An Automated Resource Leak Fixer Powered by LLMs

Jonathan Carlson
jonathancarlson@ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

ABSTRACT

Resource leaks are subtle bugs that are hard for programmers to catch by hand and can cause crashes and security vulnerabilities. Although static analysis tools can be used to find them, it's hard to integrate their warnings into developer workflows. In this paper, we introduce Leak2LLM, a new resource leak fixing system using an LLM and any static analysis tool to fix resource leaks. We evaluate Leak2LLM on the resource leaks found by static analysis tools on the NJR dataset and fix 69.4% of them. This provides similar results to existing approaches while being more generalizable to larger methods and a wider variety of bugs.

1 INTRODUCTION

Normalized Java Resource (NJR) [17] is a dataset of Java programs. While Java is a garbage collected language, meaning programmers don't have to keep track of memory manually, it still allocates resources that should be closed after being used. Some examples of resources are files, database connections, or network sockets. Usually resource leaks aren't detected during testing since it's unlikely the operating system will run out of that resource. However, these resource leaks can be identified using static analysis tools such as Infer [4]. These tools are able to detect resource leaks by analyzing the source code without running it.

Once these resource leaks are detected, users need tool support to resolve these errors. Christakis and Bird's empirical study [5] found that one of the main pain points with static analysis tools was the lack of suggested fixes. Traditional approaches to resolving static analysis tools require the programmer to look through all of the possible errors and provide fixes. Since these tools can provide hundreds of errors, this is not scalable to large codebases.

Another issue is that static analysis tools may incorrectly flag some code as causing an issue or a repair tool may generate faulty code both of which act as false positives. Having false positives is another significant annoyance when using static analysis tools [5].

2 BACKGROUND

2.1 RLFixer

RLFixer [20] fixes resources leaks found by any resource leak detector. RLFixer only attempts to fix fixable leaks, which are ones that do not escape their enclosing method. For example, if a resource is in a field then it cannot be closed automatically. If the resource is a static field then it can't be closed because the resource should live for the lifetime of the program. If not all the code is accessible at compile time (e.g. a client's code when writing a library) then it's impossible to statically identify all uses of a field.

2.2 InferFix

InferFix [10] uses Infer to find bugs in Java and C# code then queries a historic database for similar bugs and finally passes the warnings to a finetuned LLM (12B Codex). InferFix's dataset is created by running Infer over approximately 6200 Java and C# repositories on GitHub and tracking the bug type, bug location, and change history. InferFix attempts to fix the following bug types: null dereferences, resource leaks, and thread safety violations. The change history is used to track in which commit a bug caught by Infer was introduced or fixed.

3 METHODS

The overall Leak2LLM workflow is shown in Figure 1. Leak2LLM extracts previously found resource leaks from RLFixer using JavaParser [9] then prompts a large language model to fix the leak. Leak2LLM then updates the original code with the flagged resource leak with the LLM provided code using JavaParser. It then runs the static analysis tool that discovered the original leak to see if the leak is now fixed. Only after the leak is fixed according to the tool, would a programmer need to review the fix suggested by the LLM. If the LLM fix fails to compile then the compilation error and failing code is resubmitted to the LLM and passed through the pipeline again. This was implemented as a series of batch processes over all of RLFixer's unique resource leaks.

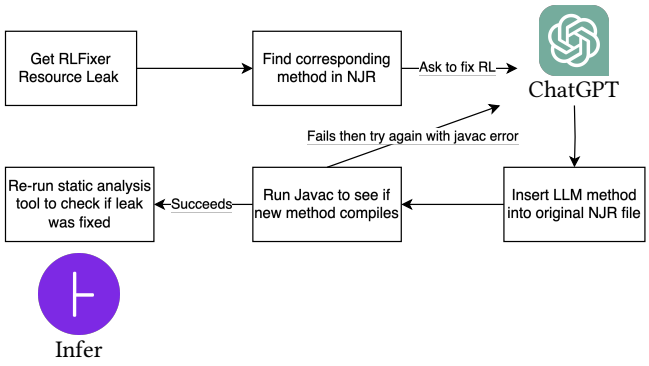

Figure 1: Overview of the Leak2LLM workflow

The RLFixer resource leak dataset is created by running the following static analysis tools on NJR:

- Infer
- Checker-framework (abbreviated to CF) [11]
- Spotbugs [2]
- PMD [1]
- CodeGuru [3]

We use mostly the same resource leaks as RLFixer, since we're both evaluating the approaches on NJR. Unlike RLFixer, we did

not calculate the fix rate for the CodeGuru tool as it cannot be run locally.

Since different tools can find the same resource leak in a program, these warnings need to be de-duplicated. However, the different tools output the warnings in different formats. While all of the tools output the program name, source file path, and line number; there can be inconsistencies in the line number. One tool may classify the leak as happening when the resource was instantiated and another might classify it where the resource was leaked. We use the same warning equality heuristics as RLFixer where we first compare the source file name and line numbers (need to be ± 2 apart), then if that doesn't match, we check if the method names are the same.

In Figure 2 you can see a resource leak found by Checkerframework, Infer, and PMD. We can see that it allocates the `File` resource on line 5 and wraps the `File` with `Scanner` on line 6 without closing the `Scanner` before returning from the function.

```
1  private static void insertQuoteAndMovie() throws Exception
2  {
3    String sql, quote, movie;
4    File fin = new File("MovieQuoteTrivia.txt");
5    Scanner sc = new Scanner(fin);
6    while (sc.hasNextLine()) {
7        quote = sc.nextLine();
8        movie = sc.nextLine();
9        sql = "INSERT INTO MovieTrivia (quote, answer)" +
10       "VALUES('" + quote + "','" + movie + "')";
11       s.executeUpdate(sql);
12       movies.add(movie);
13   }
14   System.out.println("Quote & Movie inserted");
15 }
```

Figure 2: Resource leak example from NJR

The method enclosing the resource leak is extracted based on the reported line number and source file using JavaParser.

As shown in Figure 3 we then pass to the LLM the following:

- the instruction that created the resource (`new Scanner`)
- all of the leaky method function (`insertQuoteAndMovie`)
- an example of how to close a resource using try-finally
- strict instructions on the output format

We add this retrieved information to the LLM prompt to provide additional information about the leak. This is similar to the idea of retrieval-augmented generation [12] which constructs better prompts using additional relevant information without requiring the model to be fine-tuned.

Similar to RLFixer, we use the try-finally approach to closing resources instead of try-with-resources [8] since try-with-resources can suppress exceptions changing the control flow of the program.

We used the GPT-4o [16] LLM from OpenAI which we accessed using their API [14]. In Figure 4 we can see GPT-4o initialized the `Scanner sc` variable to `null` because it needs to be accessible outside of the `try` block it wrapped the `new Scanner` instruction in. It then added a `finally` block that closes the `Scanner` resource if it was created. Re-running this new output through one of the static analysis tools that found it (e.g. Infer) demonstrates that the LLM successfully closed the resource leak.

```
The following Java code has a resource leak from instruction
5 = new <Application,Ljava/util/Scanner>@10. Please fix it.

<Found Method with Resource Leak>

Use the try-finally pattern to close a resource.
Here's an example of using try-finally to close a resource:

BufferedReader br = null;
try {
    FileReader fr = new FileReader("file.txt");
    br = new BufferedReader(fr);
    br.readLine();
} finally {
    br.close();
}

Do not add extra words, for example avoid saying
"Here is the corrected code" and instead just output
Java code. Furthermore, only output compilable and
parseable Java code. Please maintain the code structure.
For example, do not add additional helper
or placeholder functions.
Only change the original function.
```

Figure 3: Prompt to LLM to fix example resource leak

```
1  class FoundMethod {
2    private static void insertQuoteAndMovie() throws Exception {
3      String sql, quote, movie;
4      File fin = new File("MovieQuoteTrivia.txt");
5  +   Scanner sc = null;
6  +   try {
7        sc = new Scanner(fin);
8        while (sc.hasNextLine()) {
9          quote = sc.nextLine();
10         movie = sc.nextLine();
11         sql = "INSERT INTO MovieTrivia (quote, answer)"
12               + "VALUES('" + quote + "','" + movie + "')";
13         s.executeUpdate(sql);
14         movies.add(movie);
15       }
16       System.out.println("Quote & Movie inserted");
17 +    } finally {
18 +      if (sc != null) {
19 +        sc.close();
20 +      }
21 +    }
22   }
23 }
```

Figure 4: LLM Output which fixes the example resource leak

3.1 Experiment details

The experiments were carried out on the same machine as RLFixer which has 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 GB RAM.

Since we were limited by time constraints, we were unable to use the OpenAI Batch API [15] which responds in at most 24 hours and limits the number of enqueued tokens to 90,000. Note here that a token is on average about 4 English characters. Across all of the resource leak prompts we had about 930,000 tokens to evaluate which we split up across 10 batch files. This means it would have taken possibly up to 10 days for the batch responses to finish.

We therefore switched to using the streaming API to get GPT-4o's responses more quickly though at twice the cost. To make the streaming requests, we simply iterated through the batch input files. We only moved on to the next resource leak after the API responded, meaning we didn't have to rate limit our requests and could rely on OpenAI's rate limiting. The batch input files took around 19 minutes to create for the 1825 unique resource leaks: most of which came from extracting the resource leak's enclosing method to pass to the prompt. That averages to 0.625 seconds per resource leak. Getting a response for all the streaming prompts with the rate limiting took 137 minutes, a bit more than 2 hours. Over all of the submitted resource leaks that averages to about 4.5 seconds per resource leak. The overall Leak2LLM workflow timing separated by each stage is consolidated in Table 1.

Create prompt (s)	Get LLM response (s)	Update method and tool runtime (s)	Total runtime (s)
0.625	4.5	32	37.125

Table 1: Average time taken per resource leak for each step of the Leak2LLM pipeline

Table 2 has the average time it took for each resource leak to be updated and analyzed for each static analysis tool. From Table 1, we can see that this takes the majority of the time when attempting to fix a resource leak.

Tool	Update method and tool runtime (s)
Infer	33
PMD	7
Checker-framework	58
SpotBugs	9
Average	32

Table 2: Time taken per resource leak to update the method and run the static analysis tool

In total we wrote 1240 lines of Python and 367 lines of Java. The Python code handled interacting with OpenAI such as creating the batch files, orchestrating calls to JavaParser, as well as logging intermediate files and calculating statistics. The Java code was used to run JavaParser to find the resource leak's enclosing method and later updating it with the LLM provided code. When updating the method, we cannot simply use the method name but need to also compare the method's parameter types to handle Java's method overloading. Together, these form the method's signature which uniquely identifies the method.

3.2 Comparison to RLFixer approach

RLFixer attempts to fix resource leaks by closing them after the last use of the resource. Therefore, it doesn't attempt to fix field or data-structure resource escapes since other methods may use that resource after it is closed in a method. Instead, RLFixer focuses on fixing the following resource leak escape types:

- resources that are passed to other methods (Invoke-Escape)
- resources that are returned from another method (Return-Escape)
- resources that are passed in as a formal parameter (Parameter-Escape)

In constrast with RLFixer, we attempt to fix all leaks regardless of their RLFixer fixable status. We do not change our approach depending if the resource leak is labeled as fixable by RLFixer. Similar to InferFix, we attempt to fix all resource leaks then classify them as fixed if the leak was resolved according to a static analysis tool.

4 EVALUATION

Figure 6 shows the fix rate across the different static analysis tools with an average fix rate of around 69%. Since we are unable to run CodeGuru locally then we do not attempt to fix any of the CodeGuru reported leaks. Since there were duplicate resource leaks found by different static analysis tools, we prioritized the tools that we could run locally. This meant that we attempted to minimize the number of CodeGuru warnings in our evaluated dataset. If multiple local tools were found we choose the first one. This makes our dataset slightly different than RLFixer but the majority of resource leaks are the same. As a result of this selection, Figure 5 shows that the number of unique CodeGuru warnings only accounted for around 4% of the evaluated dataset. While we did run Spotbugs, only 14 leaks were selected which was about 1% of the evaluated dataset.

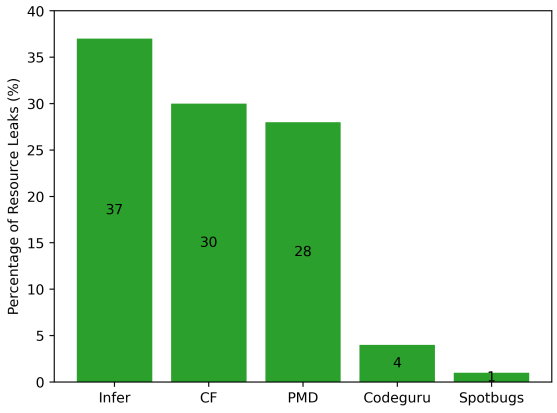

Figure 5: Percentage of resource leaks per tool in the dataset used to evaluate Leak2LLM

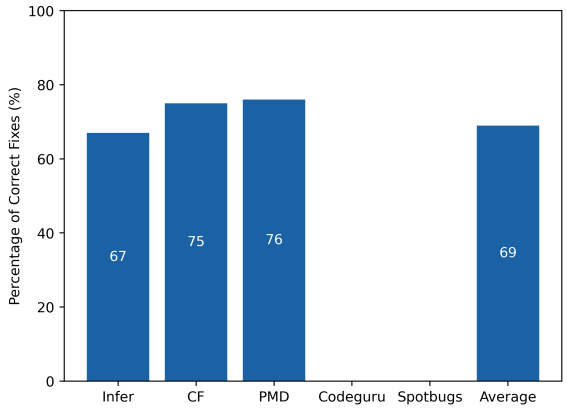

Figure 6: Percentage of correct fixes by Leak2LLM for each static analysis tool

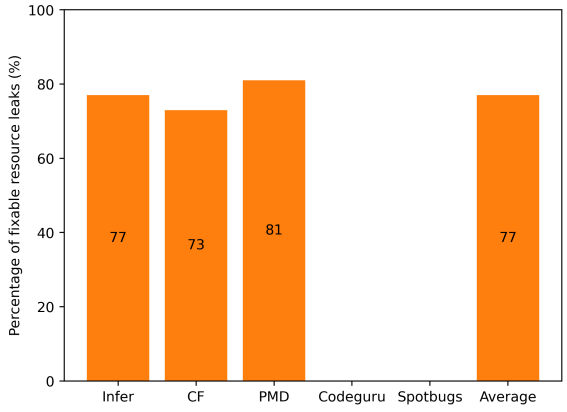

Figure 7: Percentage of fixed leaks classified as fixable by RLFixer for the static analysis tools

4.1 Comparison to InferFix and RLFixer Results

While RLFixer classifies resource leaks into fixable leaks (leaks that don't escape the enclosing method) and unfixable ones, InferFix relies on Infer verifying the leak is no longer present. Here, both InferFix and RLFixer fix roughly the same percentage of leaks fixed across their respective distinct datasets (71.2% for InferFix and 66% for RLFixer). Considering just the Infer tool, Leak2LLM fixes 67.3% of the Infer discovered leaks as shown in Figure 6. However, InferFix may have a slightly higher percentage of leaks fixed possibly because Infer classified leaks as fixed when they were not actually fixable (according to RLFixer). Using the InferFix leak verification technique, Leak2LLM fixes an average of 69.4% of the NJR leaks. Across NJR, RLFixer classified around 66% of the leaks as fixable. Figure 7 shows that if we only look at these fixable leaks, Leak2LLM fixes 77% of them, while RLFixer fixes 95% of them. However, these RLFixer fixable leaks account for 66% of all leaks meaning there's still work to be done on fixing resource leaks.

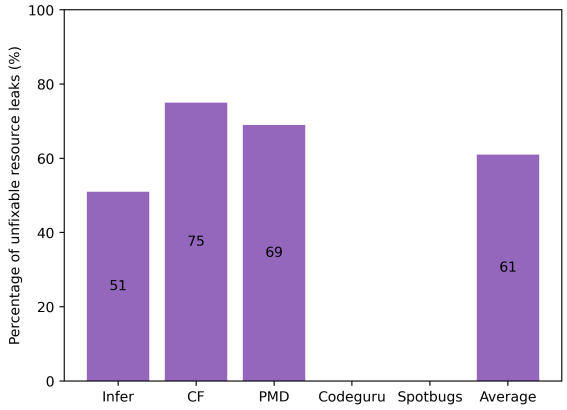

Figure 8: Percentage of fixed leaks classified as unfixable by RLFixer for the static analysis tools

Unlike RLFixer, we attempt to fix the RLFixer unfixable leaks. Figure 8 shows that according to the static analysis tools, we're able to fix on average 61% of the RLFixer unfixable leaks. However, even if the tool believes that we have fixed the leak, it may lead to new errors such as a new leak or a null pointer exception [20]. This is a similar limitation to InferFix which disregards the RLFixer fixable status and relies solely on the static analysis tool to detect whether a resource leak is fixed. The different fix rates for each category: any resource leak, RLFixer fixable leaks, RLFixer unfixable leaks for InferFix, Leak2LLM, and RLFixer is shown in 3.

Approach	Fix rate across all resource leaks (%)	Unfixable fix rate (%)	Fixable fix rate (%)
InferFix	71.2	N/A	N/A
Leak2LLM	69.4	61	77
RLFixer	66	N/A	95

Table 3: Average fix rate for all leaks and RLFixer fixable and unfixable leaks for different resource leak fixing approaches

5 DISCUSSION

5.1 Choosing an LLM

We initially experimented with running Code Llama models [18] locally. However, without a GPU and with limited memory we were only able to run up to the 13B model successfully and to achieve realistic throughput we were limited to the 7B model. Since models with more parameters and trained on larger datasets tend to perform better on a wide range of tasks [13] we decided to use the GPT-4o model from OpenAI. We initially chose GPT-4 Turbo but with the recent release of GPT-4o at half the price [16] we switched to using GPT-4o. Both GPT-4 Turbo and GPT-4o have a context

window of 128,000 tokens. GPT-4o is trained on slightly older data up to October 2023 compared to GPT-4 Turbo's December 2023. However, since we are testing its code generation abilities instead of recent information this is irrelevant.

5.2 RLFixer

RLFixer is created specifically for fixing resource leaks. Leak2LLM is an LLM-based approach that solves resource leaks but can be generalized to solve different bug types. InferFix shows that LLMs, in addition to solving resource leaks, are able to solve thread safety violations and null pointer dereferences.

5.3 InferFix

InferFix achieves a 71.2% fix rate for Java resource leaks over their InferredBugs dataset. Compared to InferFix, Leak2LLM requires less infrastructure and overhead since it only uses an LLM compared to InferFix's fine-tuned LLM and historic bug database. Their finetuned LLM (12B Codex) also has a much smaller context window of 2048 tokens compared to GPT-4o's 128,000 tokens. With a larger context window, GPT-4o can solve leaks in larger methods. While InferFix does utilize retrieval-augmented prompts, they also fine-tune the model which Lewis et al. [12] suggest is not necessary with retrieval-augmented prompts.

In addition to resource leaks, InferFix is able to fix 59.5% of Java null pointer dereferences and 77.4% of Java thread safety violations. Since we used the RLFixer resource leak dataset, we focused on solving resource leaks, although our approach is easily adapatable to solving other bug types. InferFix also integrates well into a CI pipeline where commits are analyzed using InferFix and if bugs are detected InferFix will provide a suggestion.

Similar to InferFix, the GPT-4o provided patch is re-evaluated using the static analysis tool that found it to verify that the leak was fixed. This eliminates LLM fixes that do not fix the actual resource leak, removing false positives.

One limitation of InferFix is that it relies only on the static analysis tool for checking if resource leaks are fixed. While Leak2LLM, uses the RLFixer resource leak dataset so we can decide between using RLFixer labeled fixable leaks or both fixable and unfixable ones.

5.4 Choosing resource leaks

RLFixer runs the following static analysis tools on the NJR dataset to create a list of resource leaks:

- CodeGuru
- SpotBugs
- PMD
- Infer
- Checkerframework

Since multiple tools may find the same resource leak, we need to determine which leaks are unique. RLFixer's dataset of resource leaks only provides a list of all leaks not the unique leaks it selected. We de-duplicated the leaks using the same warning equality heuristics as RLFixer as discussed in Section 3 but skipped the largest few Java programs which were on the order of a hundred thousand lines. We ended up with 1825 unique leaks compared to RLFixer's 2205 unique leaks likely because we skipped the larger programs. We prioritized non-CodeGuru tools since CodeGuru cannot be run locally. Otherwise, if there were multiple tools for one resource leak, we chose the first one listed.

6 FUTURE WORK

6.1 Solving other bugs in code

In contrast to RLFixer which uses data-flow analysis for solving resource leaks, Leak2LLM uses an LLM to fix errors. This means it could potentially solve a wider range of statically detectable problems in code. InferFix showed that with an LLM they were able to fix null pointer dereferences, thread safety violations, and resource leaks. We could extend their work by using multiple tools and fixing other detectable issues such as data races or loop hoisting.

6.2 Prompt Engineering

In-context learning provides examples to an LLM prompt of how it should respond [7]. Our prompt as shown in Figure 3 only provides an example of one fix. Providing multiple examples solving different kinds of resource leaks may lead to better performance. We could create something similar to InferFix's bug database to find similar fixes and pass those to the LLM.

InferFix creates an extended context using eWASH [6] with class-level fields, import statements, and methods that use the buggy code. Unfortunately, eWASH is not publicly available. However, with GPT-4o's larger context window, it's possible to prompt the LLM with the entire class or multiple methods, reducing the need for complex analysis.

NJR lacks unit tests for the provided program so we cannot automatically verify that the LLM provided code has the same behavior as the original code. Similar to handling Java compilation errors, if the unit tests fail, we could provide them as input for a new prompt. This new prompt could then be re-evaluated on the unit tests to see if the new code passes them.

6.3 SARIF

Since different static analysis tools have different formats and labeling of leaks, we had to use heuristics to check if two warnings are roughly equal. The Static Analysis Results Interchange Format (SARIF) [19], aims to be a common output format for all static analysis tools. PMD, CodeGuru, Infer, and Spotbugs all support SARIF. Even with SARIF, different tools may still classify resource leaks differently. However, a common output format will likely reduce the number of these differences, improving inter-tool output consistency.

7 CONCLUSION

We introduced Leak2LLM: a resource leak repair system based on GPT-4o and static analysis tools designed to fix resource leaks in Java programs. Leak2LLM is based on a retrieval-based prompt augmentation technique that utilizes the reports generated by static analysis tools. Our experiments demonstrated that Leak2LLM performs similarly to existing solutions, fixing 69.4% of the found leaks in the NJR dataset, while also being capable of solving longer methods and a larger range of bugs.

REFERENCES

[1] 2002. PMD Source Code Analyzer. https://pmd.github.io/

[2] 2017. SpotBugs Static Analysis Tool. https://spotbugs.github.io

[3] 2020. Amazon CodeGuru. https://aws.amazon.com/codeguru/

[4] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.

[5] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343.

[6] 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*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 4713–4722. https://doi.org/10.18653/v1/2021.emnlp-main.387

[7] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2024. The Java® Language Specification Java SE 22 Edition. https://docs.oracle.com/javase/specs/jls/se22/html/index.html

[9] JavaParser. 2019. https://javaparser.org

[10] 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*. 1646–1656.

[11] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D Ernst. 2021. Lightweight and modular resource leak verification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 181–192.

[12] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[13] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Nick Barnes, and Ajmal Mian. 2023. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435* (2023).

[14] OpenAI. 2024. https://platform.openai.com/docs/api-reference

[15] OpenAI. 2024. https://platform.openai.com/docs/guides/batch/batch-api

[16] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/

[17] Jens Palsberg and Cristina V Lopes. 2018. Njr: A normalized java resource. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 100–106.

[18] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[19] OASIS Static Analysis Results Interchange Format (SARIF) TC. 2023. Static Analysis Results Interchange Format (SARIF). https://www.oasis-open.org/committees/sarif/

[20] Akshay Utture and Jens Palsberg. 2023. From Leaks to Fixes: Automated Repairs for Resource Leak Warnings. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 159–171.