InferFix: End-to-End Program Repair with LLMs

Matthew Jin
Microsoft
Redmond, WA, USA

Syed Shahriar
UCLA
Los Angeles, CA, USA

Michele Tufano
Microsoft
Redmond, WA, USA

Xin Shi
Microsoft
Redmond, WA, USA

Shuai Lu
Microsoft Research
Beijing, China

Neel Sundaresan
Microsoft
Redmond, WA, USA

Alexey Svyatkovskiy
Microsoft
Redmond, WA, USA

ABSTRACT

Software development life cycle is profoundly influenced by bugs; their introduction, identification, and eventual resolution account for a significant portion of software development cost. This has motivated software engineering researchers and practitioners to propose different approaches for automating the identification and repair of software defects.

Large Language Models (LLMs) have been adapted to the program repair task through few-shot demonstration learning and instruction prompting, treating this as an infilling task. However, these models have only focused on learning general bug-fixing patterns for uncategorized bugs mined from public repositories. In this paper, we propose InferFix: a transformer-based program repair framework paired with a state-of-the-art static analyzer to fix critical security and performance bugs. InferFix combines a Retriever – transformer encoder model pretrained via contrastive learning objective, which aims at searching for semantically equivalent bugs and corresponding fixes; and a Generator – an LLM (12 billion parameter Codex Cushman model) finetuned on supervised bug-fix data with prompts augmented via adding bug type annotations and semantically similar fixes retrieved from an external non-parametric memory.

To train and evaluate our approach, we curated InferredBugs, a novel, metadata-rich dataset of bugs extracted by executing the Infer static analyzer on the change histories of thousands of Java and C# repositories. Our evaluation demonstrates that InferFix outperforms strong LLM baselines, with a top-1 accuracy of 65.6% for generating fixes in C# and 76.8% in Java. We discuss the deployment of InferFix alongside Infer at Microsoft which offers an end-to-end solution for detection, classification, and localization of bugs, as well as fixing and validation of candidate patches, integrated in the continuous integration (CI) pipeline to automate the software development workflow.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Programming by example**; **Automated static analysis**.

KEYWORDS

Program repair, static analyses, prompt augmentation, finetuning

ACM Reference Format:
Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3611643.3613892

1 INTRODUCTION

The software development lifecycle is profoundly affected by bugs. Traditional program analyses techniques can detect and localize bugs through formal reasoning, leaving the task of categorizing the bugs and coding a patch to a developer. The traditional approach of manually generating patches through examination of code changes is a time consuming and error-prone task, which could be automated.

Many of the recently proposed approaches for bug prediction, detection, and repair rely on machine learning algorithms – infamously *data hungry* – depending on large amounts of high quality data for effective training. LLMs have been successfully adapted to the program repair tasks through few-shot demonstration learning and instruction prompting, treating this as an infilling task [17]. However, while focusing on solving specific research problems they failed to provide a reliable end-to-end program repair solution that could be productized.

Static analysis tools like Infer can be used to identify critical security and performance issues. This can preempt large parts of the software development cycle, including the process of creating detailed unit tests, which can be extremely time-consuming and difficult for a large, complex project whose code is broken down into many modules or across many files [35]. They can also identify bugs and produce bug reports in a way that is machine-readable and conducive to usage in conjunction with patch generation models.

In this work we focus on three types of bugs reported by Infer: Null Pointer Dereference (NPD), Resource Leak (RL), and Thread

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3613892

Safety Violation (TSV). We focus on these because they pose critical performance, reliability and security issues, and can also be more difficult to fix than other issue types which are also more commonly detected and studied.

Language models commonly adopt two paradigms for task-specific generalization – via finetuning or few-shot learning [3, 11, 34]. In the former paradigm, the canonical model training structure is divided into two phases – pretraining and finetuning. During pretraining, a model is trained in a self-supervised way to perform denoising or generic sequence-to-sequence transformations, geared towards improving the performance on a variety of downstream tasks. In the finetuning stage the model is trained on a specialized supervised dataset to perform a concrete task, such as question answering, text summarization, or in our case, program repair. The few-shot learning paradigm allows model specialization for a downstream task via prompt augmentation, composition, or ensembling [24]. A variant of prompt augmentation, commonly called *demonstration learning*, introduces a few input-output examples for a given task, for instance "The capital of China is Beijing. The capital of Italy is Rome. The capital of South Africa is [X]", allowing to achieve good performance on a downstream task without any gradient updates, which is crucial for very LLMs like GPT-3 [4], T5 [33], and PaLM [8]. Another variant of prompt augmentation commonly referred to as *instruction prompting* aims to introduce a natural language description of a task, for instance: "write a program to determine whether a graph is bipartite". It may use prompt templates for filling in necessary information from an external source (a database, a neural model). In our approach we combine the benefits of both paradigms, by augmenting the prompts and then finetuning our model on the dataset of augmented prompts and predictions to get the best performance.

In this paper, we introduce InferFix– a program repair framework which combines a transformer encoder model pretrained via contrastive learning serving as a retriever over a database of historic bugs and fixes, and an LLM (12 billion parameter Codex Cushman model, `code-cushman-001`) instrumented with the facility to leverage retrieved information from the external database. Given the baseline Codex model has been shown to occasionally predict insecure or buggy code [31], we prioritized finetuning it on a bug-free supervised dataset of bugs and fixes with contexts enriched via relevant program repair patterns from an external non-parametric memory. The contributions of the paper are as follows: (i) we propose a program repair framework that leverages static analyses for bug detection, localization, and categorization paired with a LLM finetuned for program repair task on a dataset of augmented prompts, (ii) we curate InferredBugs: a metadata-rich dataset of bugs and fixes in Java and C# programming languages extracted with the Infer static analyzer, (iii) we introduce a dedicated prompt augmentation technique for program repair task, which leverages dense retrieval from an external database of historic bugs and fixes, bug type annotations, and syntactic hierarchies across the entire source code file affected by a bug, (iv) we evaluate our model on the InferredBugs dataset, achieving an impressive 76% top-1 accuracy of patch generation in Java, and over 65% in C#, across null pointer dereference, resource leak, and thread safety violation bug types, and finally (v) we deploy InferFix as a GitHub action and as part of the Azure DevOps CI pipeline internally at Microsoft, and document aspects of deployment.

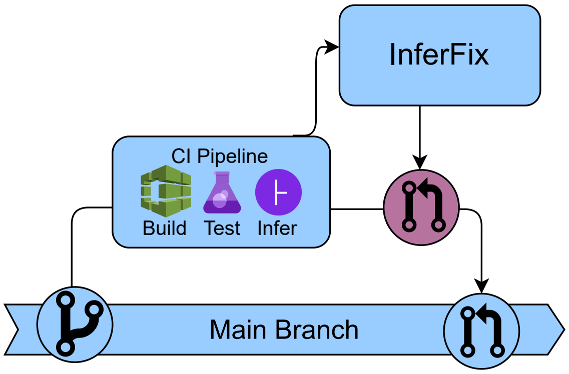


Figure 1: Software development workflow automated with InferFix. A developer creates a PR to check in code changes implemented in a feature branch, if Infer static analyzer detects a bug, a relevant code context is then prepared and InferFix generates a patch which is served as a bug-fixing PR into feature branch.

2 MOTIVATING EXAMPLE

To provide the intuition about how our approach works and to describe the concrete details of the bug detection, localization, and repair scenario we begin with a motivating example.

In a typical continuous software development workflow, software engineers make atomic, iterative changes to feature branches periodically merging to the `main` production branch, which is then continuously and automatically deployed to the end users. Consider a large software project with a modular code base spread across source code files. It can be inefficient for developers to detect, localize, and fix errors manually before they are merged to main. In addition, it requires the creation of an extensive unit test suite to ensure that a feature or change works across all possible versions of the software, and no regressions are introduced.

Figure 1 illustrates a typical software development workflow at Microsoft Developer Division in the presence of InferFix. As a pull request (PR) proposing code changes is created, CI pipeline triggers unit testing, build, and Infer static analysis steps. If bugs are detected, the InferFixpatch generation module will be invoked to propose a fix. The proposed bug fix is then validated and subsequently served as a bug-fixing PR to a feature branch allowing developer to catch bugs before merging the code to the production branch.

Our approach combines a static analyzer to detect, localize, and classify bugs with a powerful LLM (finetuned 12 billion parameter Codex model) to generate fixes.

Figure 2 provides details about InferFix workflow based on a real-world bug example from the acs-aem-common [1] repository, which is a unified collection of code for content management that optimizes authoring, and delivery of content and digital media written in Java. The Infer static analyzer detects a null pointer dereference error, due to an object in the code returned by `getResourceResolver` (this,adaptable) call, which could be null and is dereferenced at line 168. The context preprocessing module uses the information provided by the analyzer to extract the buggy method, and retains surrounding context most relevant to fixing the bug – import statements, class signature, body of the getResourceResolver method which is invoked at buggy line. The retrieval augmentation engine then searches for semantically similar buggy code snippets in the historic database, prepending similar bug-fixes to the prompt. Finally, the augmented prompt is sent to the finetuned Codex model for inference. The predicted patch is then validated by executing the Infer static analyzer and unit tests as part of the CI pipeline to ensure the error is indeed fixed and no regressions are introduced in the code base.

3 DATASET

We collect a supervised dataset of bugs detected with Infer (Infer Static Analyzer), which performs semantic analysis via Separation Logic.

We executed Infer and InferSharp over the change histories of approximately 6.2k Java and C# open-source repositories (2.9k Java, 3.3k C#) hosted on GitHub, analyzing more than 1 million commits. While a few bug datasets are already available, such as Defects4j [18], QuixBugs [22], ManySStuBs4J [19], UnifiedBug-Dataset [13] and many others, the dataset we introduce is differentiated by the amount and quality of information provided about each bug by the static analysis. Specifically, each bug in the dataset is associated with several pieces of metadata, including:

- *Bug Type*: each detected or fixed bug is marked with a bug type extracted with Infer, such as: null dereference, resource leak, immutable cast, etc. This information could be potentially used by automated program repair techniques to guide the bug-fixing attempts. Alternatively, these instances can be used as labeled data points for bug classification techniques.
- *Bug Location*: the dataset provides localization info at different levels of granularity: file, class, method, and line. For specific types of bugs, also affected variables/methods are reported.
- *Change History*: bugs are linked with the change history of the software project. Specifically, the dataset provides information on when a bug was introduced or fixed throughout the development process. Additionally, each analyzed commit is associated with the introduced/fixed or preexisting bugs involving the file touched in the commit.

3.1 Background on Infer Static Analyzer

Infer is an open-source static analysis tool originating from program analysis research on separation logic. It was first developed by the startup Monoidics Ltd, which was acquired by Facebook in 2013 and open-sourced in 2015. It computes program specifications to detect errors related to memory safety, concurrency, security, and more. It is industrially deployed at companies including Meta, Amazon, and Microsoft. Although in this work we will focus on null dereferences, resource leaks, and thread safety violations detected by Infer, it is able to detect a much wider variety of security and performance issues. For example, via taint tracking it is able to detect dataflow-related issues such as SQL injections. We believe our framework will be capable of mining and generating patches for these bug types as well, but leave the examination of this to future efforts.

At Meta, Infer runs within the internal CI system of repositories consisting of 10s and 100s millions of lines of code, including those for WhatsApp, Instagram, and Facebook core. Infer runs on diffs and reports issues to developers by writing comments within the code review system. A study conducted at Meta [27] saw a false positive rate under 20%, and issues posted saw a fix rate of 70%. The high issue relevance driven by this diff-time deployment of this system is critical; the same study saw a near-zero fix rate when it was deployed to developers as a list of assigned issues outside of the CI system. This underlines the value of our proposed system being deployed as a bug-detection-and-fix-recommendation code review module.

InferSharp [28] is the compiler frontend developed by Microsoft which translates the Common Intermediate Language (CIL) to the Smallfoot Intermediate Language interpreted by Infer, thereby enabling Infer's capabilities on all CIL languages (including C# and F#). For the purposes of this paper, InferSharp refers to the static analysis of Infer applied to CIL languages. Notably, to our knowledge it is the only interprocedural static analysis for CIL languages which is free-to-use and MIT-licensed. Considering Infer's industry track record, this creates unique opportunities in both research and industry to build bug-detection-and-fix product capabilities for a relatively underserved developer segment.

3.2 Collecting Data with Infer

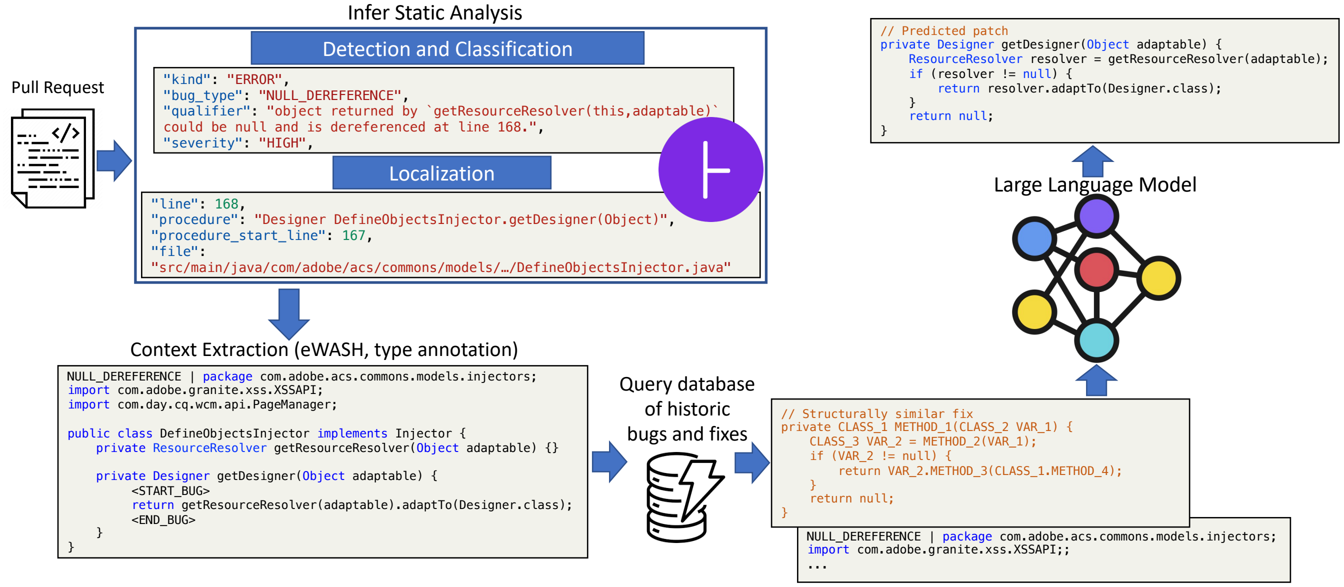
In this section we describe the data extraction process that culminated in the creation of the InferredBugs dataset. Specifically, we provide details on how we executed Infer over the change histories of software projects in order to detect introduced and fixed bugs.

Given as input the current commit *curr* and the previous commit *prev*, we begin by computing a git diff to identify the files involved in the change performed by the developer in the commit *curr*. Next, we analyze the status of the files at commit *prev*. Specifically, we checkout the snapshot of the system at commit *prev*, and we build the system using the project-specific build tool. During the build process, the infer capture command intercepts calls to the compiler to read source files and translates them into an intermediate representation which will allow Infer to analyze these files. Next, we invoke the infer analyze command specifying the files to be analyzed (*i.e.,* the files *diff* involved in the commit). This analysis produces a report *reportPrev* detailing the bugs identified within the specified files.

Subsequently, we move to the current commit *curr* and perform the same steps described for the commit *prev*, that is: checking out the commit, building system while capturing the source files, and analyzing the *diff* files in order to detect bugs.

Finally, with the infer reportdiff command, we compute the differences between the two infer reports *reportPrev* and *reportCurr*. The output *bugs* contain three categories of issues:

- *introduced*: issues found in *curr* but not in *prev*;
- *fixed*: issues found in *prev* but not in *curr*;
- *preexisting*: issues found in both *prev* and *curr*.


Infer Static Analysis
Pull Request
Detection and Classification
Localization
Context Extraction (eWASH, type annotation)
Query database of historic bugs and fixes
Large Language Model

```
"kind": "ERROR",
"bug_type": "NULL_DEREFERENCE",
"qualifier": "object returned by `getResourceResolver(this,adaptable)`
could be null and is dereferenced at line 168.",
"severity": "HIGH",
```

```
"line": 168,
"procedure": "Designer DefineObjectsInjector.getDesigner(Object)",
"procedure_start_line": 167,
"file":
"src/main/java/com/adobe/acs/commons/models/…/DefineObjectsInjector.java"
```

```
NULL_DEREFERENCE | package com.adobe.acs.commons.models.injectors;
import com.adobe.granite.xss.XSSAPI;
import com.day.cq.wcm.api.PageManager;

public class DefineObjectsInjector implements Injector {
    private ResourceResolver getResourceResolver(Object adaptable) {}

    private Designer getDesigner(Object adaptable) {
         <START_BUG>
         return getResourceResolver(adaptable).adaptTo(Designer.class);
         <END_BUG>
    }
}
```

```
// Structurally similar fix
private CLASS_1 METHOD_1(CLASS_2 VAR_1) {
    CLASS_3 VAR_2 = METHOD_2(VAR_1);
    if (VAR_2 != null) {
        return VAR_2.METHOD_3(CLASS_1.METHOD_4);
    }
    return null;
}
```

```
NULL_DEREFERENCE | package com.adobe.acs.commons.models.injectors;
import com.adobe.granite.xss.XSSAPI;;
...
```

```
// Predicted patch
private Designer getDesigner(Object adaptable) {
    ResourceResolver resolver = getResourceResolver(adaptable);
    if (resolver != null) {
        return resolver.adaptTo(Designer.class);
    }
    return null;
}
```

Figure 2: InferFix workflow. A buggy commit is detected by the Infer static analyzer, which is used to craft a prompt carrying the bug type annotation, location information, relevant syntax hierarchies (eWASH), and similar fixes retrieved from the historic database. A LLM – finetuned 12B Codex model – generates a patch.

We perform these steps for each pair of commits (*prev, curr*) over the change histories of the analyzed software projects. We optimize this process by obviating the need to build the same commit twice (*i.e.*, once as *curr* and next as *prev*) by instead reusing the build and capture stages in the next iteration.

3.3 Dataset Statistics

After running the extraction pipeline on 2937 repositories, we identified a total of 8280 bug patches. Of these bugs, 259 of these are null dereference patches which pass the filtering process, and 462 of these are resource leaks which pass the filtering process. We note that the filtered dataset contains commits which might have been detected by traditional methods involving extracting commits with certain keywords related to the desire bug type. Of the 259 null patches, 59 contain “null” or “npe” in the corresponding commit message, and of the 462 resource leak patches, 15 contain the “leak” keyword. We see from this that we are able to extract many additional fixes that would not have appeared using naive commit message keyword matching.

Table 1: Summary of the InferredBugs dataset in terms of the number of files and size of the bug-fixing two-way diff.

	NPD		RL		TSV	
	Java	*C#*	*Java*	*C#*	*Java*	*C#*
Num. bug patches	2686	1116	2382	1789	3582	40
Mean lines per patch	12.2	8.8	10.9	7.2	14.1	17.1
Mean char per patch	457.1	310.2	404.1	275.8	482.7	455.3

```
// Buggy code [snippet 1] // Fixed code [completion 1]
// Buggy code [snippet 2] // Fixed code [completion 2]
// Buggy code [snippet X]
```

Figure 3: Demonstration prompt design for program repair experiments with LLMs.

As shown in Table 1 the InferredBugs is composed of multi-line bugs, which represents a challenging case for program repair tools.

4 BASELINES

In the following, we explore several program repair baselines which are constructed around powerful LLMs (`code-cushman-001` and `text-davinci-003`) for tasks of completing code, filling code in the middle, or generating a fix following a natural language instruction. In the following, we evaluate the performance based on the accuracy of exact string match of a generated patch to the ground truth fix.

4.1 Demonstration Prompting

Demonstration learning is a prompt augmentation technique in which a few answered prompts are prepended to the context with the purpose of demonstrating how a language model should approach a downstream task. For program repair, we introduce a prefix constructed of two answered prompts as, followed by the actual buggy code snippet [X], as shown in Figure 3. Our few-shot demonstration learning experiments are based on the strong 12 billion parameter Codex language model of code.

```
Fix a [bug type string] in the following code snippet:
[snippet]
In your response, output code snippet only.
```

Figure 4: Instruction prompt design for program repair experiments with LLMs.

4.2 Conditional Language Modeling

Our next baseline is the zero-shot conditional language generation (code completion), which aims to use the next token prediction to repair programs. Specifically, given a bug-free prefix, we run Codex model inference to complete the buggy code snippet, aiming to rewrite a program without bugs. In our experiments, we apply nucleus sampling decoding algorithm with $top\_p = 1$ and a temperature $T = 0.7$ generating top 10 samples up to the length of 1024 tokens with a total length for prefix and completion of 2048. Our conditional language modeling experiments are also based on the `code-cushman-001`.

4.3 Instruction Prompting

Instruction learning is a prompt augmentation technique that introduces a natural language description of the task. To approach program repair, we prepare prompts following a template: We use the OpenaAI GPT-3 Davinci model, a 175 billion parameter language model and a close sibling of ChatGPT, to complete the prompts. Typically, Davinci outputs a natural language summary of the proposed fix followed by a code snippet. For the sake of evaluation, we instruct `text-davinci-003` to only output code snippet in its response which otherwise normally accompanied by the natural language descriptions.

5 INFERFIX FRAMEWORK

InferFix program repair framework is composed of three following key modules: (i) a static analysis tool that detects, localizes, and classifies bugs, (ii) retrieval module – a large index of historic bugs and fixes, equipped with a facility to efficiently search and retrieve "hints" – semantically-similar source code segments – given a query, and (iii) generator module – a LLM finetuned on a dataset of prompts enriched with the information provided by the static analyzer and the retriever to generate fixes.

5.1 Bug Detection & Classification Module

Our bug detection, localization, and classification module is powered by the Infer, which performs program analysis via Separation Logic. Although Infer's Pulse framework has recently been released, for the purposes of this paper we examine bugs generated by Infer's biabduction framework. Compiler frontends for Infer, such as InferSharp, translate source code into the control-flow-graph intermediate representation understood by Infer, known as the Smallfoot Intermediate Language. Infer performs automated program analysis over this graph and produces compositional method summaries in order to determine whether there are defects present in the source code.

5.2 Retrieval Module

Our retrieval module closely follows the ReACC formulation [25]. The retriever searches for semantically equivalent vulnerable code given a buggy code snippet and retrieves corresponding fix candidates based on cosine similarity between the embedding of query vector q and a buggy code snippet c.

Dense retrieval maps each code snippet to a d-dimension dense vector. The relevance of a code snippet to a given query can then be determined as a dot product of the query vector and each document vector. We closely follow the Dense Passage Retriever (DPR) model [20]. At the training stage, we adopt in-batch negatives to calculate the contrastive loss by InfoNCE [37].

Our dense retriever uses a bidirectional transformer encoder $\mathcal{E}$ to obtain encoded dense vector representations of the query ($\mathcal{E}(q)$), and for each buggy code snippet c indexed in the retrieval database ($\mathcal{E}(c)$). The retrieval database is a key-value store with encoded buggy code snippets $\mathcal{E}(c)$ serving as keys, and string representations of the corresponding fixes f serving as values.

We take the representation of the `[CLS]` token as a summary of the encoded sequences of tokens, and compute similarity between the query and each code snippet in the database as a dot-product: $sim(q, c) = \mathcal{E}(c)^T \cdot \mathcal{E}(q)$.

The bidirectional transformer encoder $\mathcal{E}$ is pretrained with the contrastive learning objective. Contrastive learning [39] is a self-supervised learning technique, in which the machine learning model is aiming to learn from the commonality of the training samples but also the attributes that make samples different. Given a contrastive pretraining dataset $D = \{q_i, p_i^+, p^-_{(i,1)}, ..., p^-_{(i,h)}\}$, $i = 0...N$, where each sample consists of a query – an encoding of a buggy code snippet; a positive sample representing a semantically similar code snippet of the same bug type; and a set of negative samples which are irrelevant code snippets of different bug types. The contrastive loss is then given by the following formula (negative log likelihood of the positive sample):

$$L(q_i, p_i^+, p^-_{(i,1)}, ..., p^-_{(i,n)}) = -\,log \frac{e^{sim(q_i, p_i^+)}}{e^{sim(q_i, p_i^+)} + \sum_{i=1}^{n} e^{sim(q_i, p^-_{(i,j)})}}, \tag{1}$$

where sim is the cosine similarity between the embedding vectors.

5.3 Generator Module

Our generator model is based on Codex Cushman, a 12B parameter decoder-only transformer language model [6] developed by OpenAI, which is a descendant of GPT-3, trained on source code.

We finetune Codex on a supervised corpus extracted from the InferredBugs dataset, with the goal of teaching the model to generate a fix for the given buggy code. Specifically, the input to the model is the buggy code augmented with additional information such as bug localization and categorization, hierarchical extended context, and retrieved similar fixes. We discuss the prompt augmentation process in detail in section 6.

We perform full model finetuning (updating all weights of the model), on sixty four 32 GB V100 GPU for 5 epochs, retaining best model checkpoint by the exact match accuracy metric. We use the Babel platform – a model repository and an AzureML designer component family bringing together state-of-the-art transformer models on Azure ML compute for rapid experimentation. We use Adam stochastic optimization procedure with the learning rate of 0.01, warmup period of 1000 optimization steps, and global batch size of 256. Particularly given the wide availability of LLMs such as GPT-3.5 at little or no cost, we believe our experimental results are reproducible with lesser access to high-end computing architecture, and with other LLMs.

6 PROMPT AUGMENTATION

Prompt augmentation has been shown to be a powerful technique for extracting high-quality outputs from LLMs, and, in particular, for domain and task adaptation. In the following we describe our dedicated prompt augmentation approach for program repair task. The proposed approach is two-fold: (i) we extract and prioritize syntax hierarchies which are most relevant to the buggy snippet region, including focal context, and (ii) retrieve hints – structurally similar bug fixes from commit histories on GitHub. By doing so we are constructing a loosely structured template which includes the following:

(1) Retrieved hints
(2) Bug type annotation
(3) Syntactic hierarchies and peer methods
(4) Focal methods
(5) Buggy method with location markers

Figure 5 shows an example of augmented prompt input for a null pointer dereference bug in Java, which includes the buggy code region surrounded by location markers, containing the method with surrounding most relevant syntax hierarchies, but type annotation string, and "hints" – structurally similar bug fixes retrieved from the historic database.

In the following subsections, we will describe each prompt augmentation technique and quantify its impact on bug-fixing performance by adding features incrementally.

6.1 Basic Prompt

The most basic prompt we can construct for the model is to provide the buggy method as input while expecting the model to generate the fix by outputting the fixed version of the given method. Thus, we perform task-oriented finetuning of our Generator model (Codex) using the buggy and fixed versions of the methods from the InferredBugs dataset.

We compare this basic prompting and finetuning against powerful LLM baselines described in Sec. 4. Table 2 illustrates the effect of finetuning as compared to zero-shot and few-shot variants. Demonstration learning appears to be the most successful few-shot learning strategy for adapting the Codex model (`code-cushman-001`) to downstream patch generation task, yielding a modest 19–25% accuracy of fixing Java bugs. Instruction learning, which also includes natural language descriptions of the downstream task in the prompt, only becomes viable as the model size increases – we repeated the instruction learning experiments with the 175 billion parameter Davinci model (`text-davinci-003`), a close sibling of ChatGPT. We observe a very competitive performance with the Davinci variant, with 40–53% accuracy of fixing Java bugs via prompt augmentation alone. Task oriented finetuning, without any prompt crafting, outperforms all few-shot baselines by a good margin, showing 11–55% relative improvement of accuracy across all bug types in Java. This improvement comes at a cost of computing resources necessary to finetune the Codex model, but provide an advantage of higher accuracy and cheaper inference as compared to few-shot Davinci.

Table 2: Top-1 accuracy for InferFix with basic prompt compared against LLM baselines

Approach	NPD		RL		TSV	
	Java	*C#*	*Java*	*C#*	*Java*	*C#*
Demonstration (Codex)	20.3	30.1	25.3	29.1	19.0	16.7
Completion (Codex)	6.7	6.1	7.8	5.7	3.9	0.0
Instruction (Davinci)	40.5	22.2	53.8	19.7	41.3	33.3
InferFix (basic prompt)	49.7	58.1	60.0	51.9	64.4	70.0

6.2 Bug Type Annotations

The simplest prompt augmentation step is to prepend a bug type annotation to a basic prompt consisting of a buggy method only. As shown in Table 3, this improves performance across all bug categories and languages (Java and C#) yielding 2.7–5.6% relative improvement in accuracy.

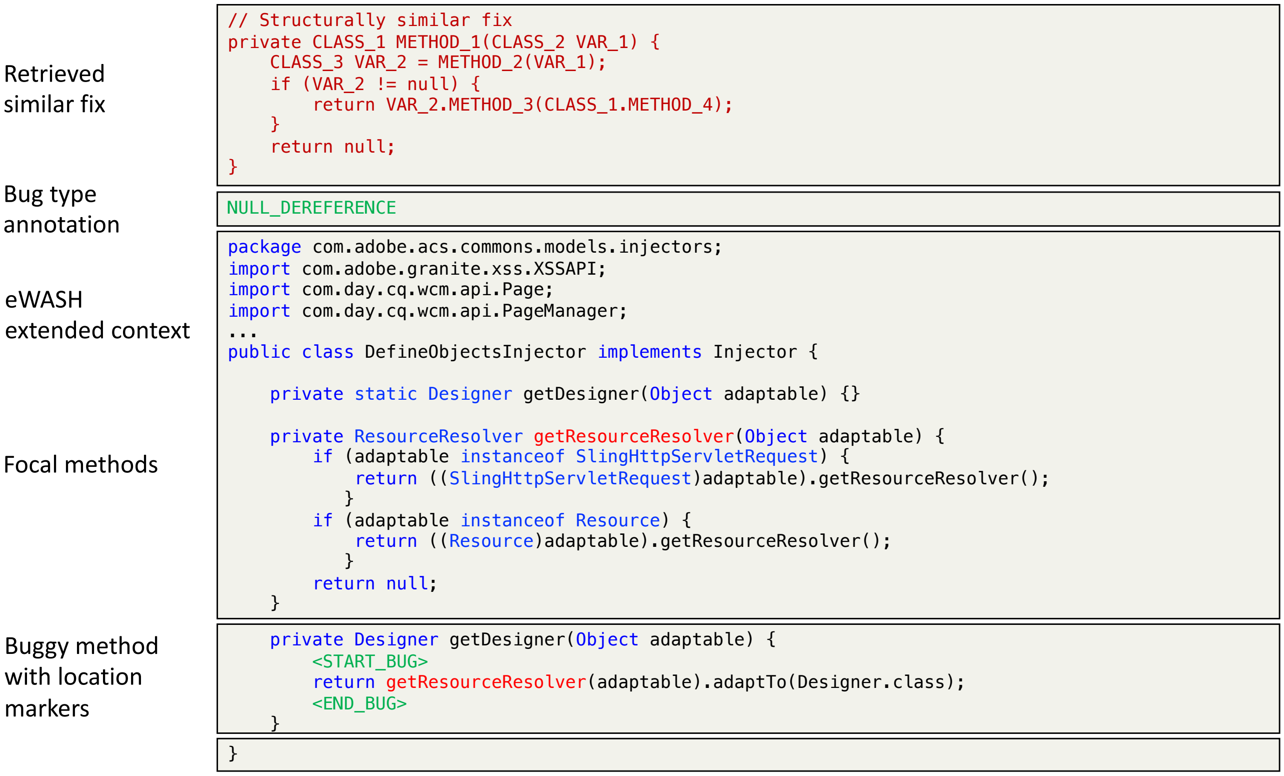
Table 3: Top-1 accuracy for InferFix demonstrating the impact of introducing the bug-type annotation in the prompt.

	NPD		RL		TSV	
	Java	*C#*	*Java*	*C#*	*Java*	*C#*
InferFix (basic prompt)	49.7	58.1	60.0	51.9	64.4	70.0
InferFix (+ bug type)	52.3	60.4	63.1	53.3	67.9	72.5

6.3 Bug Localization

Bug location information is crucial for accurate program repair. Infer static analyzer can localize bugs by tracking the flow of data through the program and detecting any violations of predefined rules or programming patterns. Infer static analyzer outputs a line number on which an error could occur at runtime, which, however, does not mean that the fix would require to edit this line only. In our dataset Table 1, the bugs are often spanning over multiple lines of code, having disjoint diff regions.

We use the bug location information output by Infer in two ways: (i) we parse the source code file affected by the bug to extract a method which contains the buggy line, and (ii) we surround the buggy region with special sentinel `<START_BUG>` and `<END_BUG>` symbols. During training, we refine the bug location by looking at the two-way diff markers with respect to the fix. During test time, we only use the information provided by the static analyzer as the fix is unknown. Table 4 demonstrates the impact of adding bug location markers in the prompt in addition to the bug type annotations. As seen, this leads to a positive effect across all categories studies, up to 3.4% relative improvement in accuracy. The effect is more pronounced for larger methods.



```
// Structurally similar fix
private CLASS_1 METHOD_1(CLASS_2 VAR_1) {
    CLASS_3 VAR_2 = METHOD_2(VAR_1);
    if (VAR_2 != null) {
        return VAR_2.METHOD_3(CLASS_1.METHOD_4);
    }
    return null;
}
```

```
NULL_DEREFERENCE
```

```
package com.adobe.acs.commons.models.injectors;
import com.adobe.granite.xss.XSSAPI;
import com.day.cq.wcm.api.Page;
import com.day.cq.wcm.api.PageManager;
...
public class DefineObjectsInjector implements Injector {

    private static Designer getDesigner(Object adaptable) {}

    private ResourceResolver getResourceResolver(Object adaptable) {
        if (adaptable instanceof SlingHttpServletRequest) {
            return ((SlingHttpServletRequest)adaptable).getResourceResolver();
        }
        if (adaptable instanceof Resource) {
            return ((Resource)adaptable).getResourceResolver();
        }
        return null;
    }
```

```
    private Designer getDesigner(Object adaptable) {
        <START_BUG>
        return getResourceResolver(adaptable).adaptTo(Designer.class);
        <END_BUG>
    }
```

```
}
```

Figure 5: Prompt augmentation for a method in Java programming language affected by a null pointer dereference bug.

Table 4: Top-1 accuracy for InferFix showing the effect of adding bug location markers in the prompt.

	NPD		RL		TSV	
	Java	*C#*	*Java*	*C#*	*Java*	*C#*
InferFix (bug type)	52.3	60.4	63.1	53.3	67.9	72.5
InferFix (+ localization)	53.5	61.4	64.4	53.9	69.6	75.0

6.4 eWASH extended context

A source code file may have nested scopes and references to other external libraries or other files. To accurately suggest patches, a model must use knowledge across different parts of the file. The length of source code files will often exceed the fixed-length window of transformer models (2048 tokens in our case), which could potentially lead to a loss of information relevant for learning to repair programs. To overcome this limitation, we use eWASH [9] to prioritize syntax hierarchies which are most relevant to the buggy snippet region. Extracting syntactic hierarchies from the entire source code files, as opposed to the tokens immediately preceding the bug location, we are able to retain most relevant code context, such as class-level fields and method arguments, and peer methods which are highly relevant to program repair. Starting with a concrete syntax tree of a source file, we organize and prioritize class-level and method-level syntactic elements such as global import statements and assigned values, class attributes, method signatures, class docstring, and global expressions in the input.

Quite often, a method affected by a bug will only contain an invocation expression or a call to a method defined elsewhere in the file – what we refer to as buggy *focal method*. For instance, in Figure 5 the buggy line of code has a return statement which is composed of a chain of method invocations with getResourceResolver and adaptTo focal methods. We conjecture that retaining the focal method implementation (signature, docstring, and body) in the prompt is crucial for program repair. We use stack traces provided as part of the Infer bug report to determine relevant focal method name, and include it in the prompt. Table 5 shows the effect of adding the eWASH syntax hierarchies and focal context in the prompt. As seen, patch generation accuracy is further improved by over 7.2–7.8% for Java and by 4.0–6.7% for C#.

Table 5: Top-1 accuracy for InferFix showing the effect of adding eWASH extended context in the prompt.

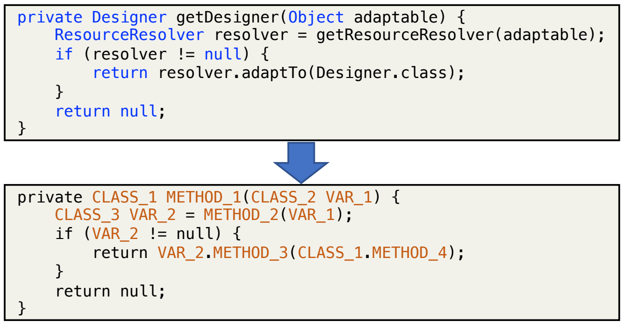
	NPD		RL		TSV	
	Java	*C#*	*Java*	*C#*	*Java*	*C#*
InferFix (localization)	53.5	61.4	64.4	53.9	69.6	75.0
InferFix (+ eWASH)	57.6	65.1	69.1	56.1	75.0	80.0

6.5 Enriching Context with Hints

To further enrich prompts, we perform a nearest neighbor search in the retrieval database for semantically similar fixes – so called hints. The resulting fixes are then prepended to the context with an instruction string `// Structurally similar fix`.

By default, we extract and prepend 2 nearest neighbors for each query. We apply quality criteria to avoid obviously incorrect matches: (i) retrieved fixes must be of the same bug type as the query, and (ii) impose a minimum similarity threshold between retrieved fixes and a query of 60%.

To focus on extracting structurally similar fixes and reduce the dependency on identifier naming we obfuscate code snippets serving as keys in the database and search queries. Namely, we parse and analyze the code identifier types and mask the names of classes, methods, and identifiers with placeholder symbols: `CLASS_NN`, `METHOD_NN`, and `VAR_NN`, where `NN` is a unique number. An example obfuscated representation is shown in Figure 6.



```
private Designer getDesigner(Object adaptable) {
    ResourceResolver resolver = getResourceResolver(adaptable);
    if (resolver != null) {
        return resolver.adaptTo(Designer.class);
    }
    return null;
}
```

```
private CLASS_1 METHOD_1(CLASS_2 VAR_1) {
    CLASS_3 VAR_2 = METHOD_2(VAR_1);
    if (VAR_2 != null) {
        return VAR_2.METHOD_3(CLASS_1.METHOD_4);
    }
    return null;
}
```

Figure 6: Code obfuscation example in Java.

Table 6 shows the improvements in bug-fixing capabilities for InferFix with prompt which incorporates retrieved hints. This prompt augmentation further improves InferFix performances by 1–2% in absolute top-1 performances.

Table 6: Top-1 accuracy for InferFix showing the effect of adding bug-fix hints.

	NPD		**RL**		**TSV**	
	Java	*C#*	*Java*	*C#*	*Java*	*C#*
InferFix (eWASH)	57.6	65.1	69.1	56.1	75.0	80.0
InferFix (+ retrieved hints)	59.5	66.7	71.2	57.0	77.4	82.5

6.6 Inference

The Inference step for InferFix involves using nucleus sampling decoding with a top_p parameter of 1.0 and a temperature of 0.7. This temperature value was selected as a higher temperature should generally be used when a larger number of predictions are generated. During this step, the tool decodes the top-10 best predictions generated by the LLM, and ranks them according to their sequence log probabilities. This ranking helps to ensure that the most likely and relevant fixes are presented to the user. The use of nucleus sampling decoding, with its specific top_p and temperature parameters, helps to balance the trade-off between diversity and quality in the generated predictions, making it possible to obtain highly accurate and diverse patch candidates.

7 RESULTS

Table 7 shows the results achieved by InferFix on the InferredBugs dataset compared against the LLM baselines discussed in section 4. InferFix is able to fix between 57% and 82% of the three categories of bugs for Java and C#, just with the top-1 prediction. The performance gap between our approach and the best performing baseline (Finetuned Codex) is between 8.6% and 13% in absolute terms.

It is important to keep in mind that the results shown in Table 7 present a conservative estimate of InferFix 's potential for generating fixes. The percentages displayed in the table are based on generated patches that exactly match the original developer's token-by-token fix. However, there may be other candidate patches that correctly fix the bug using a different token sequence.

The impressive top-1 results achieved by InferFix are critical for its efficient and effective integration into the software development cycle. With the ability to propose high-quality fixes for critical bugs, InferFix has the potential to greatly enhance the productivity and reliability of the software development process.

Table 7: Top-1 accuracy for InferFix on the InferredBugs dataset compared against LLM baselines

Approach	**NPD**		**RL**		**TSV**	
	Java	*C#*	*Java*	*C#*	*Java*	*C#*
Demonstration (Codex)	20.3	30.1	25.3	29.1	19.0	16.7
Completion (Codex)	6.7	6.1	7.8	5.7	3.9	0.0
Instruction (Davinci)	40.5	22.2	53.8	19.7	41.3	33.3
Finetuning (Codex)	49.7	58.1	60.0	51.9	64.4	70.0
InferFix	**59.5**	**66.7**	**71.2**	**57.0**	**77.4**	**82.5**

8 DEPLOYMENT

The deployment of InferFix at Microsoft as part of the Azure DevOps and GitHub CI pipeline has significantly improved the developer workflow for our internal projects at Developer Division. It has enabled our developer teams to automate the bug detection and fixing process, reducing the time and effort required to manually identify and fix bugs, and ensuring that bugs are addressed quickly and accurately. Figure 1 provides an overview of our CI pipeline with the integrated InferFix stages. When a PR is created, the CI pipeline automatically triggers three steps: (i) build, (ii) testing, and (iii) Infer (or InferSharp) static analysis. If bugs are detected, the InferFix patch generation module is invoked to propose a fix. InferFix leverages the detailed information about the bug provided by Infer, such as context, location, and classification of the bug type.

The InferFix module proposes a (configurable) set of candidate patches. Each candidate patch is packaged as a separate PR, which is individually validated. The validation process is seamless and reuses the three CI pipeline steps mentioned above. Specifically, the PR containing the candidate patch is validated through: (i) build – checking that the candidate patch is syntactically and semantically correct w.r.t. the source project; (ii) testing – ensuring that the candidate patch does not introduce regressions (failing tests); (iii)

Infer static analysis – validating that the candidate patch actually fixes the previously detected bug. The fix is then provided to the developer within the feature branch of the developer's PR. This process allows the discard of invalid fixes suggested by the model (*i.e.,* false positives), reducing the effort required by the developer to validate the candidates. The complexity of these stages is abstracted away from the developer, who will simply receive a PR comment within the system they are using (*e.g.,* GitHub or Azure DevOps). We implemented a GitHub action which receives a validated patch from InferFix and surfaces it to the developer in form of a comment in the PR. The comment provides detailed information about the bug (*i.e.,* extracted by Infer), and the resolution (*i.e.,* served by InferFix). The developer can accept or decline the recommended fix.

The deployment of InferFix in the CI pipeline for our internal projects has provided significant benefits. We are currently in the process of expanding its deployment; the benefits of this integration have been demonstrated through the seamless validation process and abstracted complexity for the developer.

9 RELATED WORK

Our approach is related to a broad set of literature on patch generation and task-oriented finetuning. We refer a reader to [29] for a more comprehensive overview on the prior research in the area of program repair, and [23] for a systematic survey of prompting methods in NLP.

Patches in the Wild [36], uses supervised machine translation to learn bug-fixing patterns for various common code defects. They mine bug-fixes from the change histories of projects hosted on GitHub and define the learning task on a method level. SequenceR [7] improves upon the Patches in the Wild by leveraging the extended context available through the source code file containing the buggy code, showing first attempt at prompt crafting. SequenceR learning objective is based around supervised machine translation with encoder-decoder recurrent neural network. Copy That! [30] builds upon an observation that patches typically only affect isolated spans of tokens. By introducing a span copying decoder they improve upon the previous state-of-the-art. While also using neural machine translation (NMT), DeepDebug [12] uses extensive self-supervised pretraining to improve upon the prior art. BugLab [2] takes a step towards self-supervised bug detection and repair, co-training two neural models: a detector model that learns to detect and repair bugs in code, and a selector model that learns to create buggy code for the detector to use for training. CODIT [5] uses a tree-based model to encode source code changes, learning bug-fixing activities. Recoder [40] generates edits in a syntax-guided manner and with a provider/decider architecture and placeholder generation. Lutellier *et al.* [26] employed ensemble learning with CNNs and NMT to generate patches with CoCoNuT. DLFix [21] is a two-tier model with the first layer focusing on learning the context of bug fixes and the second layer trying to generate the bug-fixing patch. Recently CURE [16] has reported state-of-the-art results on Defects4J and QuixBugs datasets, improving over NMT-based APR techniques with the use of a pre-trained programming language model, code-aware search, and sub-word tokenization.

These works are trained on unclassified bugs mined from open source repository change histories, and do not use bug type information during learning. Our approach takes advantage of Infer, leveraging it during the learning process to generate fixes tailored by bug category. We further differentiate ourselves by using LLMs, prompt augmentation, and task-oriented finetuning.

Our work also relates to research on pretraining strategies and prompt augmentation. [14] permutes ordering of the spans in the original prompt to train the model to infill. By randomly replacing spans of code with a sentinel token and moving them to the end of the sequence they yield a unified approach for both program synthesis (via left-to-right generation) and editing (via infilling). [32] introduces the LAMA dataset and provides manually curated cloze templates to probe knowledge in language models. [10] investigates a template-based method for exploiting the few-shot learning potential of generative pre-trained language models to sequence labeling. Specifically, they define templates such as "`<candidate_span>` is a `<entity_type>` entity", where `<entity_type>` can be "person" and "location", etc, and train a model using a filled template. [38] introduces chain of thought prompting, in which a task is broken down into a series of intermediate reasoning steps to improve LLM reasoning for complex tasks.

Our approach, InferFix, performs prompt augmentation by incorporating similar fixes identified in a historical database of bugs, along with other information. Leveraging similar fixes has also been explored in other approaches, such as SimFix [15], which forms a space by abstracting modifications from existing patches. It then analyzes similar snippets in the same program to extract concrete modifications, which forms a concrete space. The intersection of these two spaces is used to perform fine-grained code adaptation for patch generation. Differently from the AST-differencing approach proposed in SimFix, we rely on a dense retrieval model which allows for more flexibility in identifying similar code snippets with arbitrary length, not constrained by specific AST-subtrees. Furthermore, our approach enhances the prompt by providing additional information and cues to the LLM model to facilitate the repair process.

10 CONCLUSION

We introduced InferFix: an end-to-end program repair framework based on Codex and a state-of-the-art static analyzer designed to fix critical security and performance bugs in Java and C#. InferFix is based on a retrieval-based prompt augmentation technique and task-oriented finetuning that leverages bug-type annotations and extended source code context. We have also curated a InferredBugs, a novel, metadata-rich dataset of bugs extracted by executing the Infer and InferSharp static analyzers on the change histories of thousands of Java and C# repositories. Our experiments demonstrated that InferFix outperforms strong LLM baselines, reaching a top-1 accuracy of 65.6% for generating fixes in C# and 76.8% in Java on the InferredBugs dataset.

We deployed InferFix at Microsoft as a GitHub action and as an Azure DevOps plugin operating as part of the CI pipeline. This tool has significantly improved the software development workflow for our internal projects at Developer Division.

REFERENCES

[1] Adobe-Consulting-Services. 2023. acs-aem-common. https://github.com/Adobe-Consulting-Services/acs-aem-commons.

[2] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 27865–27876. https://proceedings.neurips.cc/paper/2021/file/ea96efc03b9a050d895110db8c4af057-Paper.pdf

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[5] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* (2020).

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

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

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. https://doi.org/10.48550/ARXIV.2204.02311

[9] Colin Clement, Shuai Lu, Xiaoyu Liu, Michele Tufano, Dawn Drain, Nan Duan, Neel Sundaresan, and Alexey Svyatkovskiy. 2021. Long-Range Modeling of Source Code Files with eWASH: Extended Window Access by Syntax Hierarchy. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 4713–4722. https://doi.org/10.18653/v1/2021.emnlp-main.387

[10] Leyang Cui, Yu Wu, Jian Liu, Sen Yang, and Yue Zhang. 2021. Template-Based Named Entity Recognition Using BART. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. Association for Computational Linguistics, Online, 1835–1845. https://doi.org/10.18653/v1/2021.findings-acl.161

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[12] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating Bug-Fixes Using Pretrained Transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3460945.3464951

[13] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. 2018. A public unified bug dataset for java. In *Proceedings of the 14th international conference on predictive models and data analytics in software engineering*. 12–21.

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. https://doi.org/10.48550/ARXIV.2204.05999

[15] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.

[16] 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.

[17] Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2022. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. https://doi.org/10.48550/ARXIV.2208.11640

[18] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[19] 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.

[20] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6769–6781. https://doi.org/10.18653/v1/2020.emnlp-main.550

[21] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[22] 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*. 55–56.

[23] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2022. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* (sep 2022). https://doi.org/10.1145/3560815 Just Accepted.

[24] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (jan 2023), 35 pages. https://doi.org/10.1145/3560815

[25] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 6227–6240. https://doi.org/10.18653/v1/2022.acl-long.431

[26] 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.

[27] Meta. 2023. Scaling Static Analyses at Facebook. https://cacm.acm.org/magazines/2019/8/238344-scaling-static-analyses-at-facebook/fulltext.

[28] Microsoft. 2023. InferSharp. https://github.com/microsoft/infersharp.

[29] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (jan 2018), 24 pages. https://doi.org/10.1145/3105906

[30] Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. 2021. Copy that! Editing Sequences by Copying Spans. In *AAAI*.

[31] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. https://doi.org/10.48550/ARXIV.2108.09293

[32] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language Models as Knowledge Bases?. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 2463–2473. https://doi.org/10.18653/v1/D19-1250

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[34] Sebastian Ruder, Matthew E. Peters, Swabha Swayamdipta, and Thomas Wolf. 2019. Transfer Learning in Natural Language Processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*. Association for Computational Linguistics, Minneapolis, Minnesota, 15–18. https://doi.org/10.18653/v1/N19-5004

[35] Ayse Bener Shirin Akbarinasaji, Bora Caglayan. 2018. Predicting bug-fixing time: A replication study using an open source software project. In *Journal of Systems and Software*. 173–186.

[36] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (sep 2019), 29 pages. https://doi.org/10.1145/3340544

[37] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation Learning with Contrastive Predictive Coding. *ArXiv* abs/1807.03748 (2018).

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/ARXIV.2201.11903

[39] Zhirong Wu, Yuanjun Xiong, Stella X. Yu, and Dahua Lin. 2018. Unsupervised Feature Learning via Non-parametric Instance Discrimination. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), 3733–3742.

[40] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.

Received 2023-05-18; accepted 2023-07-31