DeepPERF: A Deep Learning-Based Approach For Improving Software Performance

Spandan Garg*
spgarg@microsoft.com
Microsoft
Redmond, Washington, USA

Roshanak Zilouchian
rozilouc@microsoft.com
Microsoft
Redmond, Washington, USA

Colin B. Clement
coclemen@microsoft.com
Microsoft
Redmond, Washington, USA

Neel Sundaresan
neels@microsoft.com
Microsoft
Redmond, Washington, USA

ABSTRACT

Improving software performance is an important yet challenging part of the software development cycle. The wealth of open source software development artifacts available online creates a great opportunity to learn the patterns of performance improvements from data. In this paper, we present DeepPERF, which is a data-driven approach to software performance improvement using large transformer models. We pretrain a transformer model on English and Source code corpora and then finetune it on the task of generating performance improvement patches for C# applications. We collect a test dataset of 132 examples with a wide variety of performance improvement patches made by C# developers to open source repos on GitHub. In our evaluation, we find that our best model is able to generate the same performance improvement suggestion as the developer fix in ~60% of the cases, getting ~36% of them verbatim. Additionally, we evaluate DeepPERF on 50 open source C# repositories on GitHub with both benchmark and unit tests and find that our model is able to suggest valid performance improvements that can improve both CPU usage and Memory allocations. So far we've submitted 17 pull-requests with 26 different performance optimizations and 6 of these PRs have already been approved by the project owners.

1 INTRODUCTION

Inefficient code sequences can cause significant performance degradation and resource waste, referred to as performance bugs. Detecting and fixing performance bugs is important as they can lead to a poor user experience, reduced throughput, increased latency, and wasted resources. Given the increasing emphasis on efficient use of resources, detecting and fixing performance bugs has become more important. However, performance bugs are often hard to detect as they typically don't cause system failure and are sometimes dependant on user input. These bugs can even be introduced by expert developers in well-known applications [13, 19, 20, 25] and can propagate quickly due to prevalence of code re-use. Even when performance bugs are detected, they tend to be difficult to fix than non-performance bugs [21, 29]. As a result, better tool support is needed for fixing performance bugs.

In recent years, a variety of performance bug detection approaches have emerged both in the industry and academia to assist developers with the identification of performance issues in different scenarios. However, most of these tend to focus on detecting specific types of performance bugs. For instance, a set of tools have been developed for detecting runtime bloat [34, 36, 10], low-utility data structures [35], database related performance anti-patterns [6], false sharing problem in multi-threaded software [17], detecting inefficient loops [28, 21, 33], etc. Often these techniques rely on static code analysis or some pre-defined set of rules to detect these performance issues. Aside from performance, rule-based detection and static analyzers have been widely adopted for other problems as well such as detecting security vulnerabilities and functional bugs. However, building an analyzer is a non-trivial task because balancing between being precise, to only report the correct bugs, and coverage, in ensuring that the analysis covers all bugs following a similar pattern, can be a challenge. Creating this balance manually is difficult and can result in analyzers with a high false positive rate and for the ones that do manage to be precise, experts need to pay a high maintenance cost in order to maintain their high precision [4].

Due to the widespread availability of open source repositories, there is an opportunity to learn patterns of software improvements directly from mined data. With the recent rise of large transformer models, transformer-based approaches have been shown to achieve state-of-the-art performance, not only in various Natural Language Processing (NLP) problems, but also a variety of software engineering tasks such as code-completion [30], documentation generation [7], unit test generation [31], bug detection [9], etc. In this paper, we draw inspiration from these techniques, in an attempt to solve the problem of automatically suggesting software performance improvements. We present an approach that leverages large transformer models to suggest changes at source code level to improve an application's performance. We first pretrain our model on masked language modelling (MLM) tasks [7] on English text and source code taken from open source repositories on GitHub, followed by finetuning on thousands of performance commits made by .NET developers. Through our evaluation, we show that our approach is able to recommend patches to provide a wide-range of performance optimizations in C# applications. Further, by suggesting changes to a set of real world repositories and measuring the impact of our suggestions through benchmark tests, we show that our changes provide actual performance gains. Several of our

*Corresponding Author

changes have already been accepted by the developers of these projects, showing that our suggestions are considered to be correct and useful by the project owners.

In summary, our work makes the following main contributions:

- We propose a novel framework called DeepPERF, which is able to find performance optimization opportunities in an application and uses machine learning to automatically generate patches with performance improvements.
- We collect a test dataset with 132 performance improvement changes made by developers to various C# repositories on GitHub. Each of these examples were confirmed to be performance improvements with two C# performance experts. To encourage future research, we release this dataset alongside this paper.
- We conduct two types of evaluations of our approach. We first conduct a static evaluation on the manually curated dataset and show that our best model can produce fixes for up to 60% of the examples, getting more than 35% responses verbatim to the developer patch. Secondly, we perform a dynamic evaluation of DeepPERF on 50 C# repositories on GitHub, to see if it can generate performance optimizations for these repos. We validate the correctness of our changes using unit tests and verify that they provide actual performance improvement by running benchmark tests. We then submit 17 PRs to the repos with these changes. 6 of our PRs have been approved showing that our approach can produces fixes that are considered useful by developers.
- Finally, we conduct an analysis on the generated changes that are incorrect or are not performance improvements and how effectively our pipeline filters them out.

2 MOTIVATING EXAMPLES

Figure 1 shows examples of two suggestions made by DeepPERF to two real open-source C# projects on GitHub. In the first example change, the code prior to the change makes use of LINQ [23]. LINQ expressions have an inherent allocation associated with them due to being executed as state machines. As a result, usage of LINQ on the application hot-path can lead to excessive allocations, which can in-turn lead to a spike garbage collection, potentially reducing the application's throughput. DeepPERF recognizes the underlying type of the enumerated variable is an array and that the use of LINQ call to skip the first position is unnecessary. It recommends a change to unroll the LINQ query and use an explicit for-loop, which starts indexing from 1. Furthermore, by running the repository's provided unit tests and benchmark tests, DeepPERF verifies its correctness by running unit tests and that this change reduces allocations compared to the code prior to the change. Looking at the benchmark results, we also notice a reduction in Gen 0 GC. The second change shows a repeated array allocation happening inside a while-loop within a recursive function. DeepPERF notices that the array is independent from the loop and the method call and suggests a change to hoist the array to a static member of the containing class. This ensures that only one instance of the array is ever allocated, which reduces the application's allocations.

[1]https://github.com/CreoOne/V/pull/1
[2]https://github.com/tihilv/Omtt/pull/1

V/V/Vector.cs

```
private void EnsureConsistentDimensionality(Vector[] vectors)
{
    // ...
    foreach(Vector vector in vectors.Skip(1))
        if (dimensions != vector.Dimensions)
            throw new DimensionalityMismatchException();
}
```

```
private void EnsureConsistentDimensionality(Vector[] vectors)
{
    // ...
    for (int i = 1; i < vectors.Length; i++)
        if (dimensions != vectors[i].Dimensions)
            throw new DimensionalityMismatchException();
}
```

Omtt/Omtt.Parser/Lexical/TemplateModelLexicalParser.cs

```
private void ProcessContent(ParsingSource source, List<Lexem>
      result, bool textMode, String[] exitSymbols)
{
    var symbols = (textMode ? TextLiterals : SymbolLiterals)
                   .Union(exitSymbols)
                   .ToArray();

    Boolean foundExitSymbol = false;
    while (source.Any() && !foundExitSymbol)
    {
        // ...
        else if (!foundExitSymbol && symbol !=
              MarkupLiterals.CloseSymbol && textMode)
            ProcessContent(source, result, false, new []
                  {MarkupLiterals.CloseTagSymbol,
                  MarkupLiterals.CloseExpression});
    }
}
```

```
private static readonly String[] CloseTagLiterals = new[]
{
    MarkupLiterals.CloseTagSymbol,
    MarkupLiterals.CloseExpression
};
// ...
private void ProcessContent(ParsingSource source, List<Lexem>
      result, bool textMode, String[] exitSymbols)
{
    var symbols = (textMode ? TextLiterals : SymbolLiterals)
                   .Union(exitSymbols)
                   .ToArray();

    Boolean foundExitSymbol = false;
    while (source.Any() && !foundExitSymbol)
    {
        // ...
        else if (!foundExitSymbol && symbol !=
              MarkupLiterals.CloseSymbol && textMode)
            ProcessContent(source, result, false,
                  CloseTagLiterals);
    }
}
```

Figure 1: Two examples of the kinds of changes DeepPERF is able to suggest. The first change, taken from a PR[1]we submitted shows a patch recommended by DeepPERF to a C# project on GitHub. The change is to unroll a LINQ query into an explicit for-loop. This change results in lower allocations and Gen 0 garbage collection compared to the code prior to the change. The second change is from another PR[2]to a different C# project. In this change, the code is repeatedly allocating the same array inside a loop. Since the array is independent of the loop or the method itself, DeepPERF suggests a change to hoist the array to a static member of the containing class, which results in a reduction in allocations.

Pull-requests containing both of these changes were submitted to the corresponding GitHub repos and have since been approved by their owners within the same day.

3 OUR APPROACH

Figure 2 shows an overview of our pipeline. Below we describe our model training pipeline. We begin by first describing how we take an English-pretrained BART-large model and further pretrain it on Source code. We then describe data collection and example generation for finetuning. This is followed by a description of our two-step finetuning process on the examples generated by the example generation step. We then explain how we identify methods that are tested by both benchmark and unit tests. We use our model to generate examples for each of the identified methods using our example generation and generate performance improvement suggestions using our model. Finally, we verify our fix's correctness and validity with the help of unit tests and benchmark tests.

3.1 Pretraining

Prior work in leveraging transformers for various software engineering tasks has shown that pretraining on code snippets can significantly improve model performance on specific downstream tasks such as method and docstring prediction [7]. Inspired by such prior work, we pretrained sequence-to-sequence transformers using a span-masking objecting [15] on publicly available source code data. The span-masking objective essentially replaces random spans of input tokens with a `<MASK>` token, and the model is trained to predict all the tokens replaced by the mask, separated by mask tokens.

For pretraining, we collected 51K GitHub repositories with ≥5 stars that were composed primarily of shell scripts, resulting in 328K unique scripts with 54 million total lines of code. We then pretrained our 406M parameter transformer BART-large on this corpus for 60 epochs on four Nvidia Tesla V100 16GB GPUs, ~48 GPU-hours total for the larger model.

3.2 Data Collection

We rely on open source C# repositories on GitHub for our data. We use a repo's star count and commit recency as a metric to determine if it is popular. We take repositories that have a star count of ≥5 and a commit within the last 5 years. This leaves us with ~45k repositories. Below we describe how we use commit data from these repos to generate examples for finetuning.

3.2.1 Crawling Commits. After cloning these projects, we crawl the *main* branch's commit history and generate examples for finetuning. Within each commit, we parse the modified C# files that end with the extension "*.cs" using the tree-sitter parser. We take each class method that has been modified by the commit as a focal method and generate an example for it using the process described in example generation. In addition to the code changes, we also collect the commit message and title, which we use to design a keyword-based heuristic to tell if a commit is performance related.

3.2.2 Example Generation. For each class, we collect all the class-level metadata such as class and method names, bodies, signatures, class attributes as well as file-level metadata such as the using statements within the file, which are used in C# to import methods from other files within the repo or external packages. We also generate a static call-graph of methods within the class using its parse tree to determine the caller-callee relationship among the methods.

Next, we apply some pre-processing steps on the method bodies by normalizing white-space and removing comments. This allows us to ignore any trivial modifications. We use the method signatures to identify the corresponding versions of the method in the before and after files. We then compare the normalized method bodies between the two versions of the file and generate an example for the methods whose bodies have changed. From here on, we refer to these modified methods as focal methods.

Performance changes often require changes beyond the focal method itself (as seen in the second example in Figure 1), such as adding new class level attributes or changes to the caller-callee methods or adding new import statements to the file, etc. Following this reasoning, it appears logical to also provide this information to the model in generating the output. Many past works have shown that including additional class/file-level context along with the focal method is helpful in providing the model with useful information for generating the output [31, 9]. We believe that such contextual information would prove useful in generating performance patches as well. We describe the kind of contextual information we include in our input alongside the focal method and also explain our intuition for each kind of element.

Due to the input token window for BART-large being limited to only 1024 tokens, we construct the example input in an iterative fashion. We start by including the focal method in the example input, which is the most important part of the input as, in most cases, the bulk of the changes are expected to be located here. We indicate the focal method to the model by adding C-style comments (`/* edit */` and `/* end */`) before and after the focal method. We then add the following class level context elements, in the order below:

- *Using Statements:* These tell the model what import statements exist in the file and whether new imports will need to be added for the new methods or APIs used in the recommended changes.
- *Class Attributes:* These are the containing class's member attributes. They tell the model about the underlying types of the class attributes, which might be used within the focal method. This information could help the model learn patterns of performance improvements, which involve fixing incorrect usages of certain types of variables that may be leading to performance issues or recommending a more appropriate data structure for a task e.g. replacing `List<T>` with a `HashSet<T>`, etc.
- *Caller-Callee Methods:* These are the methods that directly make calls to or are, in turn, called by the focal method. This information can be useful when the model needs to make changes to the caller or callee of the method, which are often required to address performance issues. Examples include changes that involve hoisting/memoizing calls made by the caller to the focal method in order to reduce unnecessary computation.

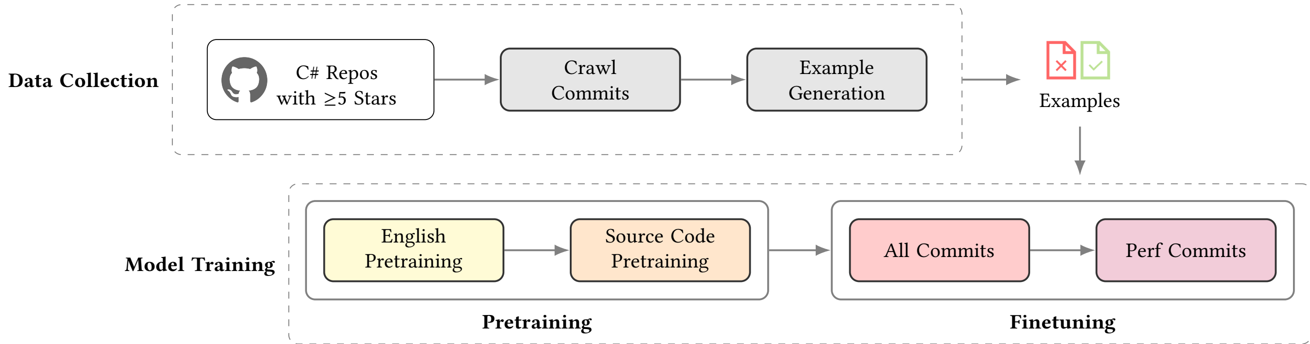

Figure 2: Our model data collection and training pipeline. We first crawl the commit history of all 45k C# repos with ≥5 stars on GitHub and generate examples for each modified method with various contextual elements important to performance (using statements, class attributes, caller-callee methods, etc.). For training, we first pretrain BART-large on denoising objectives over English text and source code, followed by a two-step finetuning. In our finetuning step, we first finetune the model on examples generated from all commits, followed by a smaller finetuning step done only on examples from commits where developer included a performance related keyword ("perf", "performance", "latency", etc.) in the title or description of the commit (complete list of keywords is provided in supplementary materials).

- *Method Signatures:* Finally, we add the signatures for any other methods that aren't caller or callee methods of the focal method. Due to limited token space, we are unable to add the bodies of each method. But we still think this information could be useful as including signatures could shed some light on the nature of the class itself and provide context as to what other methods are present in the class for the model to use in the generated patch.

Before adding each type of contextual elements from the above list to the example, we ensure that the resulting sequence will be within the allowed range of tokens i.e. ≤1024 tokens. If not, we discontinue adding further context elements and take the sequence thus far as the input. This way, we try to incorporate the as much context into the limited span of tokens while staying within the allowed limit.

For the output, in addition to changes to the focal method, we include any of focal method's caller-callee methods that are modified by the commit. We also include any additional imports that may have been added as well as class attributes defined/modified that are used by the focal method or modified caller-callee methods. This way we allow the model to output patches that make changes to not only the focal method but also the caller-callee methods, class attributes as well as add any new methods, attributes and import statements as needed. Figure 3, shows an example of an input output pair generated using the steps above.

Table 1: Number of commits and examples in our training data for the All Commits and Perf Commits finetuning steps.

Commit Type Data	# of Commits	# of Examples
All Commits	11M	16M
Perf Commits	535k	1.5M

3.2.3 Identifying Perf Changes. We divide the examples generated by the example generation step based on the title/message of the commit they come from. We look for performance related keywords ("perf", "performance", "latency", "slow", etc.) to tell whether or not a commit is performance related. Table 1 shows the number of commits and examples that come from performance-related commits. We provide a complete list of the keywords we use to collect these commits in the supplementary materials for reproducibility.

3.3 Finetuning

In this step, we use the examples generated by the example generation step, as described in Sec. 3.2.2. We finetune the pretrained model on the task of generating the performance patches, given the input sequence containing the focal method and surrounding context. Since the performance dataset is smaller, we perform a two-stage finetuning, where We first finetune our pretrained transformer model on examples from all commits. We call this the "All Commits" step. Our intuition is that this will teach the model how C# developers make changes. Following this, we further finetune the model on examples from performance commits to teach it specifically how to make performance changes. We call this the "Perf Commits" step. We split the finetuning data on the project level. We leave out leaving out two sets of test and validation repos, each containing 600 repos that are not included in either step's training data. We also dedupe the examples in each set as well as remove any near duplicates [1] among them to ensure no overlap between train and test data. We call the models resulting from the "All Commits" and "Perf Commits" finetuning steps DeepDev-CSharp and DeepDev-Perf, respectively. To show the impact of the "All Commits" step on model performance, we also finetune directly on only performance examples. We call this model DeepDev-Perf (Direct). In our evaluation, we compare the three models and discuss possible reasons for differences in their performances.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using Shared;
using Shared.Data.Extractors;
using Shared.Helpers;
using WindowsActivityTracker.Data;
using WindowsActivityTracker.Models;
```

Using Statements

```csharp
class DayFragmentationTimeline : BaseVisualization, IVisualization
{
   /* edit */
   public override string GetHtml()
   {
      var html = string.Empty;
      var orderedTimelineList = Queries.GetDayTimelineData();
      var sum = orderedTimelineList.Sum(i => i.DurationInSeconds);
      if (orderedTimelineList.Count <= 3 || sum < 5*60)
      {
         html += VisHelper.NotEnoughData(Dict.NotEnoughData);
         return html;
      }

      html += GetActivityVisualization(orderedTimelineList);
      return html;
   }
   /* end */
```

Focal Method

```csharp
   private string GetActivityVisualization(List<WindowsActivity> list)
   {
      var categories = list.Select(a => a.ActivityCategory).ToList();
      var html = string.Empty;
      html += @"<script type='text/javascript'>
            var onLoad=window.onload;
            window.onload=function()
            {
               if(typeof onLoad=='function')
               {
                  onLoad();
               }
            }";
      html += "};";
      html += "</script>";
      html += "<div style='align:center'><p id='hover'>" +\
            defaultHoverText +\
            "</p></div>";
      html += "<div id='" +\
            activityTimeline +\
            "'align='center'></div>";
      html += LegendHelper.GetLegendForCategories(categories);
      return html;
   }
```

Caller-Callee

```csharp
   private string defaultHoverText;
   private string activityTimeline;
```

Class Attributes

```csharp
   private string ReadableProcesses(List<WindowProcessItem> list);
   private string FormatWindowTitle(string windowTitle);
   private string FormatContextCategory(ActivityCategory category);
   private string FormatCategories(List<ActivityCategory> list);
}
```

Method Signatures

```csharp
using System.Text;
class DayFragmentationTimeline : BaseVisualization, IVisualization
{
   public override string GetHtml()
   {
      var html = string.Empty;
      var orderedTimelineList = Queries.GetDayTimelineData();
      var sum = orderedTimelineList.Sum(i => i.DurationInSeconds);
      if (orderedTimelineList.Count <= 3 || sum < 5*60)
      {
         html += VisHelper.NotEnoughData(Dict.NotEnoughData);
         return html;
      }

      _sb.Clear();
      GetActivityVisualization(orderedTimelineList, _sb);
      html += _sb.ToString();
      return html;
   }

   private void GetActivityVisualization(List<WindowsActivity> list,
                                          StringBuilder sb)
   {
      var categories = list.Select(a => a.ActivityCategory).ToList();
      sb.Append(@"<script type='text/javascript'>
            var onLoad=window.onload;
            window.onload=function()
            {
               if(typeof onLoad=='function')
               {
                  onLoad();
               }
            }");
      sb.Append("};");
      sb.Append("</script>");
      sb.Append("<div style='align:center'><p id='hover'>")
         .Append(defaultHoverText)
         .Append("</p></div>");
      sb.Append("<div id='")
         .Append(activityTimeline)
         .Append("'align='center'></div>");
      sb.Append(LegendHelper.GetLegendForCategories(categories));
   }

   private StringBuilder _sb = new StringBuilder();
}
```

Figure 3: An example of an Input-output pair example used in finetuning. As part of the input, along with the focal method, we include various class/file level context elements such as using statements, class-level attributes, caller/callee methods, method signatures as shown in the image. We add C-style comments (`/* edit */` and `/* end */`) before and after the focal method indicating its location to the model. In this case, the output consists of the modified versions of the focal and callee method, along with an additional using statement and class attribute added by the patch.

3.4 Using Unit Tests and Benchmarks

Many commonly used software systems and applications leverage benchmark tests in addition to performance profiling and load testing to monitor their performance across changes. For the purpose of this paper, by benchmark tests, we refer to the tests written using the BenchmarkDotNet library, which is the standard benchmarking library for C# applications. Shown in Figure 4 is an output summary of a BenchmarkDotNet test. BenchmarkDotNet automatically runs each benchmark test in the provided test suite multiple times and reports metrics such as the duration of a given benchmark test as well as the amount of memory that is allocated on average on each run. It can also give other information such as how frequently Generational GC is triggered. To run these benchmark tests we first find the folder within the repo that corresponds to benchmarking test suite based on whether BenchmarkDotNet NuGet package is mentioned within the build configuration file ("*.csproj") present within the folder.

We use performance benchmark tests for two purposes. First, to verify that our changes lead to a performance improvement either in terms of test duration or allocations. Second, we use them to decide which methods to suggest changes for as not all methods may be on the execution path of the benchmark tests included in a project to begin with.

In addition to benchmark tests, we also use the unit tests to ensure that the changes made by our model are correct because such models have been known to produce incorrect suggestions that could be badly formed or even buggy. In order to make sure our changes are actually tested by the provided unit test suite, we first measure its code coverage, only generating suggestions for methods that have a high line/branch coverage with the provided unit tests.

In summary, we generate suggestions for the methods that lie in the intersection of the above two sets of methods, i.e. methods that are being benchmarked as well as have high line and branch coverage under the provided unit tests. For each of the generated suggestions, we apply the code changes to the main branch of the repo and run the unit tests and benchmark tests to verify their correctness and performance benefits.

```
// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19044.1526 (21H2)
Intel Core i7-6600U CPU 2.60GHz (Skylake), 1 CPU, 4 logical and 2 physical cores
.NET SDK=6.0.102
  [Host]     : .NET 6.0.2 (6.0.222.6406), X64 RyuJIT
  DefaultJob : .NET 6.0.2 (6.0.222.6406), X64 RyuJIT


|                    Method |          Mean |         Error |          StdDev |   Gen 0 |   Gen 1 | Allocated |
|-------------------------- |--------------:|--------------:|----------------:|--------:|--------:|----------:|
|             SpanReadBytes |      10.82 ns |      0.254 ns |        0.250 ns |       - |       - |         - |
|            SpanReadFloats |   2,409.94 ns |     61.064 ns |      172.233 ns |       - |       - |         - |
|        SpanReadFloatArray |   4,744.82 ns |    134.336 ns |      393.986 ns | 12.9852 |       - |  27,408 B |
|       SpanReadDoubleArray |  14,250.78 ns |    319.221 ns |      921.026 ns | 25.6348 |       - |  54,792 B |
|            SpanReadDouble |  14,658.17 ns |    387.182 ns |    1,123.284 ns | 25.6348 |       - |  54,792 B |
| BinaryReaderReadAbsorbance | 250,693.74 ns | 10,387.692 ns | 29,970.867 ns | 85.2051 | 23.1934 | 263,244 B |
```

Figure 4: An output summary statistics of a BenchmarkDotNet test. The results table shows the test results for the various benchmark tests defined by the user for their application. These tests can be executed by using the "`dotnet run -c Release`" command in the folder containing the benchmark test suite. BenchmarkDotNet automatically runs each test multiple times and reports metrics such as the sample mean, std dev, as well as the memory allocated on average during each run. Additionally, it may also provide metrics such as the number of times generational GC was executed for each benchmark test.

4 EXPERIMENTS

We perform two kinds of evaluations on DeepPERF. We first conduct a static evaluation of DeepPERF models on a manually curated set of performance improvement changes collected from the commit history of a hold-out set of ~600 test repositories. Through this experiment, we intend to answer the following research questions:

- **RQ1**: Is DeepPERF able to provide a wide-range of performance optimizations and suggest changes similar to a C# developer?
- **RQ2**: Are both steps (All Commits and Perf Commits step) in our two-step finetuning necessary?

Secondly, we perform a dynamic evaluation of DeepPERF on a set of 50 C# repos on GitHub that contain both benchmark and unit tests to see if our suggested changes are correct (using unit tests) and actually lead to tangible real-world performance improvements (using benchmark tests). For suggestions where this holds true, we submit PRs containing these changes to the corresponding repo as well to verify whether developers consider our suggestions to be valid and useful. With this experiment, we intend to answer the following research questions:

- **RQ3**: How is the overall quality of DeepPERF suggestions (such as reasons for build errors, unit test failures, etc.) and what are some areas for improvement?
- **RQ4**: Is DeepPERF's able to suggest changes that lead to real performance improvements and by how much? Are these suggestions considered useful by the developers?
- **RQ5**: How effective are unit and benchmark tests in ensuring changes are correct performance improvements?

4.1 Static Evaluation

4.1.1 Manually Curated Dataset. As there were no pre-existing datasets of performance bugs in C#, for the purposes of our evaluation, we decided to collect such a dataset by finding performance improvements in the commit history of ~600 test repos that our models had not seen before. The reason we couldn't simply take all examples generated from commits with a performance related keyword in their title/description was because this heuristic was created to prioritize recall with the intention to find as many perf commits as possible for finetuning. As a result, it could result in false positives by mistakenly tagging non-performance commits as being performance related. Even if the commit actually is performance related, there may be some changes within the commit itself which aren't performance improvements (bug fixes, refactoring changes, etc.) as developers often squash multiple changes into a single commit.

In order to make our search more efficient and increase the likelihood of finding performance related changes, we manually examine performance commits that make changes to a single "*.cs" file. We manually examine a set of 1500 such commits and found 132 examples which were performance related. The authors confirmed each of these examples with two C# and .NET performance experts. To summarize this dataset, we classify the collected examples into the following categories based on our understanding of performance changes in C#:

- (Category 1) High Level Changes: These consist of algorithmic changes that rely on modifications to the overall code structure to improve performance. They could include changes such as hoisting calls/allocations to an outer scope, adding caching/memoization to avoid repeated computation, introducing a fast-path, etc.
- (Category 2) Suggesting Different API/Data Structure: These are language/API specific changes to replace or remove an existing API or data structure usage in favor of a better alternative. These could include changes like removing LINQ by unrolling queries into explicit loops, suggesting a different data structure better suited for the task like replacing List with a HashSet, etc.
- (Category 3) Improving Existing API/Data Structure Usage: These are also language/API specific changes, but suggest modifications to existing usage of an API or data structure when deemed incorrect or sub-optimal. These may include changes like condensing LINQ queries to be more optimal, fixing incorrect uses of a data structure, using a better suited overload of a library function, etc.

Table 2 shows a summary of the examples in our manually curated test set. We show examples for each of the 3 different categories of performance transformations as well as the number of examples within that category. We've provide this dataset with each example labelled with above categories in the supplementary material. In total, we identified 119 distinct performance optimizations in this test set with a mix of both high and low level changes, demonstrating that it consists of a wide-range performance improvements.

4.1.2 Evaluation Method. For each example in the manually curated dataset, we sample 800 hypotheses from our model and take the top 200 hypotheses, based on the average likelihood of tokens. We picked 200 because it was small enough for us to manually verify with experts, for cases where the model doesn't get the response verbatim and yet large enough to allow the model sufficient opportunity to succeed. We use the following methods to evaluate our models' suggestions:

Table 2: Three categories of performance issues in manually curated dataset.

Change Category	Examples of Performance Optimizations	# of Examples
High Level Change (C1)	Memoize results using `Dictionary/ConcurrentDictionary` Hoist computation/allocation to outer scope (loop, method, class, etc.) Introduce fast-path to avoid unnecessary computation Re-use types like `List`, `Dictionary`, `StringBuilder`, etc.	29
Suggest Different API/Data Structure (C2)	`string.Concat(), operator+(string, string)` → `StringBuilder`, `IEnumerable<T>.ToList()` → `IEnumerable<T>.ToHashSet()`, Remove LINQ usage (E.g.`List<T>.Any()/Count()` → `List<T>.Count`, etc.), etc.	71
Improve Existing API/Data Structure Usage (C3)	Condense/Optimize LINQ queries (E.g. `Count()` → `Any()`, `Where(<lambda>).First()` → `First()`, etc.), `Dictionary<K, V>.ContainsKey()` → `Dictionary<K, V>.TryGetValue()`, `return new List<T>()` → `return Array.Empty<T>()`, `new List<T>()` → `new List<T>(SIZE)`, `StringBuilder.Append(string.Format())` → `StringBuilder.AppendFormat()`, `string.Trim() == string.Empty` → `string.IsNullOrEmpty()`, etc.	34

- **Correctness & Ranking**: We consider a patch to be correct if it is either a verbatim match with the developer patch, or has some slight variations (variable names, braces, etc.), but is otherwise semantically equivalent to the developer patch. Since we are unable to compile the code changes for these projects, we consult two .NET experts to verify the semantic equivalency of non-verbatim suggestions.
- **Syntax Errors**: We use a parser like tree-sitter to find how many syntactical mistakes each model made in the top-200 suggestions.
- **CodeBLEU**: We measure the CodeBLEU [24] scores of the model suggestions found to be correct with the developer patches. We pick CodeBLEU because, unlike BLEU it takes into account the syntactic and semantic features of codes by checking for code syntax similarity via abstract syntax trees (AST) comparison as well as comparing code semantics through comparison of data flow between the two programs. This is done in addition to n-gram matching of BLEU. The score is then expressed as a weighted sum of the individual scores. We use the hyperparameters that were shown to have the highest correlation to human scores in the study i.e. $\alpha, \beta, \gamma, \delta = 0.1, 0.1, 0.4, 0.4$.

Table 3: Summary of the results of our three models over the manually curated dataset.

Model	Top-K Accuracy %				Verbatim Response %	CodeBLEU
	1	10	100	200		
DeepDev-CSharp	2.3	15.9	36.3	37.9	21.6	68.3
DeepDev-Perf (Direct)	3.0	18.2	39.4	41.7	26.1	70.6
DeepDev-Perf	2.3	19.7	53.0	59.8	35.8	70.7

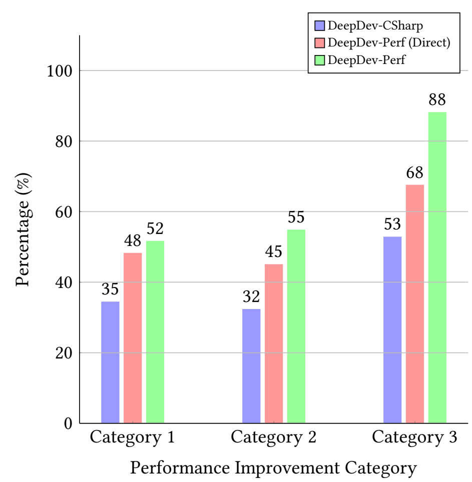

Figure 5: Performance of our models on the three categories of performance issues: High Level Changes (Category 1), Suggesting Different API/Data Structure (Category 2), Improving Existing API/Data Structure Usage (Category 3). We can see that the DeepDev-Perf model (green) tends to perform the best among the models in all three categories, followed by DeepDev-Perf (Direct) and DeepDev-CSharp.

4.1.3 *Ability To Suggest Variety Of Improvements (RQ1)*.

Table 4 and Figure 6, shows the results of the 3 models mentioned at the end of Sec. 3.3 over the collected examples using the metrics above. We see that our best model is able to solve ~60% of the examples in our dataset, getting ~36% verbatim with the developer

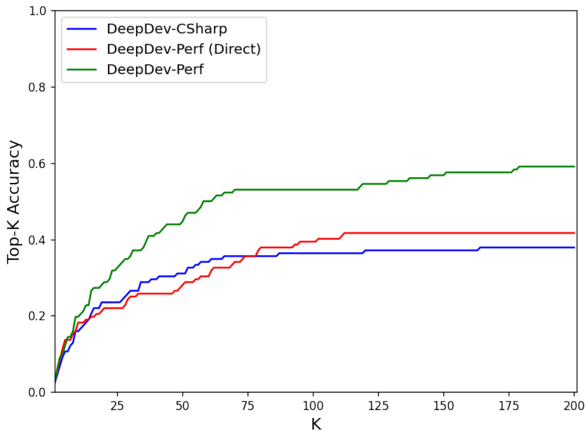

Figure 6: Top-K accuracy plot of our models on the manually curated dataset dataset. We can see that DeepDev-Perf achieves the best Top-K accuracy among the 3 models and continues to tend upwards as K grows.

fix. The correctness of suggestions that were not verbatim was verified with two C# performance experts, who aren't in the author list. The main reasons for dissimilarities were the model suggesting different variable names or other slight variations like using the `var` keyword instead of the variable's type or using a `for`-loop as opposed to a `foreach` loop where both are appropriate, difference in order of statements where relative order does not matter (such as `using` statements at the start of file), etc. Figure 5 shows the performance of our models in the 3 categories of performance changes from the previous section. We see that our best model is able to fix >50% of problems in each category.

RQ1: We conclude that our best model, DeepDev-Perf, is able to provide fixes a wide variety of performance optimizations by achieving close to 60% Top-200 accuracy on our dataset that contains a wide range of high-level algorithmic and low-level API/Data structure related performance changes. Furthermore, it is able to match the developer's suggestions verbatim in more than 35% of the cases.

4.1.4 ***Need For Two-Step Finetuning (RQ2).*** The major difference between the DeepDev-Perf model and the DeepDev-Perf (Direct) is that DeepDev-Perf is first finetuned on examples generated from all C# commits, followed by a smaller finetuning step on examples generated using commits with a performance related keyword in commit title/description. On the other hand, DeepDev-Perf (Direct) is finetuned directly on performance examples. Comparing the results of the two, we can see that the model that's been finetuned first on all commits tends to do better in terms of Top-K accuracy as well as gets more responses verbatim. Additionally, we can see in Figure 5 that it is also able to out-perform the other two models in all performance change categories. We hypothesize that this is because finetuning on all commits allows the model to learn better representations for code by seeing more examples of how changes are made by C# developers, since the dataset for All Commits step is almost 10x larger than the one containing only examples from perf commits. Another possible reason could be that there may be some performance related changes that aren't explicitly annotated with a performance keyword in the commit title and description, as developers sometimes don't explain every change they make in their commits and squash multiple changes into a single commit, mentioning only the most important in the commit message. In fact, we found several examples of a commits in our training data containing some of the performance transformations in Table 2, with no performance related discussion in the commit title or description. While we have no way of running the code changes, we verified these examples with performance experts and they confirmed that these changes are likely to have an impact on the application's performance if the code gets executed often. Furthermore, the presence of such "phantom" performance changes in the All-Commits data may also explain how the DeepDev-CSharp model, which has not been finetuned on performance commit data directly still manages to get 38% Top-200 accuracy and 22% of the responses verbatim.

RQ2: In summary, we conclude that both finetuning steps were necessary as DeepDev-Perf clearly demonstrates better performance than the other two models, which were trained excluding one of the two steps, on the overall dataset as well as in each category of performance bugs.

4.2 Dynamic Evaluation

4.2.1 Dataset. In this experiment, we evaluate DeepPERF's ability to suggest performance improvements to 50 open source projects on GitHub that were not previously seen by our model. We picked projects that contained both benchmark and unit tests, which we then use to validate the model's suggestions. Following the method outlined in Sec. 3.4, we identify which methods are tested by both the benchmarks and the unit tests. We found a total of 201 such methods across the 50 test repos. We then generate examples for each of these methods, including contextual information as described in Example generation step (Sec. 3.2.2). We use our best model, DeepDev-Perf, and sample 800 suggestions for each of these examples. This time, we take only the top 100 suggestions and validate them using tests. The reason we pick a smaller number is because compiling and running benchmark tests for the changes one by one is time consuming. Therefore, we limited the suggestions we try out to the top 100. We do not expect this to have a major impact on our model's performance because, as we saw in the static evaluation, the model was able to make majority of performance improvement suggestions within its top 100 suggestions.

4.2.2 Syntax Check. We start by filtering out changes that are syntactically incorrect. We found that ~10% of the suggestions had a syntax error. Most of there were due to truncation or repetition when generating long outputs, which are known problems when generating text using such language models.

4.2.3 Running Unit Tests. We then run unit tests for each of the remaining ~90% suggestions. This step filters out suggestions that

Table 4: Breakdown of the results of running unit tests.

Result	Occurrences	% of Suggestions
Syntax Error	2057	10.2
Compilation Error	9103	45.3
Failed Unit Tests	2288	11.4
Passed Unit Tests	6652	33.1
Total	20100 = 201 * 100	100%

fail to compile or are found incorrect based on the unit test cases provided by the developer. Table 4 shows a breakdown of how many suggestions fail at this stage. As we can see, at the end of this step we are left with ~33% of the suggestions we started with. Next, we analyze some reasons for compilation errors and unit test failures.

4.2.4 ***Quality Of Suggestions (RQ3).*** Table 5 shows the main causes of compilation errors. After grouping together similar compilation errors, we found that they fell into 4 major error categories: *Undefined Identifier*, *Incorrect Argument passing*, *Incorrect Using Statements* and *Incorrect Return Type*. Upon looking at some instances of each category, we identified patterns of mistakes in the model's suggestions that cause the build to fail with these errors.

We noticed that the *Undefined Identifier* errors tend to happen when the model tries to use methods or classes outside of the provided context. As the model can only guess what other classes are in the project and the methods contained within, it often makes calls to methods that do not exist. We believe this could be improved by adding information regarding other classes within the project, such as the ones that may be used in the input code or from imported namespaces, along with the input context.

The *Incorrect Argument* errors also tend to occur when the model calls a method outside of provided the context. This results in the model passing in the wrong arguments types or number of arguments by making calls to method overloads that don't exist. We often saw this occur when the model tried to call member methods within some project-specific classes that were instansiated somewhere in the input code.

Cases for the *Incorrect Using Statements* follow a similar pattern as well. Here the model tries to import namespaces within the repo that don't exist or from packages that aren't in the build files. Since it doesn't know what other files exist in the project or the packages included in build, it often adds incorrect import statements.

The fourth category, *Type Mismatch*, is seen when the model suggests modifications that change the types of one or more class attributes, which get used elsewhere in the class. Since it can only modify the methods that are included in the input context (due to limited window), it is unable to modify these other methods. Other reasons include mismatch caused by changing the return type of a method when the input class implements an interface, since changing the type would cause the method in the parent to not be overridden, leading to a compiler error.

While we didn't exhaustively go through every single build error, according to these observations, we believe a significant portion of above errors could be resolved by including a larger context such as more methods in the input class or even other classes/files in the project, through extended context. We leave this exploration to future work.

Table 5: Main reasons for compilation errors.

Error Cause	Error Codes	Occurrences	% of Errors
Undefined Identifier	CS1061, CS0117, CS0246, CS0103, CS1579, etc.	4619	50.7
Incorrect Arguments	CS1503, CS1501, CS1729, CS7036, CS0305, CS0029, CS0019, etc.	3060	33.6
Incorrect Using Statements	CS0234	557	6.1
Type Mismatch	CS0266, CS0738, CS0508, etc.	161	1.8
Other Mistakes	CS0021,CS0122, (~125 misc. codes)	706	7.8
Total		9103	100.0

4.2.5 Running Benchmark Tests. The next step is to run benchmark tests for each of the changes that pass unit testing stage. However, before we run the benchmark tests we had to make some changes to the provided benchmark test suite to ensure the tests track the right metrics and that results are comparable among separate runs. By default, BenchmarkDotNet tests do not track allocations. For 22 out of the repos, we found that memory tracking wasn't enabled and we had to enable it ourselves by adding a `[MemoryDiagnoser]` attribute to the class containing the benchmarks. Changing this does not affect the results for other metrics tracked by the benchmarks like test durations. Another change we had to make to make the numbers comparable between separate runs was to add seeds to instances of random number generators instantiated in the benchmarking code. This is to ensure that the tests are deterministic so that the results can be compared between separate runs of the tests.

Additionally, to ensure no interference from background workloads, we run the benchmark test in a sterile work environment with minimal workload other than the test itself. We first run the benchmark tests without any changes to measure the baseline performance of the application and then once after applying each of the changes.

4.2.6 Comparing Results. Allocations are expected to stay consistent for C# applications as long as the benchmark tests are deterministic, so it is easy to tell if the change has improved memory usage by comparing the "Allocated" column (as shown in Figure 4). We consider a change to be a performance improvement in terms of Memory if it reduces allocations compared to the baseline. For test duration, we make use of the provided metrics in the test summary, namely the "Q1" and "Q3" columns, which represent the first and third quartiles of the sample, respectively. We consider a change to be a performance improvement in terms of CPU, if the suggestion's upper Tukey fence is found to be lower than the the baseline's lower Tukey fence i.e. if ${Q_3}_{suggestion} + 1.5(IQR_{suggestion}) < ({Q_1}_{baseline} - 1.5(IQR_{baseline})$, where IQR is the interquartile range, $Q_3 - Q_1$. Since there may be noise from background processes, we picked this criteria to be robust to outliers and have fewer false positives.

4.2.7 ***Ability To Suggest Real Perf Improvements & Usefulness (RQ4).*** Upon comparing the results against the baseline, we found that 543 suggestions improve performance metrics (~10% of the suggestions that pass unit tests). These changes were saturated within 39 of the 201 methods. We verify each of these suggestions with a performance expert and submit a PR containing the change with the biggest performance improvement for a each method. In case a project has correct suggestions for multiple methods, we squash all the changes into a single PR.

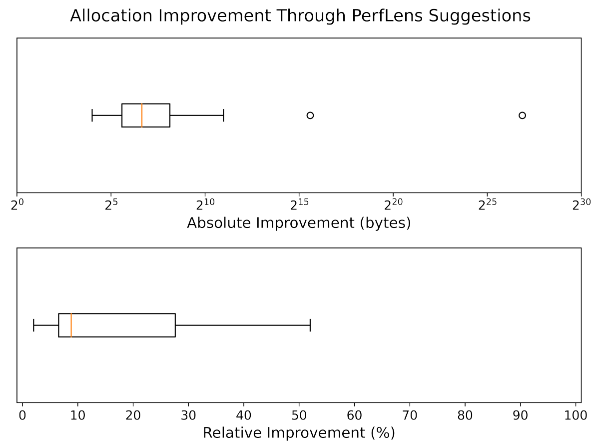

Figure 7: Above boxplots showing improvement in benchmark test allocations over baseline due to DeepPERF suggestions. Top plot shows the absolute improvement in terms of bytes and the one below shows relative improvement over the test's baseline allocations.

Figure 7 shows the improvement in allocations due to DeepPERF's suggestions, when compared to test's baselines allocations. Looking at the relative improvement, we see that DeepPERF's suggestions typically provide ~10% improvement in allocations. A few of our suggetions provide improvement on the order of KBs or even MBs. While others on the lower end do seem to veer in the territory of micro-optimizations, remember that this is from benchmark tests and the user may have simply written their benchmark test to be small. We also don't know how often the tested code is run when the application sees use by a real customer. Depending on how often the code being tested is exercised during the application's runtime such as if it appears on the application's hot-path, our suggestions could improve performance significantly.

RQ4: In summary, we found that for 26 out of the 39 methods, DeepPERF had at least one correct performance improvement suggestion. We've submitted a total of 17 PRs, 6 of which have already been approved by the project owners demonstrating the usefulness of our suggestions.

4.2.8 ***False Positives (RQ5).*** One of our PRs was closed because the repo was not open to external contributions, but the developer did not comment as to whether they considered the changes to be incorrect. Our remaining 10 PRs are still "Open" waiting for a response from the project owner. 13 out of the 39 methods were false positives i.e. they only had incorrect suggestions that seemed to improve benchmark results and somehow managed passed unit tests. This is a known issue in such models as they often generate suggestions that are test suite adequate, but are otherwise incorrect. While we make sure the methods we test have a high code coverage, that doesn't guarantee that the unit test will detect all mistakes as it may not be written to specifically test this particular method being modified. Another reason could be that the test suite itself is weak. One way to combat these cases would be to generate additional unit tests or benchmark tests and use them as further validation. One could also train an additional classifier to determine whether a change is correct and use it for filtration. We leave these explorations for future work.

For the cases where the model had generated a correct suggestion, it was usually able to suggest the correct patch within the first or the second suggestion. Often times it suggested multiple distinct correct patches that seemed to improve performance. In these cases, we submitted the correct patch with the higher performance improvement. Figure 1 shows two examples of such patches suggested during this evaluation that have been approved by the project owners.

5 THREATS TO VALIDITY

6 RELATED WORK

We describe the prior work and explain how our work differs from prior tools developed for performance bug detection in particular, bug detection in general.

6.1 Detecting Performance Bugs

There is a rich history of building tools for detecting performance bugs and improving performance. The majority of these tools identify code locations that take a long time to execute. Several tools generate or select tests for performance testing [38, 11, 5]. Other performance detection tools focus on detecting a specific type of performance bug. For instance, a set of tools have been developed for detecting runtime bloat [34, 36, 10], low-utility data structures [35], database related performance anti-patterns [6], false sharing problem in multi-threaded software [17], and detecting inefficient loops [21, 33]. Our tool extends the prior work on performance bug detection by developing a system that focuses on diagnosing general performance problems and considers both performance symptoms as well as source code features.

6.2 Automatic Bug Detection

Rule-based detection and static analyzers have been widely adopted for detecting software bugs [37, 32, 27, 8]. Beyond these traditional rule-based tools, there has been significant recent work on the usage of data-driven approaches as well as machine learning for software bug and vulnerability detection. For instance, Russell et al. [26] propose a machine learning based method for detection software vulnerabilities in C/C++ code bases. Similarly, Pang et al. [22] trained a machine learning model to predict static analyzer labels for Java source code. Li et al. [16] trained a recurrent neural network to detect two specific type of vulnerabilities related to improper use of library/API functions. Allamanis et al. [2] presents a comprehensive review of prior work on learning from "big code".

There are also several existing tools for automated repair, which both find and attempt to fix potential bugs. For example, Le Goues et al [14] present GenProg, which leverages genetic algorithms to generate repair candidates for a given error. Similarly, Long et al [18] developed Prophet that generates repair candidates based on a probabilistic model of potentially buggy code. Gupta et. al. [12] introduce DeepFix, which applies deep learning to generate fixes for simple syntax errors. Phoenix is a fully-automated pipeline that learns generalized executable repair strategies from patches for static analysis violations and applies the leaned repair to fix unseen violations [3]. We are uniquely contributing to this area of research by presenting a data-driven approach for detecting and suggesting fixes for performance bugs.

7 CONCLUSIONS

Performance bugs can cause significant performance degradation and resource waste. Detecting and diagnosing performance bugs are both important and challenging. Our work makes three contributes to improve the state of the art of detecting and diagnosing performance bugs. First, we presented a novel deep learning based approach to automatically generate patches for performance improvement. Second we collect a dataset of performance problems to conduct a static evaluation of our approach. This data set can inform future studies and applications in this space. Finally, we demonstrate a highly practical, end to end pipeline featuring this model along with unit testing and benchmarking to provide tangible performance improvements to real world projects. Further, the fact that several of our PRs have been merged shows that our changes were found convincing and valuable by the project owners.

8 ACKNOWLEDGEMENTS

REFERENCES

[1] Miltiadis Allamanis. "The adverse effects of code duplication in machine learning models of code". In: Oct. 2019, pp. 143–153. ISBN: 978-1-4503-6995-4. DOI: 10.1145/3359591.3359735.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–37.

[3] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. "Phoenix: Automated data-driven synthesis of repairs for static analysis violations". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2019, pp. 613–624.

[4] Pavol Bielik, Veselin Raychev, and Martin Vechev. "Learning a static analyzer from data". In: *International Conference on Computer Aided Verification.* Springer. 2017, pp. 233–253.

[5] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. "WISE: Automated Test Generation for Worst-Case Complexity". In: *Proceedings of the 31st International Conference on Software Engineering.* ICSE '09. USA: IEEE Computer Society, 2009, pp. 463–473. ISBN: 9781424434534. DOI: 10.1109/ICSE.2009.5070545. URL: https://doi.org/10.1109/ICSE.2009.5070545.

[6] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. "Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping". In: *Proceedings of the 36th International Conference on Software Engineering.* ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1001–1012. ISBN: 9781450327565.

[7] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. "PyMT5: Multi-mode Translation of Natural Language and Python Code with Transformers". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP).* 2020, pp. 9052–9065.

[8] *Coverity.* URL: https://scan.coverity.com/.

[9] Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. "DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons". In: *ArXiv* abs/2105.09352 (2021).

[10] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. "A Scalable Technique for Characterizing the Usage of Temporaries in Framework-Intensive Java Applications". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* SIGSOFT '08/FSE-16. Atlanta, Georgia: Association for Computing Machinery, 2008, pp. 59–70. ISBN: 9781595939951. DOI: 10.1145/1453101.1453111. URL: https://doi.org/10.1145/1453101.1453111.

[11] Mark Grechanik, Chen Fu, and Qing Xie. "Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing". In: *Proceedings of the 34th International Conference on Software Engineering.* ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 156–166. ISBN: 9781467310673.

[12] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. "Deepfix: Fixing common c language errors by deep learning". In: *Proceedings of the aaai conference on artificial intelligence.* Vol. 31. 1. 2017.

[13] Paul Kallender. *Trend Micro will pay for PC repair costs.* 2005.

[14] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. "Genprog: A generic method for automatic software repair". In: *Ieee transactions on software engineering* 38.1 (2011), pp. 54–72.

[15] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* 2020, pp. 7871–7880.

[16] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. "Vuldeepecker: A deep learning-based system for vulnerability detection". In: *arXiv preprint arXiv:1801.01681* (2018).

[17] Tongping Liu and Emery D. Berger. "SHERIFF: Precise Detection and Automatic Mitigation of False Sharing". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications.* OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 3–18. ISBN: 9781450309400. DOI: 10.1145/2048066.2048070. URL: https://doi.org/10.1145/2048066.2048070.

[18] Fan Long and Martin Rinard. "Automatic patch generation by learning correct code". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 2016, pp. 298–312.

[19] Domas Mituzas. "Embarrassment". In: *Blog post: http://dom. as/2009/06/26/embarrassment* (2009).

[20] Glen Emerson Morris. "Lessons from the Colorado benefits management system disaster". In: *Advertising and Marketing Review* (2004).

[21] Adrian Nistor, Tian Jiang, and Lin Tan. "Discovering, Reporting, and Fixing Performance Bugs". In: *Proceedings of the 10th Working Conference on Mining Software Repositories.* MSR '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 237–246. ISBN: 9781467329361.

[22] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. "Predicting vulnerable software components through n-gram analysis and statistical feature selection". In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA).* IEEE. 2015, pp. 543–548.

[23] Paolo Pialorsi and Marco Russo. *Introducing microsoft® linq.* Microsoft Press, 2007.

[24] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis". In: *ArXiv* abs/2009.10297 (2020).

[25] T. Richardson. *1901 census site still down after six months.* 2002. URL: http://www.theregister.co.uk/2002/07/03/1901%20census%20site%20still%20down/.

[26] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. "Automated vulnerability detection in source code using deep representation learning". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA).* IEEE. 2018, pp. 757–762.

[27] *SonarQube.* URL: https://www.sonarqube.org/.

[28] Linhai Song and Shan Lu. "Performance Diagnosis for Inefficient Loops". In: May 2017, pp. 370–380. DOI: 10.1109/ICSE.2017.41.

[29] Linhai Song and Shan Lu. "Statistical debugging for real-world performance problems". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications.* 2014, pp. 561–578.

[30] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. "IntelliCode Compose: Code Generation Using Transformer". In: *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20).* May 2020. URL: https://www.microsoft.com/en-us/research/publication/intellicode-compose-code-generation-using-transformer/.

[31] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. "Unit Test Case Generation with Transformers". In: *ArXiv* abs/2009.05617 (2020).

[32] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. "ITS4: A static vulnerability scanner for C and C++ code". In: *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00).* IEEE. 2000, pp. 257–267.

[33] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. "Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis.* ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery,

2013, pp. 90–100. ISBN: 9781450321594. DOI: 10.1145/2483760.2483784. URL: https://doi.org/10.1145/2483760.2483784.

[34] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. "Go with the Flow: Profiling Copies to Find Runtime Bloat". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 419–430. ISBN: 9781605583921. DOI: 10.1145/1542476.1542523. URL: https://doi.org/10.1145/1542476.1542523.

[35] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. "Finding Low-Utility Data Structures". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: Association for Computing Machinery, 2010, pp. 174–186. ISBN: 9781450300193. DOI: 10.1145/1806596.1806617. URL: https://doi.org/10.1145/1806596.1806617.

[36] Guoqing Xu and Atanas Rountev. "Detecting Inefficiently-Used Containers to Avoid Bloat". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: Association for Computing Machinery, 2010, pp. 160–173. ISBN: 9781450300193. DOI: 10.1145/1806596.1806616. URL: https://doi.org/10.1145/1806596.1806616.

[37] Zhongxing Xu, Ted Kremenek, and Jian Zhang. "A memory model for static analysis of C programs". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2010, pp. 535–548.

[38] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. "Automatic Generation of Load Tests". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. USA: IEEE Computer Society, 2011, pp. 43–52. ISBN: 9781457716386. DOI: 10.1109/ASE.2011.6100093. URL: https://doi.org/10.1109/ASE.2011.6100093.