Program Repair Competition 2024

Ridwan Shariffdeen*, Yannic Noller†, Martin Mirchev*, Haifeng Ruan*
Xiang Gao‡, Andreea Costea*, Gregory J Duck*, Abhik Roychoudhury*
National University of Singapore*, Singapore University of Technology and Design†, Beihang University‡

ABSTRACT

This report outlines the objectives, methodology, challenges, and results of the first Automated Program Repair Competition held at the APR Workshop 2024. The competition utilized Cerberus, a program repair framework, to evaluate the program repair tools using different repair configurations for each track in the competition. The competition was organized in three phases: first the participants integrated their tools with Cerberus, second the integrated tools were tested using public benchmarks and participants were able to fix any identified issues. In the last phase, the submitted tools and baseline comparison tools were evaluated against private benchmark programs.

ACM Reference Format:
Ridwan Shariffdeen, Yannic Noller, Martin Mirchev[1], Haifeng Ruan[1], Xiang Gao, Andreea Costea[1], Gregory J Duck[1], Abhik Roychoudhury[1]. 2024. Program Repair Competition 2024. In *2024 ACM/IEEE International Workshop on Automated Program Repair (APR '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3643788.3648015

1 INTRODUCTION

We report the organization of the first automated program repair competition [9] (APR-COMP) at the 5th International Workshop on Automated Program Repair (APR) held on April 2024 in Lisbon, Portugal. The objectives of the APR Competition were (i) to evaluate the performance of program repair tools submitted on new benchmarks, (ii) to compare state-of-the-art using standard configurations, and (iii) to improve the set of standard benchmarks beyond the already studied benchmarks for repair tools.

The competition was organized in 3 tracks focusing on three popular areas of research in the community: (a) fixing logical errors, (b) repairing student assignments, and (c) fixing bugs in auto-generated code. Participants were given a few public benchmarks to develop, integrate, and test their repair tools. New challenges and benchmarks were curated but kept hidden from the participants until the end of the competition. For each track, in addition to the participating tools, we also compare the performance against two baseline tools. One of the tools is empowered by GPT4, while the other is a classical (non-learning) repair tool. The results were publicly announced on the competition website [1] after the teams have inspected them.

[1]https://apr-comp.github.io/

APR '24 , April 20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0577-9/24/04.
https://doi.org/10.1145/3643788.3648015

The competition evaluates 15 repair tools, including ten tools submitted by participants and five tools used as a baseline. The participant submitted repair tools are RepairLLAMA, RepaitCATJava, APRER, ET, GRT5, RepairCatPython, Brafar, F1X, ARJA, and ARJA-e. The five baseline tools are Darjeeling, VeriFix, TBar, Refactory, and LLMR. LLMR is a custom-made tool by the organizers, which is empowered by the OpenAI GPT4 model.

2 CERBERUS: COMPETITION PLATFORM

Cerberus [8] is a language-agnostic program repair framework that standardizes the evaluations of repair tools. It is a research acceleration platform [2] that enables researchers who are interested in evaluating their program repair tools against other state-of-the-art repair tools to launch large-scale experiments in a controlled and reproducible manner.

Cerberus consists of a large number of publicly available benchmark programs in C/C++, Java, and Python programming languages, taken from existing literature and have been integrated with multiple state-of-the-art tools. Benchmark programs cover a wide variety of repair tasks, including logical errors, security vulnerabilities, student assignments, concurrency bugs, and bugs reported by static analysis tools. Cerberus implements Configuration as Code, allowing all resource constraints (i.e., number of GPUs, memory limit), runtime configurations (i.e., experiment timeout), benchmark settings (i.e., fault localization) and repair tool parameters (i.e., optimization values) to be encoded in a configuration file which can be used to reproduce and replicate the results.

For each experiment run, all artifacts, including log files, are captured and compressed to facilitate post-analysis inspection or artifact evaluations. All collected artifacts from the competition, including the configuration files to re-run the evaluation, are made available for the community in Zenodo[2].

3 COMPETITION SETUP

The competition was announced [9] at the 4th International Workshop on Automated Program Repair held on May 16, 2023 in Melbourne, Australia. Upon consultation with many researchers in the community, the setup and the program were formulated. The competition was organized in multiple phases. In the first phase, participants were requested to integrate their repair tools into Cerberus. Participants were provided with sample benchmarks, which were publicly made available via Cerberus, to test and run local experiments. In the second phase, the organizers ran the integrated tools against the publicly available benchmarks on the competition platform (i.e., hardware) and shared the results with feedback on the performance, allowing participants to fix any issues with the integration. In the final phase, all integrated tools were executed on private benchmarks to compute the final results.

[2]https://doi.org/10.5281/zenodo.10531850

Table 1: Summary of the competition tracks

Repair Track	Language	Public Repository	Repair Tasks	Avg. Tests	Timeout	Resources Provided
Functional Errors	Java	https://github.com/APR-Comp/functional-java	50	161 / 100	1 hour	8 CPU, 2 GPU, 64GB Memory
	C/C++	https://github.com/APR-Comp/functional-c	25	119 / 100	1 hour	8 CPU, 2 GPU, 64GB Memory
Errors in AI-generated Code	Java	https://github.com/APR-Comp/autocode-java	100	13 / 100	15 mins	4 CPU, 1 GPU, 32GB Memory
	Python	https://github.com/APR-Comp/autocode-python	100	14 / 100	15 mins	4 CPU, 1 GPU, 32GB Memory
Student Assignments	C	https://github.com/APR-Comp/education-c-benchmark	100	19 / 100	15 mins	4 CPU, 1 GPU, 32GB Memory
	Python	https://github.com/APR-Comp/education-python-benchmark	100	19 / 100	15 mins	4 CPU, 1 GPU, 32GB Memory

3.1 Repair Tracks

The competition has three tracks - functional errors, student assignments, and errors in AI-generated code, each having two sub-tracks representing two popular languages in that domain. Table 1 summarizes the details of each track. Column "Avg. Tests" captures the average number of test cases per repair task in the format x/y, where x is the number of public tests, and y is the number of private tests. All files necessary to reproduce the results and reuse the benchmarks are provided in their repository (see Table 1).

Functional Errors. The functional errors track consists of in-production codebases written in C/C++ or running on the JVM infrastructure, with Java being the predominant language for developing the project. Some subjects are from the ManyBugs [6] and Defects4J [5] benchmarks. Examples of the C code bases are the CPython[3], PHP[4] interpreter, and LibTIFF[5] among others. The Java codebases are mostly from the Apache Foundation which includes Apache Commons Compress[6], Apache Commons Lang[7], and projects such as Google's GSON[8] among others. The majority of the bugs have recently been disclosed and fixed, which addresses the data leakage problem with pre-trained models. As the projects have a large enough test suite, we provide a subset of them as public and the rest as private.

Student Assignments. The student assignments track consists of incorrect student submissions, taken from the Refactory [4] and ITSP [10] benchmarks, which contain Python and C assignments, respectively. For both tracks, we deterministically selected 100 submissions for the public and private benchmarks. To avoid the risk of tools being pre-trained on the benchmarks, we create an obfuscation tool for both sub-tracks. The tool changes local variable names and applies syntactic transformations, which keep the solution semantically equivalent to the original submission. In addition, new submissions were generated by injecting faults into the correct solution. To create a sufficient public and private test suite, we take the default benchmark examples and provide more test cases by random testing the reference implementation to create concrete input-output examples.

Errors in AI-generated Code. The errors in the AI-generated code track comprises Python and Java sub-tracks. The code for these tracks is generated by OpenAI's GPT-3.5 and GPT-4 models as solutions for algorithmic problems in the LeetCode platform, following the work by Fan et al. [3]. All selected subjects have at least one passing and one failing test to help fault localization. Upon tool submission, we selected the most recent problems from LeetCode to generate partially correct programs using GPT-3.5 and GPT-4. The problems are of 3 categories - Easy, Medium, and Hard with a 1:2:1 distribution. To create a sufficient public and private test suite, we take the default benchmark examples and provide more test cases by random testing the reference implementation to create concrete input-output examples.

[3]https://github.com/python/cpython/
[4]https://github.com/php/php-src
[5]https://gitlab.com/libtiff/libtiff/
[6]https://github.com/apache/commons-compress
[7]https://github.com/apache/commons-lang/
[8]https://github.com/google/gson/

3.2 Baseline Tools

For each track, we select a state-of-the-art repair tool. For the Java language, we selected TBar [7] - a template-based Java repair tool. For the education Python track, we selected the Refactory [4] tool; for the education C track, we selected verifix [1]. For the AI-code Python and functional Python track, we selected Darjeeling. Alongside these tools, we created a shadow repair tool called LLMR, which uses OpenAI's chat completion models.

LLMR employs spectrum-based fault localization to infer possible fix locations, which is used to prompt a large language model for bug fixing. LLMR uses a simple prompt - "Here is the following file Y with name X. There is a bug Z. I would like you to repair the file and respond with a patched version. Here is the code: A" where X, Y, Z, A represent the file name, programming language, a bug description and the context of the file, respectively. In addition, for the education track the reference solution is also included in the input prompt. For larger context such as the bugs in the functional track, LLMR uses a context window around the fix location computed in the localization step. For the competition, we have chosen a context window n=10.

3.3 Platform and Configurations

All experiments were run in Docker containers to ensure a controlled environment. For the AI-generated code and student assignments track, we provide 4 CPU cores, 1 GPU core, and a max usage of 32GB of RAM to the tool, while for the functional errors track, we provide 8 CPU cores, 2 GPU cores, and a maximum usage 64GB of RAM. The difference in the resources is due to the considerably smaller subjects provided to the tools. To evaluate tools that are running locally and do not depend on a third-party service, we disable network access. All experiments were run on a 192-core Intel(R) Xeon(R) Platinum 8468V machine with 512GB of Memory and 8 Nvidia A40 GPUs, running Ubuntu 22.04.03 LTS.

3.4 Evaluation Criteria

We evaluate the tools according to the *effectiveness* of the patches they produce, and the *efficiency* of the entire repair process. Should a tie occur, we consider resource usage a secondary criterion. All

Table 2: Patch classification table

Patch Effectiveness	Score	Remarks
ill-formed	-4	*refused by validation*
invalid	-2	*syntactically incorrect*
incorrect	-1	*does not pass the public tests*
no patch	0	
incorrect overfitting	1	*fails passing tests*
overfitting	2	*passes only public tests*
correct	4	*passes all private tests*

patches undergo evaluation in an isolated container, ensuring a pristine environment for each subject.

Evaluation setup. Tools are executed for each repair task in their corresponding track, generating at most five patches per task. Each patch is applied to the original program in the repair task and executed against a *public test suite* – test cases made publicly available for all the tools – and a *private test suite* – test cases generated using a reference program (the expected correctly fixed version) not disclosed to any of the tools.

Patch Score. Table 2 summarises the different effectiveness categories a patch may fall into and their corresponding score. Patches that fail in the validation environment are deemed as "ill-formed". Those that are applicable yet render the file syntactically incorrect or the project non-compilable fall under the "invalid" category. Patches that do not pass the failing tests in the public test suite are marked as "incorrect". A patch that rectifies the failing tests in the public test suite but invalidates a previously successful test is labeled as "incorrect overfitting". We classify patches that succeed in the full public test suite but not in the private one as "overfitting". The highest score, "correct", is given to patches that pass both public and private test suites. It may seem odd to award incorrect overfitting or overfitting patches, but we consider that such partially correct patches have the potential to offer further insights with regard to what a correct patch is.

Task Score. Assuming a tool generates n patches for a task i, $n \leq 5$, the score S_i for task i is computed as $S_i = (\sum_{k=1}^{n} s_k)/n$, where s_k is the score for patch k, $k \leq n$. Our goal is to identify the most effective tools – produce the most correct patches – that also strive for efficiency – they not only produce correct patches but also minimize the manual effort required to validate the results. In other words, for a given repair task, a tool that produces three patches, all correct, scores higher than one that produces three correct patches and two overfitting or incorrect ones for the same task, $4 = (4 + 4 + 4)/3$ vs. $3.2 = (4 + 4 + 4 + 2 + 2)/5$ vs. $2 = (4 + 4 + 4 - 1 - 1)/5$.

Ranking Criteria (Track). For each track, the candidate tools are ranked according to their total score for the tasks in the considered track, computed as $T_j = (\sum_{i=1}^{m} S_i)$ for a track j consisting of m repair tasks. The quality of the validation discussed above tallies with the overall tool ranking, i.e., tools that consistently produce incorrect patches may not rank high even if they produce correct patches. For example, for a track with 100 tasks:

- a tool which correctly solves 40 tasks will score 160 (60 tasks have no solutions).
- a tool which correctly solves 50 tasks, and incorrectly solves the rest of 50 tasks will score 150.
- a tool which correctly solves 40 tasks, and incorrectly solves the rest of 60 tasks will score 100.
- a tool which correctly solves 20 tasks, and incorrectly solves the rest of 80 tasks will score 0.

Table 3: Summary of the auto-generated code track

Language	Tool	#IP	#PP	CP	Score
Java	**ARJA-e**	**0**	**0**	**25**	**20**
	ET	19	2	13	10
	APRER	0	18	5	8
	LLMR	25	0	0	-5.6
	TBar	6	0	0	-24
	RepairLLAMA	379	0	1	-105.7
Python	**RepairCatPython**	**0**	**56**	**0**	**16**
	Darjeeling	0	0	0	0
	LLMR	401	12	64	-26.3

IP: incorrect patches, **PP:** partially-correct patches, **CP:** correct patches

4 EVALUATION RESULTS

We discuss the results of each track and share important and interesting insights learned from the competition evaluation.

4.1 Errors in Auto-generated Code

We first focus on the repair task of fixing errors in AI-generated code. The track consists of two sub-tracks for Python and Java programming languages. Table 3 summarizes the performance of the tools in both sub-tracks. Columns "#IP", "#PP", and "#CP" depict the number of incorrectly generated patches, partially correct patches, and correct patches, respectively. For partially correct patches, we include both overfitting and incorrect overfitting. For the Java programs, ARJA-e, an evolutionary repair system, scored the highest. A notable distinction is that ARJA-e correctly fixed five bugs where all generated patches are correct (25 patches with five patches for each bug). For the Python programs, RepairCatPython, an LLM-based repair tool, scored the highest. Notably, LLMR scored lowest for the Python sub-track, although it generated 64 correct patches, which is more than the highest-scoring tool RepairCatPython. The reason for the lowest score is the large number (401) of incorrect patches generated by LLMR. Our scoring schema penalizes a tool for generating incorrect patches, which would decrease trust in the tool and increase the manual efforts a developer has to invest. For both sub-tracks, the performance of LLMR reduced, with more than 84% of the patches generated being incorrect.

4.2 Student Assignments

Next, we focus on the repair task of providing useful feedback for incorrect student assignments. In this task, the repair tool is provided with an incorrect student-written program for a given assignment, a test suite to validate the program, and a reference solution. The objective of the repair tool is to provide a "fix" for the incorrect student program, which is custom feedback to the student. In the competition, we ran two sub-tracks for Python and

Table 4: Summary of the student assignments track

Language	Tool	#ST	#CP	Accuracy	Score
Python	**LLMR**	**87**	**367**	**75%**	**339.6**
	Brafar	74	74	75.5%	308
	Refactory	23	23	71.8%	92
C	**LLMR**	**44**	**144**	**46.3%**	**153.3**
	VeriFix	7	7	100%	28
	F1X	1	2	28.5%	6

ST: successful tasks, **CP:** correct patches

C. For each sub-track, participant tools are evaluated against a state-of-the-art tool and LLMR. Table 4 summarizes the results of this track. Column "#ST" depicts the number of assignments the tool was able to find a correct fix. Notably, for the Python track, the competing tool Brafar outperformed the state-of-the-art tool Refactory with a significant margin, correctly fixing 230% more assignments. However, the GPT4-powered tool LLMR outperformed all tools in correctly fixing student assignments in both sub-tracks, with more than 75% of the generated patches passing the public tests. This observation indicates that GPT4 is capable of providing feedback for student assignments by generating the correct fix for the incorrect program.

4.3 Functional Errors

Lastly, we analyze the performance of the repair tools in fixing functional errors from real-world programs. The competition evaluated the tools for programs written in Java and C programming languages. For the Java programs ET, an LLM-based repair tool emerged as the winner. For C programs, F1X, a search-based repair tool, scored the highest. We notice a considerable degradation in the performance of the repair tools in this track compared with the previous two tracks.

We attribute this observation to incorrect fault localization that fails to identify the fix location correctly. Evaluated tools could not find a fix location, and generate a plausible patch for most of the bugs in the track. Some of the tools used old versions of fault localization, which did not perform well on real-world applications that use more recent versions of the execution environment and dependent libraries. For example, most of the programs are compiled on Java-11 and higher versions, which are not supported by the underlying fault localization tools used by the competing tools. In addition, the usability and scalability of the repair tools need improvement to repair bugs in large-scale real-world programs.

Similar to the previous two tracks, we evaluated the efficacy of LLMR in this track. Given the restriction on context limit, we only provide the n(=10) lines as the context window for the program. For each location in the top-5 candidate fix locations computed using fault localization, LLMR will be provided with a context of n lines before and after the faulty line. Interestingly, however, we observed that LLMR could not fix any of the real-world programs.

5 CONCLUSION AND FUTURE WORK

The first international competition on Automated Program Repair evaluated 15 repair tools consisting of 10 participant tools and five baseline tools. The competition assessed the tools on three repair tracks, each consisting of 2 programming languages, totaling up to 475 repair tasks. The benchmarks were curated with new unforeseen repair challenges, and all tools were evaluated in a uniform environment that enforced strict resource constraints.

Future editions of the competition aim to incorporate more input from the community with proposals for new competition tracks, more tool submissions, as well as proposing new benchmarks. More benchmarks with diverse repair challenges across many programming languages can help the community provide a basis for fair assessments to identify the state-of-the-art repair tools.

ACKNOWLEDGMENTS

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

The five year research program on automated program repair funded by the Singapore Ministry of Education, envisions automated repair capabilities to be integrated into programming environments in the future. This competition and the setup is the initial steps to accelerate the research towards this goal.

REFERENCES

[1] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 74 (jul 2022), 31 pages. https://doi.org/10.1145/3510418

[2] Earl Barr, Jonathan Bell, Michael Hilton, Sergey Mechtaev, and Christopher Timperley. 2023. Continuously Accelerating Research. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 123–128. https://doi.org/10.1109/ICSE-NIER58687.2023.00028

[3] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. arXiv:2205.10583 [cs.SE]

[4] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 388–398. https://doi.org/10.1109/ASE.2019.00044

[5] 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[6] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[7] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[8] Ridwan Shariffdeen, Martin Mirchev, Yannic Noller, and Abhik Roychoudhury. 2023. Cerberus: a Program Repair Framework. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 73–77. https://doi.org/10.1109/ICSE-Companion58688.2023.00028

[9] Ridwan Shariffdeen, Martin Mirchev, and Abhik Roychoudhury. 2023. Program Repair Competition. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 19–20. https://doi.org/10.1109/APR59189.2023.00010

[10] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 740–751. https://doi.org/10.1145/3106237.3106262