JSRT JOURNAL
JOURNAL OF SCIENTIFIC RESEARCH AND TECHNOLOGY
PAGES: 86-92
21/06/25
VOLUME-3 ISSUE-6 JUNE

A Model For Automated Code Debugging Using Small Language Models

Kevin Gwindingwi[1], Monica Gondo[2]

[1]*Department of Software Engineering, Harare Institute of Technology, Harare, Zimbabwe, kgwindingwi@gmail.com*

[2]*Department of Software Engineering, Harare Institute of Technology, Harare, Zimbabwe, mgondo@hit.ac.zw*

ABSTRACT

The purpose of this paper is to assess the performance of small language models in fixing golang concurrency bugs with a quantitative and qualitative analysis[1]. Comparing the results of the bugs fixes against the results of Gfix and Copilot using the gfix dataset. There have been a lot of models and frameworks that have been made to try and automate the repair of bugs. However very few attempts have been made to fix concurrency bugs and none was specifically made using a language model. As such this paper attempted to find if it is possible to solve concurrency bugs using small language models. Making use of the Gcatch/Gfix dataset to produce fixes for bugs and comparing them against copilot and GFix. We find that small language models are better than GFix in the quantity and variety of bugs they can fix but perform worse as compared to copilot.

Keywords— Automated Program Repair, Large Language Model, Go Concurrency, Retrieval-Augmented Generation.

I. INTRODUCTION

Software bugs are an unavoidable consequence of software development. The more lines of code written, the greater the likelihood of bugs occurring. This has a negative impact on software usability and may have greater risk to human life in some scenarios.(Garcia et al., 2020) “concurrency and memory bugs have been found in Autonomous vehicles”. Automated program repair has been extensively researched since the early 2000 with numerous system iterations being implemented, from Heuristic based APR to constrained based APRs[3]. The latest iteration of APRs involves the inclusion of LLMs in bug localization and patch generation. However, despite the amount of work in APR LLM integration since 2018, there has been very little work done in the area of concurrency and Hardware[3]. This research adds to the existing body of knowledge on APR bug fixing for concurrency with the integration of LLMs. Part of the reason for the lack of research in these particular areas is the lack of substantial data and the complexity of the problems that are trying to be solved[4][5]. As such, part of the task will be finding open-source repositories for the purpose of fine-tuning the LLM to the domain of interest which is concurrency. The languages that lend themselves well to concurrency include golang which already has other papers that address concurrency bugs that occur in the language[6].

II. LITERATURE REVIEW

Automated Program repair attempts to fix software bugs with minimal human intervention and is a crucial aspect of software automation. Humans are prone to errors and are not as thorough as a software program. In the event that automated bug fixing becomes a solved problem, there would be no need for humans to participate in bug fixing, reducing total time required to develop and deploy software. With the advent of LLMs, we have taken a step forward in automated code analysis, and patch generation. Most of current research has been focused on issues such as semantic bugs, security vulnerabilities, static warnings etc, there is very minimal research on areas such as API misuse, concurrency and hardware bugs[3]. For concurrency, there are four categories of bugs which are data races, atomicity violation, order violation and deadlocks[5]. It should be noted that other research has split the grouping between blocking and non-blocking bugs, this is especially true for go related bugs.[7]

Small Language models

Most research suffers from resource constraints, one such issue in this research is the financial ability to run Large Language Models to satisfy the requirements of the research. It is also a way of verifying if it is possible for other entities with resource constraints to see if they can take advantage of self-hosted models in their development life-cycle without having to have a big budget to pay for Chatgpt or Copilot. As such, it would be more prudent to make use of a fine-tuned Small language models or what we can call domain specific models. A small language model is an AI model capable of processing, understanding and generating natural language content[8]. SLM parameters range from a few million to a few billion as opposed to a Large Language Model with hundreds of billions or even trillions of parameters. Small Language Models are like LLMs in that they are based on the transformer model. There are multiple ways of creating SLMs and these include pruning, Quantization, low rank factorization and knowledge distillation. All these methods fall under model compression[8].

Fine-Tuning – It is the process of training a base model to improve the results returned in a specific domain. This is done by using training methods such as the alpaca instruction training or training on raw data (unsupervised training). The method of fine-tuning depends on the size of your dataset and the resources at your disposal to train the model. In our case, the dataset size was rather small made up of close to 100 json objects after cleaning.

Research Gap

Most APRs work using the generate and validate workflow whereby the buggy code is identified, a possible fix is generated, it runs in given test cases and if all test pass, the proposed solution is then recognized as a correct patch. (Zhang et al., 2024). However, from the very moment of identifying the buggy code, a lot of false positives may be picked up thee by resulting in correction of code that wa snot faulty to begin with. The APR then generates a fix for the code that is likely faulty and based on test results we verify whether the fix is correct or not. The APR is built using RAG and Fine tuning separately to assess which one performs better in solving concurrency issues in go. The APR will als be used to assess the viability of small language models for this use case to assess if developers can fix such issues without having to access external resources.

Conceptual Framework

The body of research that currently exists has seen to the creation of a lot of datasets and these datasets have come from multiple areas of study such as debugging or concurrency bugs[7] . These datasets are formatted using the unified diff format and as such the changes made in the code are easy to track. The source of these datasets is GitHub and as such , it is easy to verify the information if need be. The dataset will be ran through a Small language model comparing the results of a RAG supported SLM and a Finetuned SLM to see if they produce similar results or if differences exist. The dataset for RAG and finetuning will be the same to offer a better comparison between the two. The finetuning will be done using the Alpaca instruction training method. Whilst the data from (Liu et al., 2021) does present information in the unified diff format, it needs to be formatted into an instruction, input , output json format.

Empirical Literature

APRs have been around for a while with one of the earliest references being a 2012 paper for genprog which makes use of genetic programming and test cases for bug fixes.[11]. Over the years many other methods of creating APRs have surfaced, including templating methods. However since 2018, the patch generation research space has seen an increase in the number of published works focusing mostly on the use of large language models in patch generation. This includes ongoing works from Carnegie Mellon to automate the repair of identified c/c++ bugs picked up by a static analysis tool. The above example serves to highlight that the field of research is still worth exploring considering that some of the most prominent APR researchers are from this very institute, and they still find works that need to be considered in the APR space.

APRs have also seen implementation from industry such as SAPfix by Facebook [12]. Google has a more relevant research regarding the use of LLMs in the generation of patches for their own code base[13](Nowakowski & Keller, n.d.). The research highlighted a 15% success rate for few bug types such as data races and buffer overflow.

Research which is specific to the area of interest include CFix[14], which is an attempt at fixing concurrency bugs using a fix strategy design. Another research attempt of doing concurrency bug fixes by generating patches using genetic programming[14]. However there has been no attempt to fix concurrency bugs using Natural language processing[15][1].Effort has been made by [16] to create a data set for go concurrency bugs for the purpose of creating automated testing of go code.

The Publication trend between 2020-2024 for LLM based APR shows an ever growing number of publications with 65 papers being published in 2023 alone[1]. This shows the relevance of APR to the field of Software Engineering and how LLM success is tightly coupled with the likely success of APRs in practical applications. However as of now, it is likely that APR use is a niche application in production environments and in those environments it sees use, it is tightly monitored by real developer to ensure high quality output and avoid false positives.
Of the publications currently available, the vast majority have APRs that target Java, Python and C. The likely reason cited reason is availability of datasets for the languages.(Zhang et al., 2024). Of the total publication list, golang specific research constitutes only 1% of the total publications.

There exist Limitations to the current methods of Fault localization which have down stream effects on the possible fix that is generated. As stated by (Lee et al., 2024) Imperfect Fault localization only focuses on the specified area where the line of code is identified, these statements are then replaced by a fix that may or may not been fixing the issue but may even introduced additional bugs.

(Zhang et al., n.d.) Indicates that there is little to no research on concurrency and hardware bugs in the APR literature and the utility of LLM/SLM in solving such issues.
[5] gives us an extensive breakdown of the existing literature regarding automated program repair for concurrency bugs. Whilst a large body of research exists for detecting and analysing concurrency bugs, the research on fixing the bugs is limited and this extends to the body of research of LLM performance of detecting and fixing concurrency bugs. According to (Haque et al., 2022)Large language models suffer from the fact that they lack project specific context as such rather than use a large language model a domain specific language model will be used.

III. PROPOSED SYSTEM

Introduction

This section introduces the main idea behind the proposed concurrency model. Patch generation was handled by a small language model, specifically gemma3 with retrieval augmented generation to improve the results of the model. Other tests were performed using a small language model finetuned on the same dataset. The data was taken from its unified diff format and converted into an instruction/input/output json file. The cleaning meant that some data was lost as such the sample size decreased significantly.

Dataset

Previous concurrency research has created several datasets with concurrency bugs and as such there is no need to create a dataset from scratch. Available options include go bench [16][18] or asplos 2019 Dataset[7]. The Language of focus as is easily noted from the choice in dataset is go. Whilst it is not necessary to focus on a specific language, it is also true that golang is a relatively newer language with concurrency built as part of its core designed and not a lot of research has gone into it as compared to a language like JAVA. In order to avoid a situation of data contamination, a third distinct dataset is required, alternatively few shot was used initially to see if there is a high level of success with very little training done to the SLM. [19]
Gobench was designed in 2021 by Yuan and colleagues(Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization, 2021).It classifies concurrency bugs in go into blocking and non-blocking bugs. Blocking bugs include resource deadlocks and communication deadlocks. Non-blocking bugs include data races. GOBench has two test suites Goker and Goreal each with 82 bugs and 103 bugs respectively from real world systems such as CockroachDB.[18]
Bug detection will be done through another dataset. There existed the possibility of using another SLM to do the fault localization and emulate the work of [17], however that would introduce too many other factors and make it difficult to determine the improvement in performance and it would also increase the scope of work which is limited by the amount of time available.

Language model

The model that is used is the gemma 3 model designed by google. This is due to the fact there is tooling by google which helped streamline the process of training the model, google colab, unsloth and hugging face. The licensing of the model also makes it easy to access and from multiple references, it is easy to work with especially in areas of finetuning. Copilot was used as the point of comparison to determine if there is a significant difference between the performance of SLMs and LLMs . the third point of comparison was the historic data from gfix.

D. Research Materials/Tools

We used Ollama to host and Run our LLMs Locally.The PC in use did have an Nvidia Quadro T1000 gpu. Unsupervised finetuning and alpaca instruction fine-tuning where used to train the models. The use of unsupervised fine tuning saw the data output degrade and as such was abandoned early on. The research indicated that a larger data size would be required for the method to be effective.

Google Colab was the platform used to train the models due to the free credits afforded and the unsloth notebooks as reference for training different models. The tooling for LLMs has come a long way making the process far simpler. However, the ability to develop effectual fine-tuned LLMs is still a skill that takes time to acquire especially if no existing clean dataset is available.

The model was developed in python due to the wide availability of resources for python in LLM work.

Implementation of Code Fix Generation

For Finetuned fixes, the input was split into two with the buggy code being one part of the input recognized as input and the prompt being recognized as the instruction. Both would be fed to produce and output or result.

For the RAG, the instruction and the buggy code would make part of the initial input. A vector search result would then be added before it is all fed into the LLM. This would then produce an output.

IV. IMPLEMENTATION

RESULTS AND DISCUSSION

Introduction

The initial findings indicate that Using LLMs such as copilot is a better option as compared to SLMs such as gemma3, but it is also true that SLMs are slightly better than an APR like GFIX which uses specific strategies to fix concurrency bugs. However, our results are extremely limited due to the sample size used which was a result of time constraints.

Insights From the data

The model produced 6 possibly correct results out of 13 bugs. Whilst 6 fixes were successful, the level of specificity required in order to get the model to fix the issue would invalidate its utility in picking up bugs in a production environment. Copilot was also used on the same data to assess the degree of success obtained from a general LLM. In the case of Copilot, the large language model had more success producing 11 correct fixes for the 13 bugs. However due to the closed source nature of of copilot and its ability to reference the internet, it is possible that it was pulling results directly from repositories on github. This is noticeable due to a fix that also implemented a whole new method being generated as it is on the github commit. Two language models were used, gemma3 finetuned using a concurrency dataset and the base model as is with RAG(Retrieval-Augmented Generation). Better results where obtained from using gemma3 in its base form, which may be a result of the size and quality of the training data, it is also possible that the size of the training data and the skill required for fine-tuning where lacking. It is hard to determine if the implementation of RAG in the program had any impact on the quality of the output since copilot could not serve as a good control. A single comparison was done for a bug containing data race. The fix performed by both the RAG assisted model and copilot where very similar, however the wording was different, this might just be due to the different methods of prompting used as several iterations where required in some instances. The problems that are easily picked up by small language models are also problems that may be picked up by language servers as they have more to do with single line changes and any multi-line fix may not be picked up. As such its utility is questionable, it may be used as a reference point for possible issues with the code.

Exploratory Data Analysis

The total number of cases tested is 13 real world concurrency bugs found on the gfix dataset set which they obtained from GitHub commits. The models tested are the use of a base SLM with RAG , finetuned gemma, Copilot and data from the Gfix dataset[6]. The Bugs where from 6 major repositories. These are: docker, cockroach, bolt, kubernetes, etcd, grpc-go.

The bugs are of varying complexity with some being multi-line fixes and others only being a slight change In a single line such as buffering. The choice in regards to the specific bugs used was a result of accessibility, some of the links would return 404 errors and as such in order to save on time it is simpler to just move on to the next bug. There are more bugs from the docker repository because that was the first repository available in the dataset. All the bugs used

are included in an excel sheet and the results of the test are included in a zip file showing the results returned as well as the starting code.

Critical Findings:

Copilot had the highest success rate failing to return the correct result 2 out 13 times. Followed by gemma3 with RAG failing 7 out of 13 times and lastly Gfix and finetuned gemma with only 3 successes. Whilst the data is correct please note that GFIX primarily fixes BMOCc bugs (Blocking Misuse-Of-Channel bugs). Whilst the selection used is a mix of a lot of bugs.

Table 1. Summary of Results

Method	Success Rate	successes
Copilot	92.3%	12/13
Gemma3 with RAG	46.2%	6/13
Finetuned gemma	23.1%	3/13
GFIX	23.1%	3/13

Table 2. Types of Bugs Solved

Error type	Copilot	Gemma3 with RAG	Finetuned Gemma	Gfix	Total bugs
Struct field		1	1		1
Double Lock	3	2			4
Forget Lock	1	1	2		2
BMOCc	7	3		4	7

There is no exact rational as to why some did not have the correct bug fix and it may be down to poor prompting. Multiple attempts where made to produce correct results varying the prompt slightly all with no good result being produced. In some instances , the problem description from github was used as a prompt , but even then the SLM was unable to resolve the issue correctly.

Interpretation of Results

The utility of SLM in fixing concurrency bugs is limited, based on the results below, there is no statistical significance in using RAG, finetuned model or GFIX for concurrency bugs fixes. The only outlier is Copilot.

Comparison with existing Methods

Whilst SLM with RAG or finetuned models did not perform as well as we had anticipated, we suspect that with better training data and a larger data set the quality of the result would likely improve. The limitations of Gfix are its inability to scale very well and being limited to very specific context. With better training we believe that the variety of problems that can be solved with SLM is greater and the ability to take into consideration references outside the present function make the complexity of the problems that can be solved that much greater. With the limited time and resources, the finetuned gemma3 was able to perform as well as GFix with the Rag method solving more bugs than GFix.

Critical analysis of findings

Limitations/ Threats to Validity

Overlapping datasets and data leaks are a real concern with base model training data. We have no way of knowing the dataset used to train gemma3 and copilot does not include the data that we use to assess the model. The other issue is the internet access of copilot, if Copilot is accessing the internet to improve search results, we cannot objectively assess its ability to solve the problem from its own trained data and we do not have the resources to run it locally. We excluded go code from the bug assessment for fear of it being part of the training data for gemma3.

Validating patch correctness is also a challenge and code has to be manually inspected to ensure that it is validate. However [17] uses a similar method and we feel it is sufficient for our purposes, this is especially true for minor changes. As such we restricted the scope of tests to a single function.
The primary focus of this study was to assess the viable of a small model. And as such the size of the model is likely to have an impact on its performance, a such choosing the 1 billion parameter model may have severely hampered the chances of the language model producing significantly better results.

Comparison with existing methods

We have works such as DeepPerf that also use finetuning models that show that research into bug fixing is affected by false positives that pass test cases but may not be a correct fix.[20]. In regard to earlier research by [21],Du it held true that longer code sample resulted In poorer results, however the results we found suggested that there have been improvments in the LLM space resulting in better resulting in better results even when the problem is buried in multiple functions. However this is only true for Copilot but not for Small Language models. The Copilot is the model as provided by Microsoft in windows.

VI. CONCLUSION

Whenever the conversation to make use of a model comes up, the effectiveness of the model is always weighed against the cost of using it. In this paper we attempt answer whether the options available are able to meet our needs in regards to performance. The work done shows that there Is a marked improvement from the use of small language models as compared to heuristics or strategy-based solution due to the diversity of issues that can be tackled by the small language models. As such it scales well with and is not as limited as strategy-based solution which need to be adjusted when the type of problem being fixed changes slightly from the expected template. However, it is also clear to see that the success rate leaves much to be desired and may result in many false positives. Which means It is best If an individual oversees the bug fixing process.

REFERENCES

[1] Q. Zhang et al., “A Systematic Literature Review on Large Language Models for Automated Program Repair,” May 2024, [Online]. Available: http://arxiv.org/abs/2405.01466
[2] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, “A comprehensive study of autonomous vehicle bugs,” in Proceedings - International Conference on Software Engineering, IEEE Computer Society, Jun. 2020, pp. 385–396. doi: 10.1145/3377811.3380397.
[3] Q. Zhang, C. Fang, Y. Xie, and Y. Ma, “A Systematic Literature Review on Large Language Models for Automated Program Repair.” doi: 1.
[4] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, “LLM4SecHW: Leveraging Domain Specific Large Language Model for Hardware Debugging,” Jan. 2024, doi: 10.1109/AsianHOST59942.2023.10409307.
[5] H. Fu, Z. Wang, X. Chen, and X. Fan, “A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques,” Software Quality Journal, vol. 26, no. 3, pp. 855–889, Sep. 2018, doi: 10.1007/s11219-017-9385-3.
[6] Z. Liu, S. Zhu, B. Qin, H. Chen, and L. Song, “Automatically detecting and fixing concurrency bugs in go software systems,” in International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, Association for Computing Machinery, Apr. 2021, pp. 616–629. doi: 10.1145/3445814.3446756.
[7] T. Tu, X. Liu, L. Song, and Y. Zhang, “Understanding Real-World Concurrency Bugs in Go,” in International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, Association for Computing Machinery, Apr. 2019, pp. 865–878. doi: 10.1145/3297858.3304069.
[8] “What are Small Language Models (SLM)? | IBM.” Accessed: Feb. 05, 2025. [Online]. Available: https://www.ibm.com/think/topics/small-language-models
[9] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Semantic Similarity Metrics for Evaluating Source Code Summarization,” in IEEE International Conference on Program Comprehension, IEEE Computer Society, 2022, pp. 36–47. doi: 10.1145/nnnnnnn.nnnnnnn.
[10] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, “Automated Concurrency-Bug Fixing.”

[11] C. Le Goues, T. V. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 54–72, 2012, doi: 10.1109/TSE.2011.104.
[12] A. Marginean et al., "SapFix: Automated End-to-End Repair at Scale."
[13] J. Nowakowski and J. Keller, "AI-powered patching: the future of automated vulnerability fixes," May 2024.
[14] J. S. Bradbury and K. Jalbert, "Automatic Repair of Concurrency Bugs."
[15] H. Fu, Z. Wang, X. Chen, and X. Fan, "A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques," Software Quality Journal, vol. 26, no. 3, pp. 855–889, Sep. 2018, doi: 10.1007/s11219-017-9385-3.
[16] F. Tsimpourlas, C. Peng, C. Rosuero, P. Yang, and A. Rajan, "Go-Oracle: Automated Test Oracle for Go Concurrency Bugs," Dec. 2024, [Online]. Available: http://arxiv.org/abs/2412.08061
[17] C. Lee, C. S. Xia, J. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A Unified Debugging Approach via LLM-Based Multi-Agent Synergy," Apr. 2024, [Online]. Available: http://arxiv.org/abs/2404.17153
[18] Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization. Institute of Electrical and Electronics Engineers, 2021.
[19] Y. Dong et al., "Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models," Feb. 2024, [Online]. Available: http://arxiv.org/abs/2402.15938
[20] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, "DeepPERF: A Deep Learning-Based Approach For Improving Software Performance," Jun. 2022, [Online]. Available: http://arxiv.org/abs/2206.13619
[21] X. Du, M. Liu, J. Li, H. Wang, X. Peng, and Y. Lou, "Resolving Crash Bugs via Large Language Models: An Empirical Study," Dec. 2023, [Online]. Available: http://arxiv.org/abs/2312.10448