}^1, y_{ij}^2, \ldots, y_{ij}^m) = \mathcal{M}_{\text{patch}}(x_i, t_{ij}) \tag{6}$$

Here, $y_{ij}^1, y_{ij}^2, \ldots, y_{ij}^m$ denote the m generated patches for the input pair (x_i, t_{ij}) using the patch generation model $\mathcal{M}_{\text{patch}}$.

By employing the subset of sorted repair templates k and adjusting the number of patches m generated per template, we can also maximize the patch space even with the constraint

of GPU memory. For example, we might be restricted to generating only 10 candidate patches at a time due to potential OOM. However, by associating 10 repair templates with a single bug code, we can iteratively generate 10 diverse groups of the candidate patches, expand the candidate patch space to include 100 potential patches.

In summary, this method not only mitigates the computational constraints but also leverages the model's capacity to explore a broader range of repair possibilities, substantially increasing the chances of generating effective patches.

III. Experiment Setup

A. Research Questions

- **RQ1: How well does NTR fix common bugs?** We will explore NTR's repair capabilities in Java bug repair scenarios and compare with baselines. (Repair Effectiveness)
- **RQ2: How much does NTR's design choice contribute to the overall repair capability?** NTR uses a two-stage repair strategy, and here we will explore the impact of different repair strategy settings on the results. (Ablation Study)
- **RQ3: How good is NTR at fixing security vulnerabilities?** To further evaluate NTR's generalization ability, we conducted experiments in vulnerability fixing scenarios. (Generalizability Study)

B. Dataset

1) Training Dataset:

- **Transfer dataset**. In the main experiment, we use the Transfer dataset [23] to implement model training and tuning. It contains about one million bug-fix pairs and corresponding fix templates that have been used to drive TRANSFER [23] and TENURE [21]. For example, the template-based APR work TENURE selected about 570K of these data for training. Considering the huge cost of training LLMs, we randomly selected 100K samples from them for NTR. The size of each set is as follows: Train/Val/Test = 97,416/2,029/2,030.
- **Recoder dataset**. In the generalizability study, we use the Recoder dataset [18] to implement model fine-tuning. We chose the Recoder dataset because Wu et al. [6] have successfully applied the Recoder dataset to vulnerability repair tasks with effective results. In addition, their work also indicates that some vulnerabilities still share similar repair patterns with general bugs. The size of Recoder dataset is as follows: Train/Val/Test = 129,310/7,178/7,178.
- **VulGen dataset**. In the generalizability study, we also use the latest VulGen dataset [42] to implement model training and tuning. We chose the VulGen dataset because it is the most comprehensive collection of C/C++ vulnerability repair datasets currently available. In particular, VulGen [42] has successfully extracted templates from the dataset for vulnerability generation. The size of VulGen dataset is as follows: Train/Val/Test = 9,392/522/522.

2) Testing Benchmark:

- **Defects4J V1.2**. Defects4J V1.2 [43] contains 395 Java bugs and is one of the most popular testing benchmarks [7]. Since the training set (Transfer dataset) contains multi-hunk fix examples, NTR can get the multi-hunk repair capability, and we use both single and multi-hunk samples from Defects4J V1.2 in our evaluation.
- **HumanEval-Java**. HumanEval-Java [12] contains 163 single-hunk Java bugs, and has no risk of data leakage. Here we use all samples in HumanEval-Java.
- **Vul4J**. Vul4J [44] contains 79 Java vulnerabilities and has been applied to LLM4APR [6]. Here, we followed the baseline work [6] by selecting 35 single-hunk samples.
- **CBRepair**. CBRepair [45] contains 55 C/C++ vulnerabilities and has been successfully applied to vulnerability repair tasks [45]. Here, we selected 37 single-hunk samples.

C. Baselines

- **APR Baselines**. We collected recent APR works to serve as baselines. This includes: ChatRepair [20], FitRepair [46], GAMMA [10], TENURE [21], Tare [29], Repatt [47], AlphaRepair [9], RAP-Gen [11], KNOD [26], Recoder-T [18], [29], TBar [3], Jiang et al. [12]. Following the common practice in the APR community [9]–[11], [18], [20], [29], [46], we reuse the reported results from previous studies [9]–[12], [20], [21], [26], [29], [46], [47] instead of directly running the APR tools.
- **LLM Baselines**. To present more clearly the improvement of NTR's strategy for LLM's repair capability, we implement NMT fine-tuning of LLMs to provide additional baselines, which we call **LLM×10** (StarCoder×10/CodeLlama×10). Since NTR employs multiple candidate templates to guide patch generation, it can obtain a larger patch space with a small beam size. In order to more fairly compare the performance of NTR against LLMs, we borrow from previous work [46] by sampling the model multiple times with 10 distinct sets of temperature parameters (i.e., LLM×10). This comparison is fair because NTR, utilizing 10 candidate templates, generates an equivalent patch space to that of LLM×10 through 10 rounds of sampling.

D. Implementation

1) Template Selection: As previously mentioned, CodeT5 was selected for NTR's template selection model due to its superior performance in classification tasks among million-level models [24]. Specifically, the template selection model underwent full parameter fine-tuning with CodeT5-220M [40]. Following the insights from previous work [7], we conducted training for 10 epochs and subsequently chose the checkpoint with the best perplexity. This approach addresses the limitation that a single epoch of fine-tuning is insufficient for million-level LLMs, preventing them from achieving convergence and adequately learning repair templates. Regarding the training hyper-parameters, we configured the learning rate to 5e-5 and set the maximum input/output sequence length to 512.

TABLE II: Different implementations of NTR.

Implementation	Model Components	
	Template Selection Model	Patch Generation Model
NTR$_{cs}$	CodeT5-220M	StarCoderBase-15B
NTR$_{cl}$	CodeT5-220M	CodeLlama-70B

2) Patch Generation: Considering the pivotal role of patch generation in NTR, which demands advanced natural and programming language understanding for effective patch synthesis, we selected two of the foremost LLMs for code families currently available, CodeLlama [48] and StarCoder [49], as identified on the Big Code Models Leaderboard [50]. These models offer the robust capabilities necessary for guiding the synthesis process. The patch generation model for NTR was parameter efficiently fine-tuned using StarCoderBase-15B [51] / CodeLlama-70B [22]; we implemented the fine-tuning using the LoRA [52] and Bit Quantization [53] (also called QLoRA [54]) techniques to reduce memory and computational costs, and trained only one epoch based on the findings from previous work [6], [12]. This is because for large-scale LLMs, one epoch can already make the model converge, and more epochs will destroy the model's generalization ability, thereby adversely affecting its ability to effectively perform repairs. In the training hyper-parameter settings, we set the learning rate to 5e-5 and the maximum input/output length to 2048; in the model inference phase, we set the beam size to 10 (or 100) to generate 10 (or 100) candidate patches for each template, and used Top-10 templates to expand the patch size 100 (or 1,000). The chosen size of the patch space is deemed suitable for APR tasks, as supported by previous studies [18], [26], [29], [46] that typically involve hundreds to thousands of candidate patches. Note that due to GPU memory constraints, the maximum beam size we can set for StarCoder and CodeLlama is 100 and 10, respectively.

3) Fault Localization and Patch Validation: In order to avoid additional biases introduced by the fault localization tool, our experiments were conducted under conditions of perfect fault localization. As noted by previous works [10], [20], [46], this is the preferred evaluation setting as it eliminates any differences in performance caused by running fault localization tools. Prior to patch validation, duplicate patches within the patch space are removed. The validation process for NTR aligns with established APR methodologies as documented in prior works [6], [9], [10], [12], [18], [20], [21], [23], [46]. Specifically, for each bug version, we allocate a maximum of 5 hours for the validation run, retaining only the first (top-1) plausible patch candidate that successfully passes all test cases. Should a candidate meet these criteria, the validation for that bug ceases immediately. Finally, the plausible patches will be manually checked to verify whether they are semantically correct or not.

We summarize the different implementations of NTR in Table II. The above is implemented based on Python 3.10 and PyTorch 2.2, and all models are from HuggingFace. We conduct the main experiments on a 12-Core workstation with Intel(R) Xeon(R) Bronze 3204 CPU, 256 GB RAM and a TESLA A100 GPU, running Ubuntu 20.04.6 LTS.

IV. Experiment Result

A. RQ1: Repair Effectiveness

1) Setup: In the main experiment, we aim to explore the effectiveness of NTR in common bug fixing scenarios. Here, we select the Transfer dataset for model fine-tuning and Defects4J V1.2 / HumanEval-Java for testing. Note that when testing the repair capabilities of the NTR implementation on Defects4J V1.2, we set the maximum beam size to 100 for the StarCoder-based NTR implementation (NTR$_{cs}$) and 10 for the CodeLlama-based NTR implementation (NTR$_{cl}$). This is reasonable because we use the beam size is smaller than or equal to most baseline works [10], [11], [18], [21], [26], [29]. In addition, when testing the repair capability of the NTR implementations on HumanEval, all NTR implementations set the beam size to 10, which is consistent with the compared baseline work [12]. The default setting for NTR is to use Top-10 candidate templates, generating at most 100 candidate patches when the beam size is set to 10, and at most 1000 candidate patches when the beam size is set to 100.

2) Result: Table III and Table IV present the repair results of the specific implementation of NTR on Defects4J V1.2 and HumanEval-Java. We will now proceed to analyze the repair results as outlined below.

Defects4J V1.2. As shown in Table III, the implementations of NTR outperform the current best baseline work, the ChatGPT-based APR tool ChatRepair [20]. Specifically, the StarCoder-based NTR$_{cs}$ fixes 14 more bugs (128 vs. 114) than ChatRepair, and the CodeLlama-based NTR$_{cl}$ fixed 25 more bugs (139 vs. 114) than ChatRepair. Importantly, the results show a clear enhancement of the NTR strategy on the LLMs' repair capability, when comparing the difference in the repair capability of the LLM-based NTR implementation over the LLM baselines. Specifically, the StarCoder-based NTR$_{cs}$ fixes 11 more bugs (128 vs. 117) than StarCoder×10, achieving a 9.40% improvement; the CodeLlama-based NTR$_{cl}$ fixes 13 more bugs (139 vs. 126) than CodeLlama×10, achieving a 10.32% improvement.

HumanEval-Java. As shown in Table IV, the implementations of NTR outperform all the baselines from the LLM4APR study [12]. Specifically, the StarCoder-based NTR$_{cs}$ fixes 59 more bugs (129 vs. 70) than InCoder-6B, the CodeLlama-based NTR$_{cl}$ fixed 66 more bugs (136 vs. 70) than InCoder-6B. However, due to the relatively small size of InCoder, comparing it directly to NTR does not yield meaningful insights. Therefore, our analysis primarily concentrates on evaluating the improvements offered by the LLM-based NTR implementation over the foundational model. Specifically, the StarCoder-based NTR$_{cs}$ fixes 11 more bugs (129 vs. 118) than StarCoder×10, achieving a 9.32% improvement; the CodeLlama-based NTR$_{cl}$ fixes 10 more bugs (136 vs. 126) than CodeLlamax10, achieving a 7.94% improvement.

Multi-Hunk Fixes. We draw on previous work [7] to mark out repair behaviors at multiple bug locations, which allows

TABLE III: Repair results for NTR and baselines on Defects4J V1.2 under the perfect fault localization.

APR Tool	**NTR$_{cl}$**	**CodeLlama×10**	**NTR$_{cs}$**	**StarCoder×10**	**ChatRepair**	**FitRepair**	**GAMMA**	**TENURE**	**Tare**	**Repatt**	**AlphaRepair**	**RAP-Gen**	**KNOD**	**Recoder-T**	**TBar**
Beam Size	10	10	100	100	/	/	250	500	100	/	25	100	1000	100	-
Patch Size	10×10	10×10	100×10	100×10	500	1000×4	/	500	100	200+1000	5000	100	1000	100	-
Chart	14/16	14/17	12/17	14/17	15/-	8/-	11/11	7/-	11/-	11/21	9/-	9/-	10/11	9/-	11/-
Closure	40/52	33/41	41/50	41/49	37/-	29/-	24/26	26/-	25/-	20/31	23/-	22/-	23/29	25/-	16/-
Lang	29/38	26/35	23/37	19/35	21/-	19/-	16/25	16/-	14/-	16/27	13/-	12/-	11/13	12/-	13/-
Math	39/51	39/50	38/53	31/48	32/-	24/-	25/31	22/-	22/-	22/51	21/-	26/-	20/25	20/-	22/-
Mockito	12/13	8/9	10/11	8/10	6/-	6/-	3/3	4/-	2/-	2/5	5/-	2/-	5/5	2/-	3/-
Time	5/7	6/8	4/7	4/8	3/-	3/-	3/5	4/-	3/-	4/7	3/-	1/-	2/2	3/-	3/-
#Total(Corr./Plaus.)	**139/177**	126/160	**128/175**	117/167	114/-	89/-	82/101	79/-	77/-	75/142	74/109	72/-	71/85	71/-	68/95

TABLE IV: Repair results for NTR and baselines on HumanEval-Java under the perfect fault localization.

APR Tool	**NTR$_{cl}$**	**CodeLlama×10**	**NTR$_{cs}$**	**StarCoder×10**	**InCoder-6B**	**InCoder-1B**	**CodeGen-6B**	**CodeGen-2B**	**CodeT5-large**	**CURE**	**RewardRepair**	**Recoder**	**KNOD**
Beam Size	10	10	10	10	10	10	10	10	10	10	10	10	10
Patch Size	10×10	10×10	10×10	10×10	10	10	10	10	10	10	10	10	10
#Total(Corr./Plaus.)	**136/150**	126/145	**129/146**	118/135	70/-	64/-	52/-	53/-	54/-	18/-	22/-	11/-	18/-

TABLE V: Multi-hunk bug repair on Defects4J V1.2.

Multi-Hunk Bugs						**Multi-Hunk Bugs**					
Bug ID	Edits	**Hercules**	**ITER**	**NTR$_{cs}$**	**NTR$_{cl}$**	Bug ID	Edits	**Hercules**	**ITER**	**NTR$_{cs}$**	**NTR$_{cl}$**
Chart-14	4	✓		✓	✓	Math-4	2	✓	✓	✓	✓
Chart-16	3	✓	✓	✓	✓	Math-22	2		✓	✓	✓
Closure-4	2	✓	✓		✓	Math-24	2	✓		✓	
Closure-78	2			✓	✓	Math-35	2	✓	✓	✓	✓
Closure-79	2		✓	✓	✓	Math-43	3	✓	✓		
Closure-106	2					Math-46	2	✓	✓		
Closure-109	2	✓				Math-62	3				
Closure-115	2		✓	✓	✓	Math-65	2				
Closure-131	2		✓			Math-71	2				
Lang-7	2					Math-77	2		✓	✓	✓
Lang-27	2					Math-79	2			✓	✓
Lang-34	2		✓	✓	✓	Math-88	3				
Lang-35	2				✓	Math-90	2				
Lang-41	4					Math-93	4				
Lang-47	2	✓	✓	✓	✓	Math-98	2	✓	✓	✓	✓
Lang-60	2	✓		✓	✓	Math-99	2				
Math-1	2		✓			**#Total(33 bugs)**		**12**	**15**	**15**	**16**

TABLE VI: Repair results of different fine-tuning strategies on Defects4J V1.2.

Variants	**Repair Strategy**	**Beam Size**	**Patch Size**	**Result**
StarCoder	❶ Basic fine-tuning	100	100	115/152
StarCoder-T	❷ One-stage template-based fine-tuning	100	100	106/146
StarCoder×10	❸ Basic fine-tuning with multiple-sampling	100	100×10	117/167
NTR$_{ns}$	❹ NTR without template selection	100	100×10	118/170
NTR$^{*}_{cs}$	❺ NTR without the special template	100	100×10	119/170
NTR$_{cs}$	❻ NTR	100	100×10	128/175

TABLE VII: Repair results of repair strategies ❶ and ❷ based LLM implementation on Defects4J V1.2.

LLMs	**StarCoder-15B**	**CodeLlama-13B**	**CodeGen25-7B**	**InCoder-6B**	**CodeGeeX2-6B**
Beam Size	10	10	10	10	10
Strategy ❶	101/122	95/112	88/111	74/93	75/93
Strategy ❷	87/109	82/98	77/102	72/97	62/74

NTR to learn and gain multi-hunk repair capabilities. As shown in Table V, we extract the repair results of ITER [55] and Hercules [56] that can repair multi-hunk bugs and compare them with the specific implementation of NTR. The results show that NTR successfully repairs 15 (or 16) out of 33 multi-hunk bugs, which rivals the repair results of the state-of-the-art multi-hunk repair tool ITER. Note that ITER and Hercules do not use perfect fault localization. Therefore, we wouldn't say that NTR is superior in repairing multi-hunk bugs. The motivation for conducting experiments on multi-hunk bugs is that existing template-based approaches (e.g., AlphaRepair and GAMMA) are often limited to single-hunk bugs due to the use of templates. Thus, we aim to demonstrate that our template-guided approach, NTR, has multi-hunk repair potential as well.

In summary, NTR achieves effective results on both Defects4J V1.2 and HumanEval-Java, which is better than many current LLMs as well as learning-based and traditional APR tools. And NTR also shows potential for multi-hunk fixes. In particular, NTR further enhances the repair capability of the LLM. NTR$_{cs}$ repairs 22 more bugs than the foundation model, achieving a 9.36% improvement; NTR$_{cl}$ repairs 23 more bugs than the foundation model, achieving a 9.13% improvement. These results indicate that NTR's strategy is successful, which is a feasible path to continuously optimize LLM4APR research.

B. RQ2: Ablation Study

1) Setup: In the ablation experiment, we aim to explore the impact of NTR's different design choices on the repair results. Therefore, we designed several variants to reveal the effects of different fine-tuning strategies. Specifically, we designed six fine-tuning strategies: ❶ **Basic fine-tuning strategy**: We used the NMT fine-tuning strategy from the recent LLM4APR studies [7], [12] to present the repair ability of directly fine-tuning LLMs. ❷ **One-stage template-based fine-tuning strategy**: The closest to our work is the recent TENURE [21], which uses a one-stage strategy to simultaneously predict templates and patches, while NTR employs a two-stage strategy to optimize template selection and patch generation tasks separately. Here we implement TENURE's strategy on top of LLMs to evaluate its effectiveness against ours. ❸ **Basic fine-tuning strategy with multiple-sampling**: As previously discussed, to achieve a fair comparison with NTR, we enhance the basic fine-tuning model's patch space by sampling it multiple times. ❹ **NTR strategy without template selection**: To clarify the template selection component's role in enhancing the repair results, we remove the template selection model from NTR and follow the practice of previous work [3], [23] to indiscriminately apply repair templates one by one. ❺ **NTR strategy without the special template**: To show the importance of setting the special template, we remove $OtherTemplate$ from NTR and compare its performance with the complete NTR strategy. ❻ **NTR strategy**: For reference, we include the original outcomes achieved by the two-stage NTR strategy.

Due to the computational costs associated with fine-tuning LLMs and the need for a diverse set of test samples, we select StarCoder as the foundational model and Defects4J V1.2 as the testing benchmark for our ablation study.

```
case Token.MOD:
    if (rval == 0) {
-       error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right);
        return null;
    }
    result = lval % rval;
    break;
case Token.DIV:
    if (rval == 0) {
-       error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right);
        return null;
    }
```

Fig. 3: Developer Patch for Closure-78.

TABLE VIII: Average correct template rank on Defects4J V1.2.

Project	Chart	Closure	Lang	Math	Mockito	Time	Average
NTR_{ns}	7.92	6.98	6.96	7.44	6.87	5.23	6.99
NTR_{cs}	1.49	2.19	2.25	2.42	3.03	1.57	2.26
Improvement	81.19%	68.62%	67.67%	67.47%	55.90%	69.98%	67.67%

2) Result: Table VI presents the repair results of LLM variants on Defects4J V1.2 using different fine-tuning strategies. Next we will analyze the impact of detailed design choices.

The limitations of one-stage template-based repair strategy. By comparing the repair results of repair strategies ❶ and ❷ (StarCoder vs. StarCoder-T), we can conclude whether TENURE's one-stage strategy is beneficial for basic NMT fine-tuning. To our surprise, the results in Table VI show that StarCoder-T is even worse than StarCoder, with 10 fewer bugs fixed. This result is confusing because TENURE performs well on the non-pre-trained model (RNN) of its original paper [21], yet it performs even worse on the LLM. To further confirm this finding, we select top-5 foundation model families on the Big Code Models Leaderboard [50] for additional experiments. According to Table VII, TENURE's one-stage strategy still performs worse than the basic fine-tuning. In analyzing the underlying causes, we identified a significant issue: the one-stage strategy tends to repeatedly predict the same templates during the simultaneous prediction of templates and patches. This redundancy limits the diversity of template types considered, increasing the likelihood of overlooking the correct template. To illustrate, we present an example where the bug is successfully fixed by StarCoder but not by StarCoder-T. As shown in Figure 3, the repair behavior of Closure-78 removes buggy statements, and thus its correct repair template should be $RemoveBuggyStatement$. However, we found that the StarCoder-T's predicted repair template is always $OtherTemplate$, which makes it miss the chance to synthesize the correct patch. Overall, TENURE's strategy proves suboptimal for predicting repair templates, mainly due to the joint cross-entropy that tends to favor the longer fixed code over the template, as detailed in Section II-A2. This demonstrates that using an independent template selection (template ranking) step is a more reasonable strategy.

The benefits of two-stage template-guided strategy. By comparing the repair results of repair strategies ❸ ❹ ❻ (StarCoder×10 vs. NTR_{cs} and NTR_{ns} vs. NTR_{cs}), we can assess the contribution of NTR's two-stage strategy in expanding the patch space as well as improving the repair effectiveness. **1) NTR_{cs} vs. StarCoder×10:** Limited by GPU memory, we set the maximum beam size to 100, which means at most 100 patches are generated at a time. Indeed, how to extend the patch space for enhancing the repair capability of LLMs is a challenge that needs to be solved in the era of LLMs [7]. NTR is designed to extend the patch space by using multiple repair templates, StarCoder×10 extends the patch space by sampling multiple times. As shown in Table VI, NTR_{cs} fixes 11 more bugs than StarCoder×10, which indicates that NTR's patch space extension strategy is more effective. NTR motivates the LLM to explore towards different repair behaviors by combining different repair templates, which is more effective than blindly exploring the patch space by multiple sampling. **2) NTR_{cs} vs. NTR_{ns}:** Some template-based APR works [3], [10] simply use all templates in a fixed order, ignoring the importance of template prioritization. Conversely, NTR performs template ranking by training a template selection model. As shown in Table VI, NTR_{cs} fixes 10 more bugs than NTR_{ns}. Besides, NTR_{cs} suffers from a milder degree of patch overfitting than NTR_{ns}. Specifically, we calculated the degree of overfitting using the number of overfitting patches divided by the number of plausible patches, NTC_{cs} vs. NTR_{ns} = 47/175 vs. 52/170 = 26.86% vs. 30.59%. We hypothesize that this improvement results from the template selection model effectively elevating the rank of the correct repair template within the candidate template space. This, in turn, indirectly boosts the rank of the correct patch within the candidate patch space. As shown in Table VIII, NTR_{cs} with the template selection model achieves a 67.67% reduction in the average ranking of correct templates. In addition, Table IX shows that NTR_{cs} also gets a 56.45% reduction in the average ranking of correct patches compared to NTR_{ns}. Overall, these results lend support to our hypothesis, indicating that the template selection component within NTR plays a crucial role in reducing patch overfitting and improving the repair effectiveness.

TABLE IX: Average correct patch rank on Defects4J V1.2.

Project	Chart	Closure	Lang	Math	Mockito	Time	Average
NTR_{ns}	15.79	64.68	38.43	23.04	75.45	11.25	43.12
NTR_{cs}	1.96	13.17	15.41	30.87	26.00	17.00	18.78
Improvement	87.59%	79.64%	59.90%	-33.98%	65.54%	-51.51%	56.45%

```
public int parseArguments(Parameters params) throws CmdLineException {
+       String param = null;
+       try {
-               String param = params.getParameter(0);
+               param = params.getParameter(0);
+       } catch (CmdLineException e) {}
        if (param == null) {
                setter.addValue(true);
```

Fig. 4: Developer patch for Closure-83.

```
public int parseArguments(Parameters params) throws CmdLineException {
+       String param = null;
+       try {
-               String param = params.getParameter(0);
+               param = params.getParameter(0);
+       } catch (Exception e) {}
        if (param == null) {
                setter.addValue(true);
```

Fig. 5: NTR_{cs} patch for Closure-83.

TABLE X: Repair results for NTR on Vul4J and CBRepair.

APR Tool	NTR$_{cs}$	StarCoder×10	VulMaster	Codex-12B	VulRepair
Beam Size	10	10	10	10	100
Vul4J	**11/14**	7/12	9/-	6.2/-	4/10
CBRepair	**17/28**	15/28	-	-	11/24

The importance of the special repair template. By comparing the repair results of repair strategies ❺ and ❻ (NTR$^*_{cs}$ vs. NTR$_{cs}$), we can clarify the importance of setting the special template $OtherTemplate$. As shown in Table VI, NTR$^*_{cs}$ without the special templates fixes 9 fewer bugs than NTR$_{cs}$ with the special templates, which suggests that the setting of the special templates is important to the NTR strategy. Recall that the introduction of the $OtherTemplate$ in NTR aims to address the template coverage issue, facilitating the learning and generation of more complex repair behaviors. We present an example where the NTR strategy, aided by the special template, successfully fixed a bug. As shown in Figure 4, the correct repair behavior of Closure-83 involves complex changes in replacement and insertion. Consequently, approaches with template coverage issues like GAMMA [10] and AlphaRepair [9], those focusing on single-line repairs like TENURE [21], and NTR variants without the special template (NTR$^*_{cs}$), all failed to fix it. In contrast, Figure 5 shows that, NTR$_{cs}$ effectively utilizes the special template to synthesize the correct patches. This evidence further supports that NTR$_{cs}$ mitigates the template coverage problem through the special template, enabling the synthesis of patches for complex repair behaviors.

C. RQ3: Generalizability Study

1) Setup: The strategy employed by NTR is theoretically general. To validate its generalizability, we conduct experiments on vulnerability repair. Specifically, we use StarCoder as the foundation model, Recoder and VulGen datasets as the training dataset, and Vul4J and CBRepair as the test benchmarks. Since SC4FT is Java-specific, we manually labeled templates for C/C++ samples in VulGen to accommodate language differences. We extracted the best repair results from the recent vulnerability repair work [6], [57] and re-implemented VulRepair [8] in our dataset for comparison. We set the same beam size of 10 as in recent work [6], and considering that VulRepair uses a small-scale model, we set the beam size to 100 for it. The purpose of this experiment is to explore whether NTR can work effectively in vulnerability repair scenarios.

2) Result: Table X presents the repair results of NTR on Vul4J and CBRepair. In total, NTR repaired 6 more vulnerabilities than the foundation model StarCoder×10, achieving a 27.27% improvement. This result again demonstrates the effectiveness of the NTR strategy. In addition, NTR fixed 2 more samples than the SOTA vulnerability repair work VulMaster [57] on Vul4J, achieving a 22.22% improvement. This shows that the LLM-based implementation of NTR can generalize well to other similar tasks.

V. Threats to Validity

1) Data Leakage: The pre-training data of LLMs may contain correct patches for buggy projects, which could affect the evaluation. We mitigate this threat from three ways: **First**, the LLM we used, StarCoder, provides a page [58] that can detect data leakage, which helps us detect and understand how our experimental results suffer from data leakage. Specifically, we detected the correct patches generated by NTR$_{cs}$ on Defects4J V1.2 in our main experiment and found that 8 (Closure-15/38/63, Lang-10/24/33/43, Math-104) of them were threatened by data leakage. Even removing these 8 bugs, NTR$_{cs}$ still repairs 120 (128-8) bugs, which is still ahead of the best baseline ChatRepair (120 vs. 114). **Second**, our main experiment was also conducted on HumanEval-Java, a synthetic benchmark designed to prevent data leakage. **Third**, since the NTR strategy is designed to further improve the repair capability of LLMs, we have compared NTR with the foundation model. Regardless of potential data leakage in the LLM, one strategy can be deemed successful as long as it demonstrates enhanced repair results. As shown in Table III, Table IV, and Table X, the results from NTR consistently outperform the foundational model, and thus the threats of data leakage is minimized.

2) Repair Efficiency: The substantial size of LLMs implies more time and memory costs for training and inference. We recognize this as a potential impact on repair efficiency, which we intend to analyze through two ways: **1) Time Cost**. During the model fine-tuning phase, the StarCoder-based NTR implementation (NTR$_{cs}$) took 35 hours and the CodeLlama-based NTR implementation (NTR$_{cl}$) took 85 hours. Fortunately, model fine-tuning is a one-time cost and the fine-tuned model can be used directly in the repair workflow. In the model inference phase, it took an average of 0.61s for NTR$_{cs}$ to generate a patch, and 3.76s for NTR$_{cl}$. It is undeniable that using a larger scale CodeLlama costs more time while bringing better repair capabilities. In the future, we expect more model acceleration techniques to mitigate this problem. **2) Memory Cost**. We use QLoRA [54] to save memory without sacrificing performance. Specifically, we use 8 Bit Quantization and LoRA for StarCoder so that only about 20G of memory is needed to load the model, we use 4 Bit Quantization and LoRA for the larger CodeLlama, which allows us to load the model with 40G of memory. Such memory costs allow researchers to deploy these LLMs on consumer-grade GPUs (e.g., two RTX 3090 24G). Also, NTR iteratively guides patch synthesis, which expands the patch space under memory constraints. Overall, we mitigate the memory cost threat using the QLoRA technique and NTR's design choices.

VI. Related Work

APR research has entered the era of LLMs. Researchers are devoted to designing novel repair strategies to maximize the repair potential of LLM. Based on the methodologies of patch generation, we categorize LLM-based APR approaches into *mask prediction*, *sequence transformation*, and other methods.

Mask prediction-based (i.e., *infilling*) works typically draw on specifically designed repair templates to guide the LLM to fill in masked locations by constructing masking prompts based on these templates. This approach uses pre-training tasks (e.g., MLM [59]) for the LLMs to generate patches directly. The earliest work was AlphaRepair [9], which introduced the zero-shot learning paradigm to the APR task, yielding surprisingly effective results. Subsequent works such as FitRepair [46] and Repilot [60] also use this paradigm. Similarly, GAMMA [10] leverages templates derived from prior template-based studies and applies the zero-shot learning paradigm. Another similar work is TypeFix [61], which automatically mines templates from the repair history and focuses only on Python type errors. However, most advanced LLMs, such as CodeLlama-70B, are not designed for cloze tasks, making it difficult to directly leverage them for mask prediction.

Sequence transformation-based (i.e, *NMT*) works typically tag the fault location and simultaneously feed the buggy code along with its context into the model. This enables the model to generate the appropriate transformations (patches). For example, VulRepair [8], InferFix [62], and recent empirical studies [6], [7], [12] have adopted this approach. The advantage of this approach is that the model is able to generate patches based on the defective code and its context, and has a broad applicability. However, merely fine-tuning LLMs might not represent the most effective strategy for automated program repair. Compared to the above work, our approach effectively incorporates prior domain knowledge (i.e., templates) into the fine-tuning process, thereby enhancing its overall effectiveness.

Additionally, LLMs with language comprehension and generation capabilities can also be utilized to automatically generate fixes in an interactive way. For instance, ChatRepair [20] interleaves patch generation with instant feedback to perform APR in a conversational style. More recently, the focus has shifted toward developing agents that enable LLMs to emulate the human debugging process to address real-world bugs, exemplified by RepairAgent [63], FixAgent [64], and AutoCodeRover [65]. These approaches enhance automated repair systems by incorporating interactive tools and iterative refinement processes. In the future, building frameworks for multi-agent collaboration to tackle real-world programming challenges appears to be a promising direction.

VII. Conclusion

This paper presents NTR, a fully LLM-powered, template-guided repair framework that enriches the template-based APR method by fine-tuning LLMs for enhanced template selection and patch generation. This two-phrase strategy effectively empowers LLMs to synthesize accurate patches under the direction of repair templates, which further unlocks their generative capabilities. Furthermore, NTR's patch generation is not confined by template coverage limitations. Through experiments, NTR achieves SOTA results, demonstrating the effectiveness of our approach.

Acknowledgement

We thank the reviewers for their insightful comments and suggestions. This research is supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering, TSE*, vol. 38, no. 1, pp. 54–72, 2011.

[2] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *35th International Conference on Software Engineering, ICSE*, 2013, pp. 772–781.

[3] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *28th International Symposium on Software Testing and Analysis, ISSTA*, 2019, pp. 31–42.

[4] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology, TOSEM*, vol. 33, no. 2, pp. 1–69, 2023.

[5] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy, SP*, 2022, pp. 1–18.

[6] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *32nd International Symposium on Software Testing and Analysis, ISSTA*, 2023, p. 1282–1294.

[7] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *38th International Conference on Automated Software Engineering, ASE*, 2023, pp. 1162–1174.

[8] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2022, pp. 935–947.

[9] C. S. Xia and L. Zhang, "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*, 2022, pp. 959–971.

[10] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *38th International Conference on Automated Software Engineering, ASE*, 2023, pp. 535–547.

[11] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2023, pp. 146–158.

[12] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *45th International Conference on Software Engineering, ICSE*, 2023, pp. 1430–1442.

[13] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *arXiv preprint arXiv:2312.15698*, 2023.

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

[15] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering, TSE*, vol. 47, no. 9, pp. 1943–1959, 2019.

[16] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *29th International Symposium on Software Testing and Analysis, ISSTA*, 2020, pp. 101–114.

[17] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *42nd International Conference on Software Engineering, ICSE*, 2020, pp. 602–614.

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

[19] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *44th International Conference on Software Engineering, ICSE*, 2022, pp. 1506–1518.

[20] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *33rd International Symposium on Software Testing and Analysis, ISSTA*, 2024.

[21] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, "Template-based neural program repair," in *45th International Conference on Software Engineering, ICSE*, 2023, pp. 1456–1468.

[22] "Hugging face: codellama/codellama-70b-hf," https://huggingface.co/codellama/CodeLlama-70b-hf, 2024.

[23] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *44th International Conference on Software Engineering, ICSE*, 2022, pp. 1169–1180.

[24] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *45th International Conference on Software Engineering, ICSE*, 2023, pp. 2136–2148.

[25] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *43rd International Conference on Software Engineering, ICSE*, 2021, pp. 1161–1173.

[26] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," in *45th International Conference on Software Engineering, ICSE*, 2023.

[27] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "Seqtrans: automatic vulnerability fix via sequence to sequence learning," *IEEE Transactions on Software Engineering, TSE*, vol. 49, no. 2, pp. 564–585, 2022.

[28] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *44th International Conference on Software Engineering, ICSE*, 2022, pp. 511–523.

[29] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *45th International Conference on Software Engineering, ICSE*, 2023.

[30] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *35th International Conference on Software Engineering, ICSE*, 2013, pp. 802–811.

[31] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *26th International Conference on Software Analysis, Evolution and Reengineering, SANER*, 2019, pp. 1–12.

[32] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering, EMSE*, vol. 25, pp. 1980–2024, 2020.

[33] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering, EMSE*, vol. 20, pp. 176–205, 2015.

[34] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2016, pp. 298–312.

[35] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *32nd International Conference on Automated Software Engineering, ASE*, 2017, pp. 648–659.

[36] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *31st International Symposium on Software Testing and Analysis, ISSTA*, 2022, pp. 39–51.

[37] X. Meng, "Sc4ft: Syntax checker for fix templates," https://github.com/mxx1219/SC4FT, 2022.

[38] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2020, pp. 1433–1443.

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

[40] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *2021 Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2021, pp. 8696–8708.

[41] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022.

[42] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "Vulgen: Realistic vulnerability generation via pattern mining and deep learning," in *45th International Conference on Software Engineering, ICSE*, 2023, pp. 2527–2539.

[43] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA*, 2014, pp. 437–440.

[44] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, "Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques," in *19th International Conference on Mining Software Repositories, MSR*, 2022, pp. 464–468.

[45] E. Pinconschi, R. Abreu, and P. Adão, "A comparative study of automatic program repair techniques for security vulnerabilities," in *32nd International Symposium on Software Reliability Engineering, ISSRE*, 2021, pp. 196–207.

[46] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *38th International Conference on Automated Software Engineering, ASE*, 2023, pp. 522–534.

[47] J. Jiang, Z. Zhao, Z. Ye, B. Wang, H. Zhang, and J. Chen, "Enhancing redundancy-based automated program repair by fine-grained pattern mining," *arXiv preprint arXiv:2312.15955*, 2023.

[48] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[49] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[50] "Big code models leaderboard," https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard, 2024.

[51] "Hugging face: bigcode/starcoderbase," https://huggingface.co/bigcode/starcoderbase, 2024.

[52] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *10th International Conference on Learning Representations, ICLR*, 2022.

[53] "Hugging face: Quantization," https://huggingface.co/docs/transformers/main/main_classes/quantization, 2023.

[54] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems, NIPS*, vol. 36, 2024.

[55] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *46th International Conference on Software Engineering, ICSE*, 2024, pp. 1–13.

[56] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *41st International Conference on Software Engineering, ICSE*, 2019, pp. 13–24.

[57] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *46th International Conference on Software Engineering, ICSE*, 2024, pp. 1–13.

[58] "Data portraits," https://stack.dataportraits.org/, 2024.

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

[60] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2023, pp. 172–184.

[61] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type

errors," in *46th International Conference on Software Engineering, ICSE*, 2024, pp. 1–13.

[62] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2023, pp. 1646–1656.

[63] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," *arXiv preprint arXiv:2403.17134*, 2024.

[64] C. Lee, C. S. Xia, J.-t. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via llm-based multi-agent synergy," *arXiv preprint arXiv:2404.17153*, 2024.

[65] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *33rd International Symposium on Software Testing and Analysis, ISSTA*, 2024.