Leveraging Large Language Models and Historical Patterns for Efficient Automated Program Repair

Babafunmise Adebowale
University of Saskatchewan, Canada
nec314@usask.ca

Pallavi
University of Saskatchewan, Canada
pan585@usask.ca

Wahhaj Javed
University of Saskatchewan, Canada
muj975@usask.ca

ABSTRACT

Automated Program Repair (APR) aims to reduce software maintenance costs by automatically generating patches for buggy code. Traditional APR methods rely on predefined fix patterns or limited bug-fix pairs, which often limits their ability to generalize to unseen bugs. With the rise of Large Language Models (LLMs) such as DeepSeek, it becomes possible to leverage extensive pretraining on large code corpora to learn more flexible and context-aware repair strategies. Although the base DeepSeek model performs reasonably well without fine-tuning, its effectiveness is constrained when applied to diverse and real-world bug scenarios, where bugs can be more complex and context-specific. This research investigates the efficacy of a fine-tuned DeepSeek-based APR system for single-statement bug fixes in Python. We fine-tuned DeepSeek on 10,000 samples from the RunBugRun dataset and 18,700 samples from CTSSB-1M dataset. We evaluated the model's in-domain performance on 1,000 held-out samples from RunBugRun and 2200 held-out samples from CTSSB-1M. To evaluate the out-of-domain performance of the model, the QuixBugs benchmark was used. The evaluation was carried out using both structural and semantic metrics, such as Exact Match, BLEU, CodeBLEU, and Levenshtein similarity, as well as functional correctness measured by unit test execution. Improved performance was observed on the in-domain test sets, while a slight decline was noted on the out-of-domain QuixBugs benchmark for both datasets. This highlights a trade-off between in-domain specialization and cross-domain generalization. By combining the representational power of LLMs with supervised fine-tuning on large-scale bug-fix data, our study contributes to more scalable and adaptable approaches to APR. Code and evaluation tools are released to facilitate reproducibility and further research.

CCS CONCEPTS

• **Software and its engineering** → *Software repository mining*; *Software maintenance and evolution*; *Practitioners' perspective*; *Software maintenance tools*; • **Computing methodologies** → *Machine learning*.

CMPT 470, April, 2025, Saskatoon, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

Automated Program Repair (APR), LLMs, Fine-Tuned LLM, Software Bugs, Bug Fix Pattern

ACM Reference Format:
Babafunmise Adebowale, Pallavi, and Wahhaj Javed. 2025. Leveraging Large Language Models and Historical Patterns for Efficient Automated Program Repair. In *Advanced Software Engineering (Winter 2025), April 2025, Saskatoon, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Background: Software bugs are pervasive, costly, and often time consuming to fix. More than 50% of the total cost of software is consumed in identifying and fixing defects and losses due to software failures in a production environment [27]. Automated Program Repair (APR) has emerged as a crucial area in software engineering to alleviate this burden by generating patches with minimal human intervention[30]. Since the early days of GenProg, APR techniques have evolved considerably [17], ranging from search-based and constraint-based approaches to templates-driven and machine learning–based models.

With the advent of Large Language Models (LLMs), APR has entered a new paradigm. LLMs trained on massive code corpora can model the semantics of buggy and fixed code in powerful ways, making them suitable for tasks like patch generation, bug localization, and code completion. This has led to the rise of LLM-based APR techniques that leverage zero-shot, few-shot, and fine-tuning paradigms for automatic repair [41]. However, while zero-shot prompting has seen extensive exploration, the fine-tuning potential of open-source LLMs like DeepSeek remains underexplored.

Problem Statement: Although APR has advanced significantly, key challenges remain that limit its effectiveness in real world software maintenance. These challenges include:

- Limited Generalization to Novel Bugs: Traditional APR methods rely on predefined templates to fix known bug types, while learning-based APR models rely on curated bug-fix datasets to predict repairs. In both cases, the reliance on historical patterns leads to limited patch diversity, making it difficult to generate fixes for complex or unseen bugs. [33] This limitation hampers their applicability in dynamic, real-world environments where new and diverse bug patterns frequently arise.
- Lack of Systematic Studies on Fine-Tuning LLMs for APR: Although LLMs have shown strong zero-shot capabilities [28], the potential of fine-tuning these models for specific bug repair tasks has been relatively underexplored. Most of the existing work focuses on prompting LLMs, leaving

the fine-tuning paradigm underexplored. This gap makes it difficult to assess the trade-offs between specialization (in-domain performance) and generalization (cross-domain robustness) in LLM-based APR, thus limiting their practical deployment for broader bug-fixing scenarios. [7].
- Challenges in Identifying the Most Effective Bug Types for Fine-Tuned LLMs: Although fine-tuned LLMs have shown success in fixing bugs [36], it is not yet clear which bug categories they handle most effectively and which pose challenges [18].

Current State of the Art: Traditional APR methods are based on three main approaches:

- Search-based APR: treats program repair as an optimization problem. It generates numerous candidate patches (often via genetic programming or random mutations) and evaluates them against test suites. Classic examples include GenProg and RSRepair. These methods often suffer from inefficiency and overfitting—patches that pass tests without genuinely resolving the defect [16].
  Recent Innovations: Modern tools like ARJA-e integrate search with template-based strategies, improving accuracy and reducing search time [40]. LSRepair extends traditional search-based methods by incorporating code retrieval from external repositories. It performs live code search to find syntactically and semantically similar code snippets from large codebases (e.g., GitHub) and reuses those snippets to synthesize repair candidates [22]. Prophet and PROPR improve patch prioritization by learning statistical models over past human-written patches. These models capture common successful editing patterns and contexts, termed learned patch priors, to rank candidate patches according to their likelihood of being correct before executing tests [23].Search-based APR can produce valid but semantically incorrect patches. The generate-and-validate approach is computationally expensive. Models like HDRepair [15] help alleviate this by biasing towards known good edits to reduce search space.
- Semantic-Based APR: Also known as constraint-based APR, leverages program semantics to generate logically sound patches. It works by translating the buggy program and its associated test cases into formal logical constraints. These constraints are then solved using SMT solvers (Satisfiability Modulo Theories), often in conjunction with symbolic execution, to systematically explore program paths and identify patches that satisfy all correctness conditions. Tools such as Angelix [25] and SemFix [26] formally encode correctness conditions and synthesize repairs that satisfy them.
  Recent Innovations: FAngelix uses Monte Carlo Tree Search to accelerate synthesis while preserving correctness [38]. JFix brings constraint solving to Java using Symbolic PathFinder [13]. S3 combines syntactic pruning with semantic validation to handle large programs [14]. Semantic APR is accurate but does not scale well to complex or multi-function code. The problem of solving constraints is computationally expensive.
- Template-Based APR relies on predefined fixed templates or patterns to detect and correct bugs by matching them to known solutions. These systems maintain a curated library of templates—such as "replace == with !="—that are applied when a bug is identified. The repair process involves attempting each template on the buggy code and validating the resulting patch using a test suite. Systems such as PAR [11] and TBar [21] use manually or automatically extracted templates to replace buggy code segments.
  Recent Innovations: TBar integrates diverse patterns into a unified system. FixMiner [12] and AVATAR [20] mine real-world patches to expand the template base. GAMMA improves templates using LLM to fill in missing patch components, improving coverage and generalization [42]). Template-Based APR still fails on novel bugs that do not match any template

As APR research progressed, the limitations of traditional methods, such as rigidity, poor generalization, and reliance on handcrafted rules, led to the emergence of data-driven approaches such as:

- Learning-Based APR: utilizes machine learning models to automatically learn bug-fixing patterns from data. They treat bug fixing as a supervised translation task, mapping the buggy code to its fixed version. Models like SequenceR [4] and CoCoNuT [24] learn from large datasets of bug-fix pairs.
  Recent Innovations: CoCoNuT uses multiple encoders to model code and context. CURE [8] pretrains on general code before fine-tuning on repairs to enhance generalization. These models often struggle with rare or complex bugs outside their training distribution. Generalization is limited, especially without large and balanced datasets.
- LLM-Based APR: leverages pre-trained large language models (e.g., ChatGPT, CodeT5, DeepSeek) for zero-shot or fine-tuned bug repair. These models have seen massive code corpora, enabling repair even without explicit training on bug-fix pairs.
  Recent Innovations: TFix fine-tunes T5 on ESLint errors [1]. ThinkRepair [39] and ChatRepair [35] use multi-turn interaction or chain-of-thought prompting to improve patch correctness. LLM-based APR offers significant benefits, including strong generalization to novel bug types, reduced dependence on curated datasets, and flexibility across programming languages and natural language input. However, these models are computationally expensive and may occasionally hallucinate, producing syntactically valid but semantically incorrect patches that fail to address the underlying bug.

Objectives: This research aims to:

- Fine-tune DeepSeek-Coder for single-statement bug repair: Adapt the DeepSeek-Coder-V2-Lite-Instruct [44] model to the specific task of repairing single-line Python bugs.
- Evaluate In-Domain and Out-of-Domain Generalization: Benchmark the fine-tuned model on a held-out set of in-domain samples (from RunBugRun [29] and CTSSB-1M [31]) and an out-of-domain benchmark (QuixBugs), using multiple structural and functional metrics (Exact Match, BLEU, CodeBLEU, Levenshtein similarity, and unit test pass rate).
- Compare Post-Finetuning Performance to Base Model: Quantitatively assess how fine-tuning impacts the performance

of DeepSeek relative to its base model, focusing on improvements in generalization, precision, and robustness.
- Characterize Strengths and Weaknesses of Fine-Tuned LLMs: Analyze which bug types are most effectively repaired by DeepSeek, identifying areas where the fine-tuned DeepSeek model succeeds or struggles (e.g., expression swaps, control logic).
- Compare DeepSeek with Traditional APR and Other LLM-Based Tools: Place the performance of the model within the broader APR landscape using publicly available QuixBugs results for a variety of traditional tools[37] (e.g., JGenProg, Arja, RSRepair, Cardumen, Nopol, JKali, and others), as well as LLMs [6] such as O1-preview, GPT-4o, and others.
- Support Reproducibility: Release code, evaluation metrics, and results to facilitate transparency and further research on LLM-based APR.

Method Summary: To evaluate the performance of LLM after fine-tuning for automated program repair (APR), we adapted the DeepSeek-Coder-V2-Lite-Instruct model to repair single-statement bugs in Python programs. Our approach involved training the model using a method-level context, which provided a balanced view of the surrounding code while maintaining computational efficiency. We conducted two separate training runs:

- The first model was fine-tuned on 10,000 bug-fix pairs from the RunBugRun dataset and evaluated on a 1,000-sample held-out test set from the same dataset for in-domain performance.
- The second model was fine-tuned on 18,700 bug-fix pairs from the CTSSB dataset and evaluated on a 2,200-sample held-out test set from CTSSB for its in-domain performance.

To assess generalization capabilities, both models were further evaluated on an out-of-domain test set, using 40 samples from the QuixBugs benchmark, which consists of algorithmic bugs structurally distinct from both training datasets. Performance evaluation was carried out using both structural metrics—Exact Match, BLEU, CodeBLEU, Levenshtein similarity— and functional correctness, measured through the execution of unit tests. We also compared the fine-tuned models with:

- Its base (no fine-tuning) DeepSeek-Coder version.
- A range of traditional APR tools (e.g. JGenProg, NPEFix, Nopol, Cardumen).
- Other LLM-based tools (e.g. GPT-4o, CIRCLE, Codex, O1-preview), using published results on QuixBugs.

Research Questions:

RQ1: How effectively can fine-tuned LLMs generalize learned bug-fix patterns to suggest repairs for unseen bugs? It is important to assess whether LLM can overcome the generalization limitations of template-based and dataset-constrained learning-based models, as it addresses the core challenge of adapting to real-world scenarios where bug patterns are often new and unpredictable.

RQ2: What bugs are most effectively fixed by fine-tuned LLMs, and which remain challenging? Understanding the types of bugs that LLMs can or cannot fix helps identify the strengths and weaknesses of the approach. This insight is crucial to refine the model, select appropriate training data, and guide future improvements in automated program repair systems.

2 SIGNIFICANCE OF THE TOPIC

Importance: Automated Program Repair is a critical research area in software engineering, aiming to reduce the time and cost associated with debugging and maintenance. As software systems grow in complexity, manual debugging becomes increasingly inefficient. By improving APR with Large Language Models, this study contributes to the advancement of automated debugging techniques that can be applied across various software development domains.

Impact: This research has both theoretical and practical implications. Theoretically, it advances the study of LLMs in software repair, bridging the gap between template-based APR and learning-based APR. Practically, it provides an approach that could improve the effectiveness of real-world debugging tools, potentially reducing software downtime and increasing developer productivity.

Motivation: The motivation behind this study arises from the limitations of existing APR methods, which rely on rigid fix templates, computationally expensive constraint-solving, or learning-based methods constrained by dataset biases. With the recent success of LLMs in code generation tasks, there is an opportunity to explore their effectiveness in automated bug fixing.

Applications: The outcomes of this study could be applied in various areas, including:

- Software Development: Integrating improved APR into IDEs and CI/CD pipelines for real-time bug fixes.
- Security Patching: Automatic detection and repair of vulnerabilities in open source and enterprise software.
- Education: Assist students in learning debugging by providing AI-generated fix suggestions.

The research benefits the following:

- Developers – Faster debugging and patching tools to improve productivity.
- Software companies – Reduced maintenance costs and faster response times for bug fixes.
- Security Analysts – Automated vulnerability detection and patching.
- Researchers – Further exploration into LLM-based APR and its effectiveness in various programming languages.

3 METHODOLOGY

Our methodology aims to investigate the effectiveness of fine-tuned LLMs for automated repair. This section describes the construction of the dataset, training pipeline, and the evaluation setup, to ensure reproducibility and transparency.

3.1 Dataset Construction and Characteristics

We used two main datasets for fine-tuning and in-domain evaluation.

RunBugRun Dataset : A multilingual dataset containing 145,370 buggy-fixed code pairs along with associated test cases. [29]

- Filtering: Extracted Python-specific entries from the original multilingual RunBugRun dataset.
- Cleaning & Normalization: Parsed and validated JSONL entries, removed corrupt or noisy samples, and ensured uniform structure.
- Public Release: The cleaned dataset was uploaded to Hugging Face to support reproducibility.
- Preparation for Fine-Tuning:
  - Selected 10,000 samples for training and 1,000 samples for testing.
  - Reformatted dataset with prompt-based instructions (e.g., "Fix this Python code bug:").
  - Saved in JSONL format compatible with DeepSeek training scripts.

CTSSB-1M Dataset: A large-scale dataset of more than one million single-statement bug-fix instances curated from open-source Python repositories. [31].

- Contains isolated single-statement bugs annotated with SSTUB patterns.
- Refinement Criteria:
  - Entries labeled as likely bug fixes.
  - Entries in which changes were made in a function or a method.
  - Selected equal number of entries for each of the 22 SSTUB patterns.
- Dataset Breakdown:
  - 18,700 training samples (850 per SSTUB pattern).
  - 2,200 test samples (100 per pattern).
  - Entries with excessive token length were truncated to meet the model input limits.

3.2 Model Fine-Tuning Strategy

Model: For the model, we used the DeepSeek-Coder-V2-Lite-Instruct model (15.7B parameters) as the model is pre-trained for code-specific tasks [44]. We fine-tuned the model using PyTorch and applied QLoRA with 4-bit quantization and bfloat16 to make fine-tuning feasible on available compute. We also employed the Hugging Face Transformers library for efficient fine-tuning and model adaptation. For training, Instructional fine-tuning using next-token generation was utilized to fix single-statement bugs.

```
You are an AI programming assistant, utilizing the DeepSeek Coder
model, developed by DeepSeek Company, and you only answer questions
related to computer science. For politically sensitive questions,
security and privacy issues, and other non-computer science questions,
you will refuse to answer.

### Instruction:
Fix the buggy code below:

<BuggyCode>

Provide only the corrected code

### Response:
```

Figure 1: Instruction prompt used during training to guide the model in repairing single-statement bugs.

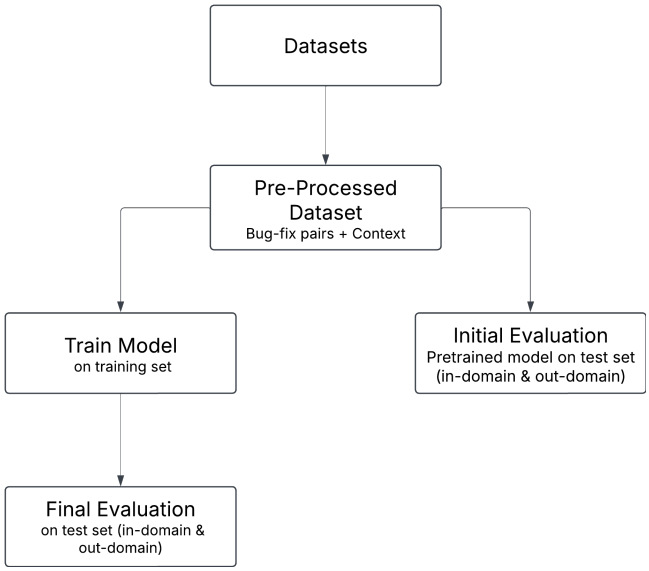

Figure 2: Study Methodology

Finetuning Pipeline: The finetuning pipeline with the following hyperparameters for each dataset is as follows:

- RunBugRun Model:
  - Epochs: 3
  - Batch Size: 16
  - Learning Rate: 2e-5
  - Max Sequence Length: 1024
  - Optimizer: Cosine LR scheduler
  - Tools: Hugging Face Transformers, PyTorch, DeepSpeed
  - Gradient Accumulation: 4
  - Checkpoints: Saved every 100 steps
- CTSSB Model:
  - QLoRA with 4-bit quantization
  - Epochs: 3
  - Max Sequence Length: 2048
  - Batch Size: 2
  - Optimizer: Cosine LR scheduler
  - Tools: Hugging Face Transformers, PyTorch, DeepSpeed
  - Trained using the official DeepSeek fine-tuning script with the default prompt [5].

Context Level: Method-level context was included with each bug-fix pair during both training and evaluation to improve the model's understanding of the code structure and semantics. For this research, the method-level context is the most suitable approach, as it maintains a balance between contextual awareness and computational efficiency. It allows the model to analyze function definitions, parameters, and return types, which are crucial for generating meaningful fixes and to capture the intent and dependencies of the function while maintaining a manageable scope for learning and inference. The method-level context provides a more comprehensive view of input-output relationships, control flow, and variable dependencies, leading to more informed repairs.

3.3 Evaluation Metrics and Benchmarks

We evaluated the two separate DeepSeek models: one on RunBugRun (10,000 training / 1,000 test samples) and another on CTSSB (18,700 training / 2,200 test samples).

Evaluation Metrics Used:

- Exact Match – Byte-for-byte similarity with ground-truth fix.
- AST Match - Structural similarity between the generated fix and the correct solution.
- BLEU Score – N-gram level overlap between generated and correct fix.
- CodeBLEU – Captures both syntax and dataflow-aware similarity.
- Levenshtein Similarity – Measures character-level edit distance.
- Levenshtein Similarity Ratio - Normalized version of the Levenshtein distance.
- Unit Test Pass Rate – Assesses whether repaired code passes associated test cases.

Then both models were evaluated on the QuixBugs benchmark to assess cross-domain generalization and real-world bugs.

QuixBugs Benchmark: A dataset containing 40 algorithmic problems implemented in multiple languages. [19]

- Used solely for out-of-domain evaluation.
- Contains 40 Python implementations of algorithmic problems, structurally distinct from RunBugRun and CTSSB.
- Provides a robust benchmark for testing model generalization.

This helped us to understand how well the fine-tuned models perform on unfamiliar bug patterns beyond their respective training distributions.

3.4 Baseline Comparisons

To contextualize the performance of our fine-tuned DeepSeek models, we compared them against traditional and modern baselines in Automated Program Repair (APR). This comparative analysis helps to highlight the strengths and limitations of LLM-based APR approaches relative to existing methods.

We reference publicly reported results from several classical APR tools on the QuixBugs benchmark:

- GenProg: A search-based approach using genetic programming.
- RSRepair: An APR system that uses redundancy-based patch generation.
- Nopol: A constraint-based tool focusing on fixing conditional statements.
- Cardumen: A template-based APR technique that synthesizes code fragments for repairs.

These tools often rely on predefined fix templates or program mutation techniques and have demonstrated varying degrees of success, particularly in fixing syntactic bugs.

We also compare our results with those of other large language models evaluated on the same benchmark such as ChatGPT and GPT-4 - closed-source LLMs evaluated using prompt-based (zero-shot/few-shot) learning - and CIRCLE and O1-preview - open and semi-open models designed for code completion and repair.

4 STUDY FINDINGS

Overview of Findings: Our study demonstrates that fine-tuning the DeepSeek-Coder model on curated bug-fix datasets can significantly improve Automated Program Repair (APR) performance. Compared to the base model, the fine-tuned version achieved higher scores in exact matches, semantic similarity (BLEU/CodeBLEU), and character-level similarity (Levenshtein). These improvements were consistently observed across both the RunBugRun and CTSSB test sets, validating the effectiveness of fine-tuning on domain-specific bug-fix data. The results supported our hypothesis that LLMs trained in comprehensive real-world datasets are capable of producing syntactically valid and context-aware fixes. However, evaluation on the QuixBugs benchmark revealed the challenges of out-of-domain generalization, where performance gains were present but less pronounced.

4.1 Findings on Fine-Tuned LLMs' Ability to Generalize Bug-Fix Patterns (RQ1)

To assess how well fine-tuned LLMs can generalize bug-fix patterns, we evaluated two versions of the DeepSeek-Coder-V2-Lite-Instruct model, each fine-tuned on a different dataset — RunBugRun and CTSSB — using both in-domain and out-of-domain test sets. We report results based on structural metrics (Exact Match, AST Match), semantic metrics (BLEU, CodeBLEU, Levenshtein), and functional correctness (Unit Test Pass Rate).

4.1.1 ***In-Domain Evaluation.*** For the RunBugRun fine-tuned model, in-domain evaluation was conducted on a held-out test set from the same dataset. As shown in Table 1, the fine-tuned model outperformed the base model across all metrics. These results support the hypothesis that fine-tuning enhances the model's ability to learn context-aware and syntactically valid repairs when the test data shares structural similarities with the training data.

Table 1: RunBugRun - Comparison of Metrics between Base Model and Fine-Tuned Model for In-Domain Evaluation

Metric	Base Model	Fine-Tuned Model
Exact Match (%)	11.5	17.1
AST Match (%)	17.4	20.9
BLEU Score (%)	58.43	64.25
CodeBLEU Score (%)	67.47	71.0
Levenshtein Distance	197.208	195.020
Levenshtein Similarity (%)	82.61	83.61

For the CTSSB fine-tuned model, a similar trend was observed on its in-domain test set (see Table 2). The fine-tuned model also outperformed the base model across all metrics. These results demonstrate that even on large-scale, diverse datasets, the fine-tuning process significantly improves structural and semantic accuracy.

4.1.2 ***Out-of-Domain Evaluation (QuixBugs Benchmark).*** To test generalization to structurally different bugs, we evaluated

Table 2: CTSSB - Comparison of Metrics between Base Model and Fine-Tuned Model for In-Domain Evaluation

Metric	Base Model	Fine-Tuned Model
Exact Match (%)	1	17
AST Match (%)	4	18
BLEU Score (%)	83	94
CodeBLEU Score (%)	78	91
Levenshtein Distance	371.43	200.45
Levenshtein Similarity (%)	88	97

both fine-tuned models on the QuixBugs benchmark, which was not part of the training data.

As shown in Table 3, the RunBugRun-trained model achieved a CodeBLEU score improvement but experienced performance drops in the other metrics, including Unit Test Pass Rate. This suggests that while the model learned meaningful syntax-aware patterns, its fixes were less likely to pass functional tests when faced with unfamiliar bug types.

Table 3: RunBugRun Trained Model on QuixBugs Out-of-Domain Evaluation

Metric	Base Model	Fine-Tuned Model
Exact Match (%)	27.5	22.5
AST Match (%)	37.5	30.0
BLEU Score (%)	72.2	67.37
CodeBLEU Score (%)	79.5	84.83
Levenshtein Similarity (%)	88.28	80.36
Unit Test Pass Rate (%)	72.5	60.0

In contrast, the CTSSB-trained model (Table 4) showed a more consistent improvement in semantic and structural metrics. There was improvement in all metrics except Unit Test Pass rate which was slightly lower. This indicates a better balance between generalization and structural awareness, likely due to the more diverse training patterns in CTSSB.

However, the CTSSB fine-tuning data did not include import statements. Manual inspection of the QuixBugs outputs showed that the model consistently omitted module-level imports when they were present in the prompt. Since these imports are required for the QuixBugs unit tests to run, we manually added them when they were missing in the output, resulting in the slightly improved evaluation metrics shown in Table 4. This behavior suggests that the model tends to overlook prompt elements outside the method body, such as import statements, probably because, during fine-tuning, it only encountered isolated function definitions beginning with def, without surrounding context.

Table 4: CTSSB Trained Model on QuixBugs for Out-of-Domain Evaluation

Metric	Base Model	Fine-Tuned Model	Fine-Tuned Model Manual Imports
Exact Match (%)	27.5	50	55
AST Match (%)	37.5	50	55
BLEU Score (%)	72.2	91	92.0
CodeBLEU Score (%)	79.5	92	92.8
Levenshtein Similarity (%)	88.28	98.5	98.8
Unit Test Pass Rate (%)	72.5	62.5	67.5

> **Summary of RQ_1**: In summary, both fine-tuned models generalized learned bug-fix patterns effectively to their respective in-domain test sets, with the CTSSB-trained model showing stronger improvements on the out-of-domain QuixBugs benchmark. This supports our hypothesis that fine-tuned LLMs can internalize transferable bug-fix knowledge, though some trade-offs in exactness and test pass rates exist when crossing domain boundaries.

4.2 Findings on Bug Types Effectively Fixed (RQ2)

The evaluation metrics used in Tables 1 and 2 were also calculated individually for each SSTUB pattern. The average of exact match, AST match, BLEU, CodeBLEU, and Levenshtein similarity for each pattern was taken to create overall scores for the different patterns, which could be used to gauge the fine-tuned model's performance for each pattern. The averages for the bottom three and top three patterns are given in Table 5.

Table 5: Bottom three and top three patterns for which the fine tuned model performed the worst and the best

Bug Type	Base	Fine-Tuned
Add Elements To Iterable	0.49	0.54
More Specific If	0.50	0.57
Less Specific If	0.50	0.57
Same Function Wrong Caller	0.53	0.71
Same Function Swap Args	0.50	0.72
Change Identifier Used	0.56	0.74

The fine-tuned model performs the best on the Change Identifier Used bug pattern. This pattern is also the most common bug pattern in the CTSSB dataset. As an equal number of entries for each pattern were used for training, the fine-tuned model's performance on this bug pattern is likely due to two causes:

(1) This pattern being the most common in the CTSSB dataset indicates that this type of bug was also common in the dataset used to train the base model. As shown in Table 5, the base model also performs the best for this pattern. So, the fine-tuned model's performance is due to the learned behaviour of the base model.
(2) This type of bug is easy to fix using the surrounding context. It is common to use the wrong identifier from the ones

defined in a function, and the correct one can often be determined just by looking at what the function is doing and which identifiers are defined in it.

Contrast this with the Add Elements To Iterable pattern in which an iterable used inside a method is missing an entry. An example from the testing dataset is `if control.type in ['text', 'textarea']` for which the correct code is `if control.type in ['text', 'textarea', 'password']`. Just by looking at the method in which this statement occurs, it is not possible to determine what the possible values for `control.type` could be as they are not defined in the method. Without this additional context, it is not possible for a human to fix this bug, much less an LLM.

These results show that fine-tuning allows the model to internalize common, reusable repair strategies and increase its performance. Moreover, fine-tuning is more beneficial for certain types of bugs than it is for others. As shown in Table 5, the base model's range is from 0.49 to 0.56 whereas the fine-tuned model's range is from 0.54 to 0.74. For the bug types that can be fixed using the surrounding context, the fine-tuned model's performance increases drastically from the base. However, performance gains are minimal for bugs that require a broader context to be fixed, which is limited in a single-statement repair setup.

> **Summary of RQ_2**: In summary, the bug-type analysis revealed that the fine-tuned DeepSeek-Coder-V2-Lite-Instruct model performs best on syntactic and localized bug categories, particularly those involving variable renaming and function argument swapping. These patterns are more prevalent in the training data and often require minimal contextual reasoning, allowing the model to internalize and generalize effective fix strategies. These findings highlight the strengths of fine-tuned LLMs in addressing structurally simple and high-frequency bugs, while also exposing their limitations in handling cases that require broader reasoning.

4.3 Findings on Comparison with Existing APR Tools

In this section, we compare the performance of the fine-tuned DeepSeek model with several other state-of-the-art Automated Program Repair (APR) tools, including both LLM-based and non-LLM-based systems, as evaluated on the QuixBugs benchmark. The comparison highlights the strengths and limitations of the fine-tuned DeepSeek model relative to other tools based on the number of repaired programs from Quixbugs benchmark set, as reported in prior comparative studies [6, 37], and is summarized in Table 6.

The DeepSeek-Coder-V2-Lite-Instruct model repaired 24 out of 40 programs, which places it in the middle of the performance spectrum, outperforming other models like CIRCLE, Codex, and GPT 3.5 Turbo, but falling short of models like O1-preview and O1-mini. This indicates that while the fine-tuned DeepSeek model performs competitively, it does not yet reach the level of some other leading APR tools in terms of repair success rate. Despite good performance in terms of structural and semantic matching, the unit test pass rate remained consistently lower at 60% for both the fine-tuned RunBugRun and CTSSB models. This suggests that

Table 6: Comparison of APR Tools Based on Number of Repaired Programs from QuixBugs Benchmark

Repair Tool	APR Type	# Repaired Programs
O1-preview	LLM	38/40
O1-mini	LLM	37/40
GPT-4o	LLM	35/40
Finetuned DeepSeek-Coder-V2-Lite-Instruct	LLM	24/40
CIRCLE	Learning	23/40
Codex	LLM	21/40
GPT 3.5 Turbo	LLM	19/40
Cardumen	Semantic	5/40
JGenProg	Search	4/40
Nopol	Semantic	4/40
NPEFix	Template	2/40

while the generated fixes are syntactically and semantically close to the correct solutions, they might not always be functionally correct, and additional refinement or validation may be needed. DeepSeek-Coder-V2-Lite-Instruct, despite its slightly lower repair rate compared to the best LLM models, still demonstrates strong potential in the APR field. Its performance is noteworthy given its fine-tuned nature and the challenges of applying a single model to a diverse set of real-world bugs. Further fine-tuning and optimization could close the gap between DeepSeek and the leading LLM-based models.

> **Summary of Comparison with existing APRs**: In summary, the fine-tuned DeepSeek-Coder-V2-Lite-Instruct model performed competitively when compared to existing Automated Program Repair (APR) tools, repairing 24 out of 40 programs from the QuixBugs benchmark. While it fell slightly short of top-performing LLMs like O1-preview (38/40) and GPT-4o (35/40), it outperformed several traditional tools and LLM baselines such as Codex and GPT-3.5 Turbo. This result is notable given DeepSeek's general-purpose architecture and single-model fine-tuning approach, highlighting its strong capacity for generalization. With further optimization, it has the potential to rival or complement more specialized APR systems.

5 DISCUSSION

Key Findings: Our study shows that fine-tuning Large Language Models (LLMs), specifically DeepSeek-Coder-V2-Lite-Instruct, substantially enhances performance in repairing single-statement bugs across diverse datasets. Fine-tuning improves the model's ability to generate syntactically valid and semantically appropriate patches that align with training patterns. On in-domain evaluations, the model learns reusable repair strategies for localized errors like identifier renaming, argument reordering, and single-token edits. The model also demonstrates meaningful generalization to out-of-domain bugs, particularly those structurally aligned with training data. However, performance declines for more complex or novel bugs not represented in the training set. Compared to traditional and learning-based APR tools, the fine-tuned DeepSeek model delivers competitive results using a single model configuration. This

approach bridges the gap between template-based rigidity and dataset-specific learning, offering a more adaptable solution for real-world software maintenance tasks while still needing improvement in generalizing to diverse, complex bugs.

Interpretation of Results: The results suggest that fine-tuned LLMs can generalize beyond seen bug-fix pairs by leveraging patterns acquired through both pretraining and supervised fine-tuning. For RQ1, the model demonstrated good generalization well to unfamiliar bugs, especially when the target structure or syntax had similarities to training data. This indicates that the model effectively learns patterns that can be applied to new but structurally similar bug types. For RQ2, DeepSeek excelled at fixing frequent syntactic bugs, such as identifier renaming and argument reordering, which are common in the training data. However, it struggled with more complex bug types, such as those involving intricate logic, rare programming constructs, or cases requiring multi-function or interprocedural understanding. This suggests that while the model can internalize domain-specific knowledge and structural repair patterns, it faces challenges in broader semantic reasoning and context integration. These limitations highlight the need for further model advancements to handle more nuanced and context-dependent repair scenarios effectively.

Limitations and Areas for Improvement: While the results are promising, there are several limitations that need to be addressed for further improvement:

- Context Scope: The use of method-level context offers a good trade-off between performance and computational cost, but bugs requiring class-level or project-wide understanding may not be effectively repaired. Expanding the model's ability to consider larger contexts could improve its performance on more complex bugs that span multiple methods or files.
- Bug Type Coverage: The datasets used, RunBugRun and CTSSB-1M, are biased toward specific types of single-statement bugs. This bias may limit the model's ability to perform well on rarer or more complex bugs that involve deeper reasoning or less common coding patterns. Future work should focus on diversifying training datasets to cover a broader range of bug types.
- Lack of Multi-Statement Support: The current model is tailored for single-statement bug repair. Extending the model to multi-line or multi-function bugs is an open challenge. Multi-statement bugs often require a more comprehensive understanding of the program's logic and flow, which the current method-level approach may not capture.

These findings suggest that while LLM-based APR can substantially improve automated debugging tools, future research should focus on scaling the context, enhancing semantic reasoning, and diversifying training datasets to handle more complex, multi-statement bugs.

6 RELATED WORK

Previous Studies: Similar studies have been conducted in this area that have greatly contributed to the progress of APR, focusing on both template-based and machine learning-driven approaches.

- Early work in APR, such as GenProg [16], employed genetic programming to evolve program patches based on test suites, demonstrating the potential for automated bug fixing.
- Template-based APR (such as TBar) – Liu et al. [21] demonstrated the efficiency of TBar by refining predefined templates and emphasized the exploration of hybrid approaches to improve generalization.
- Learning-based APR (such as CoCoNuT, Recoder) – Xia et al. [34] have focused on the use of pre-trained LLMs for program repair, comparing them to traditional APR models and highlighting the potential for generalization beyond training data. However, further improvements are needed to fully integrate automated bug fixing with real-world software maintenance. More recent research, including CIRCLE [43] and RSRepair [2], has introduced more sophisticated learning-based techniques, using large-scale datasets to train models capable of repairing bugs in a more data-driven manner.
- PySStuBs and ManySStuBs4J datasets - Kamienski et al. [9] and Karampatsis and Sutton [10] analyzed the frequency and distribution of single-statement bugs extracted from open-source Java and Python projects, highlighting that small code changes often fix significant issues.
- OpenAI's Codex [3] and GitHub Copilot leverage models such as GPT-3, pre-trained on massive code corpora, to generate code and suggest repairs. These models, although powerful, struggle with bug generalization beyond the scope of their training datasets and often generate syntactically correct but logically incorrect fixes.

Limitation of Existing Work: Despite substantial progress in Automated Program Repair (APR), the existing literature reveals persistent gaps that limit the robustness and generalizability of repair systems. First, template-based approaches (e.g., TBar) rely heavily on predefined fix patterns, making them rigid and unable to adapt to unfamiliar or complex bug types. These approaches often struggle when confronted with novel or unseen bugs that deviate from pre-defined templates. Second, learning-based models like CoCoNuT and Recoder [2] often overfit training distributions and lack robustness when applied to bugs that deviate from the patterns seen during training. This overfitting limits their ability to handle diverse real-world bug scenarios. Third, most APR systems are evaluated on isolated benchmarks, such as ManySStuBs4J or PySStuBs, which focus primarily on single-statement bug reports. This narrow evaluation scope does not account for the structural diversity or out-of-domain bugs that are commonly encountered in real-world software repair. Consequently, there is a disconnect between research evaluations and the practical challenges of software maintenance. In addition, few studies have performed a comprehensive comparison of LLM-based APR with classical and modern baselines under consistent evaluation settings.

Our study addresses these gaps in several ways. Although we focus primarily on single-statement bugs, we introduce cross-dataset generalization as a core research objective by evaluating fine-tuned LLMs on both in-domain datasets (RunBugRun, CTSSB) and out-of-domain test sets (QuixBugs). This cross-domain evaluation helps assess the model's ability to generalize to bugs that may not have been

represented in the training data. We also employ an instruction-based fine-tuning strategy with a method-level context, enabling the model to learn repair logic beyond the surface-level syntax and adapt to new bug types. To bridge the gap between LLM-based research and traditional APR approaches, we conduct a direct comparison with existing tools, including template-based, learning-based, and LLM-based models, on a unified benchmark. This approach positions our work as a step toward creating more scalable, generalizable, and context-aware solutions for single-statement bug repair, advancing the field toward more practical real-world applications.

Comparative Analysis: Unlike template-based methods such as TBar, which rely on predefined fix patterns, the LLM model (such as DeepSeek) learns from a wide range of bug-fix pairs within the dataset, enabling it to generate novel fixes. The LLM model can also repair incorrect code segments while preserving the surrounding context, avoiding the sequential fix generation that template-based systems often employ. This results in more precise and context-aware repairs, as the model can adapt to the nuances of the bug and its surrounding code.

APR models relying on historical bug-fix pairs often suffer from dataset bias, as their performance is limited to the types of bugs seen in their training data. These models excel at fixing previously encountered bugs, but struggle with novel defects outside of the scope of their training datasets [32]. This overfitting problem restricts their practical applicability in real-world software development, where bugs can be highly diverse and unpredictable. In contrast, our study evaluates how DeepSeek-based APR generalizes to unseen bugs, particularly those from a held-out test set, simulating real-world debugging scenarios. Unlike template-based APR approaches, which rigidly apply fixed templates, our model is designed to recognize common bug-fix structures while maintaining the flexibility to generate novel repairs when necessary.

LLMs like ChatGPT have been shown to repair bugs through iterative dialogue, achieving around 48% success on real-world defects [35]. Although this approach has conversational flexibility, our study focuses on fine-tuning DeepSeek using structured bug-fix pairs to prioritize precision over conversational flexibility. This distinction emphasizes the trade-offs between generalization and task-specific optimization, highlighting how fine-tuning on a well-defined bug-fix dataset enhances the model's ability to deliver accurate and reliable repairs in focused repair scenarios, as opposed to the broad, less structured fixes offered by more general conversational models.

Innovation: Instead of the generic form of LLM, this research fine-tunes DeepSeek on several datasets for targeted single-statement bug repairs.This approach utilizes the generalization powers of LLM combined with structured learning from bug-fix datasets rather than relying on fixed patterns (TBar) or dataset training (CoCoNuT). LLMs can overcome the rigidity of template-based APRs and the dependence on the dataset of learning-based methods. Template-based methods (e.g., TBar) achieve high accuracy for recurring bugs but fail for novel patterns, while learning-based models (e.g., CoCoNuT) depend heavily on dataset quality. Our LLM-based approach addresses these limitations by leveraging pre-training to encode broad programming knowledge, enabling the model to infer fixes for both known and unseen bugs without rigid templates or dataset constraints.

Improvements: The expected improvements over existing methods include:

- The LLM model is expected to outperform template-based APR and supervised leaning approaches in handling novel bug types.
- This research aims to achieve higher repair success rates with minimal supervision, in contrast to template-based methods.

This study brings together APR methods based on templates and learning to offer a more scalable solution that is more suitable for real-world applications.

7 THREATS TO VALIDITY

To ensure a comprehensive and transparent evaluation of our research, we acknowledge several potential threats to the validity of our study. These threats pertain to the confidence that our results can be interpreted, generalized, and trusted to reflect the actual performance and applicability of the fine-tuned DeepSeek model for Automated Program Repair (APR).

Internal Validity: Threat: Training and evaluation datasets can introduce unintentional biases that influence the performance of the model. Since bug types are not uniformly distributed across datasets (e.g. RunBugRun, CTSSB), the model may have implicitly learned to perform better on overrepresented bug categories.

Mitigation: We tried to balance training samples and included diverse pairs of bug-fix in the datasets. The model was also evaluated on a previously unreported set of unseen bugs to reduce the risk of overfitting. However, further stratified sampling and validation across varying bug categories could strengthen internal validity.

External Validity: Threat: Our findings are based solely on single-statement bugs in Python. As such, it is uncertain whether our conclusions hold for more complex, multi-line bugs, other programming languages, or different codebases not represented in our datasets.

Mitigation: To improve generalizability, we selected datasets from a wide range of open-source repositories and focused on preserving method-level context. Future extensions could include multi-line bug scenarios and broader language coverage (e.g., C++, JavaScript) to enhance external validity.

Construct Validity: Threat: The evaluation metrics used (Exact Match, AST Match, CodeBLEU, Syntax/Dataflow Match) may not fully reflect the real-world effectiveness of code repair, especially in terms of functional correctness or compilation success.

Mitigation: Although our metrics capture structural and semantic similarity, we acknowledge the need for execution-based validation (e.g., running repaired code against test cases). In future work, we plan to incorporate functional testing and real-world debugging scenarios to more accurately measure the true effectiveness of repair.

8 CONCLUSION

This study was designed to address a critical limitation in Automated Program Repair (APR): the inability of traditional and learning-based methods to generalize beyond predefined fix patterns and limited datasets. The primary goal was to investigate whether a fine-tuned Large Language Model (LLM), specifically DeepSeek, could learn and enhance the ability to repair single-statement bugs in Python, including those that were not present in the training data.

Key Observations:

- DeepSeek fine-tuning significantly improved repair performance over its initial version, particularly in repairing previously unseen bugs, demonstrating its potential for generalizing across a wide range of programming issues.
- The fine-tuned model excelled at handling common syntactic errors, such as identifier renaming and argument reordering, but faced challenges with more complex semantic bugs, especially those involving higher-level logic or multi-statement repairs.

The main contribution of this work is the development of a hybrid APR model that combines the strengths of template-based accuracy and learning-based adaptability. Using method-level training, our approach improves repair precision without relying on rigid templates or the need for extensive hand-crafted rules. This allows the model to generate more context-aware and novel repairs, marking a significant step forward in the field of APR.

Contributions: This study contributes the following to the field of software engineering and program repair:

- A DeepSeek-based APR framework fine-tuned on large-scale Python datasets (CTSSB-1M, RunBugRun).
- Empirical evaluation metrics and methodology for assessing LLM-based APR systems.
- A demonstration that LLMs can generalize bug-fix patterns better than template-based systems, moving the field toward more adaptable and intelligent repair tools.
- Insights into the role of LLM generalization in handling unseen bug types.

Limitations and Future Work: Our current approach is limited to single-statement bugs and focuses on syntactic and semantic similarity rather than functional execution correctness.

Future research should explore:

- Extending the model to handle multi-line and semantic bugs.
- Adding test-case-driven validation to measure functional correctness of repairs.
- Expanding the datasets to include more languages and real-world bugs from industrial repositories.
- Integrating the model into development tools (e.g., IDEs, CI/CD pipelines) for real-time code repair.

REFERENCES

[1] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.

[2] Heling Cao, Fangzheng Liu, Jianshu Shi, Yonghe Chu, and Miaolei Deng. 2021. Automated repair of Java programs with random search via code similarity. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 470–477.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[4] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[5] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. *CoRR* abs/2401.06066 (2024). https://arxiv.org/abs/2401.06066

[6] Haichuan Hu, Ye Shang, Guolin Xu, Congqing He, and Quanjun Zhang. 2024. Can GPT-O1 Kill All Bugs? An Evaluation of GPT-Family LLMs on QuixBugs. *arXiv preprint arXiv:2409.10033* (2024).

[7] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[8] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[9] Arthur V Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. 2021. Pysstubs: Characterizing single-statement bugs in popular open-source python projects. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 520–524.

[10] Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th international conference on mining software repositories*. 573–577.

[11] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 802–811.

[12] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25 (2020), 1980–2024.

[13] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–379.

[14] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.

[15] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[17] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2025. The Evolution of Automated Software Repair. *IEEE Transactions on Software Engineering* 51, 3 (2025), 870–873. https://doi.org/10.1109/TSE.2025.3533309

[18] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic Program Repair. *IEEE Software* 38, 4 (2021), 22–27. https://doi.org/10.1109/MS.2021.3072577

[19] 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 *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Vancouver, BC, Canada) *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 55–56. https://doi.org/10.1145/3135932.3135941

[20] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[21] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 31–42.

[22] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 658–662.

[23] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 298–312.

[24] 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 *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[26] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[27] Jalaj Pachouly, Swati Ahirrao, Ketan Kotecha, Ganeshsree Selvachandran, and Ajith Abraham. 2022. A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools. *Engineering Applications of Artificial Intelligence* 111 (2022), 104773.

[28] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2339–2356. https://doi.org/10.1109/SP46215.2023.10179324

[29] Julian Aron Prenner and Romain Robbes. 2023. RunBugRun–An Executable Dataset for Automated Program Repair. *arXiv preprint arXiv:2304.01102* (2023).

[30] Mohaimenul Azam Khan Raiaan, Md. Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. 2024. A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. *IEEE Access* 12 (2024), 26839–26874. https://doi.org/10.1109/ACCESS.2024.3365742

[31] Cedric Richter and Heike Wehrheim. 2022. TSSB-3M: Mining single statement bugs at massive scale. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 418–422.

[32] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 479–490.

[33] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179* (2022).

[34] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[35] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.

[36] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Multi-objective fine-tuning for enhanced program repair with llms. *arXiv preprint arXiv:2404.12636* (2024).

[37] 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 (2021), 110825.

[38] Jooyong Yi and Elkhan Ismayilzada. 2022. Speeding up constraint-based program repair using a search-based technique. *Information and Software Technology* 146 (2022), 106865.

[39] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1274–1286.

[40] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward better evolutionary program repair: An integrated approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–53.

[41] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A systematic literature review on large language models for automated program repair. *arXiv preprint arXiv:2405.01466* (2024).

[42] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 535–547.

[43] Ying Zhou, Qingqing Li, Ruiting Zhang, Wenhua Zhang, Shenqiang Yan, Jinjin Xu, Shuyue Wang, Minming Zhang, and Min Lou. 2020. Role of deep medullary veins in pathogenesis of lacunes: Longitudinal observations from the CIRCLE study. *Journal of Cerebral Blood Flow & Metabolism* 40, 9 (2020), 1797–1805.

[44] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931* (2024).