A Smart Assistant for Debugging

Steven P. Reiss
spr@cs.brown.edu
Department of Computer Science, Brown University
USA

Mohamed Salem
salem.medhat@edu.pte.hu
Computer Science Engineering, University of Pécs
Hungary

Abstract

We introduce our dynamic intelligent assistive debugger, DIAD, as a smart assistant for debugging within an IDE. DIAD uses an LLM to integrate a variety of data sources useful for debugging, including source analysis, debugger interrogation, queryable flow analysis, simulated partial execution with full traces, and trace analysis. These additional sources let it provide more accurate and complete debugging information to the developer in a timely fashion. DIAD runs automatically whenever the developer encounters a breakpoint, detecting if there is a symptom of a potential problem, and if one is found, tries to explain and, if possible, fix the underlying problem using the various sources. DIAD includes a user interface that lets the developer ask debugging questions or followup on the responses. We have done a preliminary evaluation of the still experimental system and plan to use it to drive future work.

The source code for the system and its various components is available on GitHub. A video demonstrating the tool and how it is used is available at https://youtu.be/fLiStsz4PJ8.

CCS Concepts

• **Software and its engineering → Software creation and management**; *Software development techniques.*

Keywords

Debugging, Interactive Repair Framework, Automated Program Repair, Integrated Development Environment

ACM Reference Format:
Steven P. Reiss and Mohamed Salem. 2026. A Smart Assistant for Debugging. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834620

1 Introduction

Developers use a wide variety of information while debugging. They use knowledge of the program's structure; knowledge about the current execution, such as the call stack and the values of variables; knowledge of the current execution, such as previous values and how and when a variable was updated; knowledge of what particular functions are supposed to do; knowledge of what code is solid and what code might be new or relatively untested or overly complex; knowledge of similar problems that have arisen in this and other systems; knowledge gained from previous debugging runs; and general knowledge as to what types of things might have gone wrong.

Developers use this information in many different ways depending on the nature of the problem, its symptoms, how it is discovered, its complexity, their inherent knowledge of the system, and their overall experience with both the particular system and with debugging in general. The specific information needed for a particular problem can be difficult to predict and can vary considerably depending on the problem, the underlying system, and the developer.

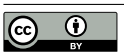
Although tools exist to provide much of the needed information, such tools are not convenient to use for interactive debugging and are not integrated into a unified approach. The tools can be difficult to run, can take considerable time, and the information they provide is not consolidated — that is left to the developer. We use a large-language model (LLM)-based approach to remedy this situation. LLMs are good at synthesizing multiple sources of information. They can summarize and combine the results from these sources. They can determine which sources are particularly relevant to a specific question.

Our approach, embodied in our system DIAD, is to create a flexible interactive framework within an IDE using a LLM which has been augmented with tool agents that provide the additional information needed for debugging. This framework can then interactively answer developer debugging questions by drawing information from these tools as needed. It can also help the developer understand potential problems in a timely manner, generally in under a minute.

DIAD has been developed for complex, event-driven, multi-threaded, interactive Java programs. It makes use of a variety of static and dynamic analyses. It is always active while the developer is debugging. It offers interactive assistance. It can explain why problem symptoms occur and often propose and make repairs. It can validate local repairs by seeing if they fix the symptoms of the original problem. At the same time, DIAD is different from other approaches. It does not depend on having a test suite or even a test program. It does not require rerunning the application, but instead works with the current run-time state through the debugger. It does not require the current stack to reflect a complete program run or test case; it can just reflect an event callback.

The contributions of this research include:

- A framework for debugging that integrates a LLM into a development environment as part of the debugger.
- A interactive user interface for LLM-based debugging that automatically analyzes the situation at a stopping point to identify a potential problem and then can describe its root cause and potential repairs.
- A set of agents that integrate advanced program and execution analysis to assist a LLM in debugging.



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834620

2 Our Approach

In the last decade we have developed and incorporated into an IDE a variety of tools to provide information that can potentially assist the developer while debugging. These include:

- **SEEDE:** This is a live programming tool that can start at the beginning of a routine on the run time stack during program execution. It executes that routine and anything it calls and yields a complete execution trace including graphic output that the developer can analyze using a time slider. It reexecutes this routine on any program change [12].
- **Brepair:** This tool supports debugging a failing test case. It does classical program spectra analysis over a test suite based on Ochiai [1] combined with user feedback to do fault localization [8]. It then uses SEEDE to let the developer manually explore the failing test execution.
- **FAIT:** This does an abstract-interpretation-based flow analysis which is continually updated as the program is edited [10]. While designed for analyzing security problems as the user codes, it provides a query interface that can be used to obtain a backward slice for a particular value or values, showing their dependencies and how they might have been computed.
- **ROSE:** This tool provides an interface for interactive bug repair where the developer can specify a symptom at a breakpoint while debugging and the system attempts to find program repairs that will fix that symptom using a variety of patch generators [11]. This tool uses FAIT for fault localization based on analysis of the symptom and SEEDE for validating potential patches.
- **LIMBA:** This tool provides a generic interface to an LLM using Ollama that provides basic tool agents for accessing and understanding the source code. It can do test-based code generation where it repeatedly asks the LLM to fix code until it passes all the tests. It also provides source code to the LLM using a RAG. It works with optional descriptions of the overall system and the desired coding style.

DIAD's goal is to use an LLM to integrate the relevant information from these and other sources in order to provide assistance while debugging.

DIAD works by monitoring a process being debugged in an IDE. Whenever a thread in the process stops during execution, DIAD starts by analyzing the stopping point and the relevant source code to determine if there is a potential problem, identifying the symptom of that problem. For example, if the thread stopped due to an exception, it assumes that the exception should not have been thrown; if the thread stopped in defensive code, it assumes that execution should not have gotten to that point. If a symptom is found, DIAD analyzes it to identify critical values and locations in the source code at the stopping point. Then DIAD uses the FAIT query mechanism to do a backward slice for those values and locations to find the set of program points that can affect the symptom and hence might be the location of a potential program fault. This slice is somewhat restricted, looking only at ten levels of variable assignment.

Next, DIAD identifies a routine on the current stack to start simulated execution from, attempting to include as many of the identified locations as possible. Once a starting point is found, DIAD

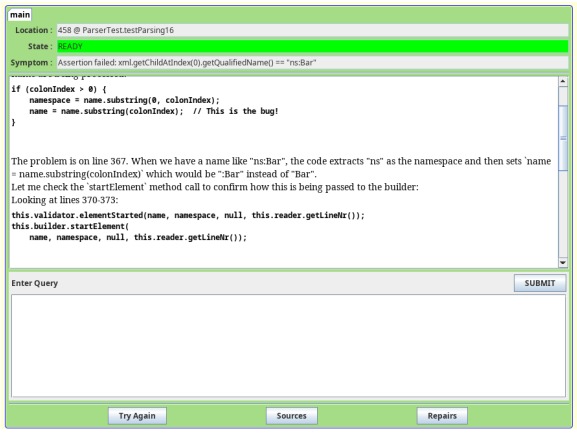


Figure 1: DIAD display panel showing a symptom and its explanation. The top lines show the thread, breakpoint location, DIAD's status, and a description of the symptom. The text area in the middle shows the explanation and the text area at the bottom allows user queries.

uses SEEDE to get an execution trace from the start of that routine that includes the current stopping point and duplicates the identified symptom, possibly resetting initial values so that the simulated execution matches the actual execution at the stopping point. The next step is to find a restricted set of relevant program locations by only considering those that were covered in the simulated execution.

Once all this is done, DIAD generates a query to the LLM using LIMBA, asking it to explain the problem. This query includes a detailed description of the identified symptom, information about the system, hints as to what approaches might be taken, and a description of the information available to the LLM. It provides the LLM with tool agents to access the SEEDE simulated execution trace; to retrieve the relevant locations, either the complete list or the list of those that were executed; to query the debugger to access the stack or evaluate arbitrary expressions; to access the source code; to get syntactic and semantic information about the system; and to analyze the execution trace to provide information about how a particular variable or value was computed.

If DIAD cannot use the flow analysis to do fault localization or it cannot find a matching simulated execution, it will provide the LLM with the information that it does have, limiting the set of agents, and still ask for an explanation. Our experience and that of other systems shows that even without this extra information the LLM can provide a useful analysis, although not as complete or accurate. This facility broadens the applicability of DIAD since there are cases where the flow analysis can fail (for example because of issues involving the use of reflection), or the simulated execution might not be able to duplicate the symptom.

DIAD includes a front end to display the relevant information to the developer, to let the developer change or provide additional information, to let the developer ask questions or follow up on responses, and to find and make repairs to the program. This is

shown in Figure 1. The front end is part of the IDE debugger interface and is automatically updated at every stopping point based on the above analysis, with separate tabs for different threads.

The top lines of the panel show the thread as the tab label, the stopping point, and the current DIAD processing status. The latter is color-coded and will revert to "No Symptom Found" if the stopping point does not seem to reflect a problem. The symptom line provides a description of the symptom found by DIAD. A more detailed version of this is passed to the LLM.

A pop-up menu lets the user change the symptom, the LLM model, the starting routine for the simulated execution, and various parameters for the simulated execution. The developer can specify or change the symptom that DIAD is analyzing. This is useful where the symptom might not be obvious or correctly identified by DIAD. This can be simple, for example, the developer can specify a variable that has the wrong value and possibly provide constraints on what the value should be. It can also be arbitrary with the developer providing their description of the symptom which will be passed to the LLM. In this case, the developer is asked to optionally identify one or more variables that affect the symptom so that flow-based fault localization information can be provided to the LLM. Specifying a new or modified symptom will cause DIAD to restart its analysis. The developer can change the starting routine for the simulated execution to any routine on the current call stack. This is useful in cases where simulated execution can not duplicate the original, for example, if a routine on the stack reads input from an external source. The developer can also change the LLM model that DIAD should use. They can choose between any of the open source models loaded into Ollama or a commercial model if the developer provides appropriate keys for OpenAI, Anthropic, or Gemini.

Below the description lines of the user interface is a text panel showing a simplified version of the prompts and the responses from the LLM. This starts with the initial explanation that DIAD finds. The user can ask their own debugging questions or follow up on previous responses by entering text in the bottom box, with the questions and responses shown in the text panel. The buttons at the bottom can be used for quick prompts, either asking the LLM to retry the explanation, useful due to its stochastic nature; to have the IDE display the source code associated with the current response; or to show the repairs suggested by the response. The user can also provide additional information and then hit the "Try Again" button to ask the LLM to try again with the additional information. This useful when the LLM hallucinates or goes off in the wrong direction.

If the user clicks on the "Repairs" button to see potential repairs, DIAD will create a separate display as shown in Figure 2. The changes are shown using different colors in the original and patched code. The display lets the developer review the suggested patches, excluding any that they feel are wrong or should not be made. If they are satisfied, they can click on "Make Repairs" to actually edit the code. As part of this display, DIAD will attempt to validate that the patch fixes the symptom. It displays the validation status in the colored top line of the display. Validation is done by having SEEDE redo the simulated execution using the patched code, and then comparing the new trace with the original to check if the symptom went away and the code ran to completion.

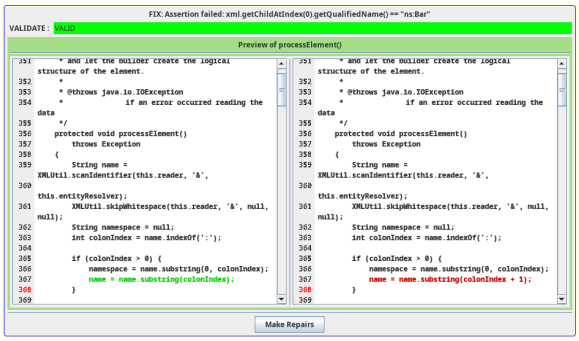


Figure 2: DIAD patch display show suggested repairs for the problem. The validation status of the repair is show at the top in green. The changes from the original to the patched version are shown in different colors.

Realizing that DIAD is useful outside of the debugger, we have added a separate user interface for debugging from a stack trace. This interface lets the developer paste a stack trace that can be obtained from a bug report, a crash log, or a trace file. DIAD then does symptom analysis based on the exception causing the stack trace, and uses FAIT to do appropriate fault localization for that symptom. No attempt is made to simulate the execution. DIAD then prompts the LLM to provide an explanation of the trace. It uses the standard display window (Figure 1) to show the result and let the developer do interactive exploration and repair.

3 Related Work

Most early work on debugging assistance has focused on fault localization and bug repair in the context of a test suite and assumed a program that can be easily rerun. An early example of a debugger assistant is **WhyLine** [6] which reran the program to get a detailed trace and then could answer questions about the execution. More recent work in this area has demonstrated the capabilities of LLMs to explain errors and find correct patches either from test cases or from bug reports.

Addressing the problem of providing assistance from within the debugger has been more recent [7, 9, 13]. This adds several difficulties, especially for complex, interactive programs. One cannot assume the program can be rerun to explore the execution. There is no test suite that can provide information for fault localization. The run time stack is often partial, starting with an event handler or a worker thread and may not include the fault.

ROSE attacked this problem, letting the programmer define a problem symptom at a stopping point and then attempting to find a program repair for that symptom [11]. It did this without rerunning the application and without a test suite. ROSE used a variety of techniques for finding patches including asking a LLM to find a correction for a particular source line. It was limited to particular types of failures. Much of the DIAD analyses are based on the techniques used successfully in ROSE with extensions to handle other types of failures and to provide more general assistance. ROSE relies on brute-force techniques to find repairs and does not offer

a root cause analysis. DIAD uses the LLM to provide root cause explanations and repairs.

ChatDBG [7] and **Cursor** [4] take a different approach, integrating an LLM with the debugger so that the LLM controls the debugger through agents. They have demonstrated strong interactive bug diagnosis. This approach does not provide a unified integration of extensive static-flow, slicing, and repair validation for various debugging contexts, which are limitations [5]. DIAD differs by first limiting access to the debugger to only query operations that do not affect the execution, and second by providing the LLM with information based on flow analysis, in particular potential fault locations, and by providing a detailed program trace relevant to the symptom.

Another related system is **debug-gym** [13] which provides a text-based environment for interactive Python debugging. It considers debugging as active information seeking rather than one-shot patching. This system is fundamentally a text-based testing sandbox, while DIAD is a complete debugging assistant integrated into the IDE with additional debugging-specific tools. A final related system is the **AI-Debugging Assistant** [2] which uses a LLM with a RAG to access the source and tools to analyze the fixes generated by the LLM to provide a student assistant to help teach debugging skills.

4 Implementation and Evaluation

DIAD is implemented as a separate process that communicates with the IDE, SEEDE, and FAIT using a message bus. The back end is implemented on top of the Eclipse IDE using a plugin to communicate messages. We have experimental plugins for IntelliJ and LSP (VSCode) that demonstrate the feasibility of using DIAD with other environments. The relatively small user interface front end is implemented as a plugin for the Code Bubbles IDE [3].

DIAD itself creates a worker thread for each stopped thread. This worker goes through the various processing steps described in Section 2 from finding a symptom to querying the LLM, generating messages to the front end for each step. DIAD listens for command messages from the front end to request any additional processing such as follow up queries or updating the symptom.

We experimented to determine appropriate prompts that eliminated or minimized hallucinations and directed the LLM to act appropriately. We also experimented with a variety of LLMs, settling on *qwen3-coder:latest* as having a good balance of accuracy and performance while running locally to ensure privacy.

While DIAD is still in the prototype, experimental stages, we have done preliminary evaluations using NanoXML, a simple XML parser used in prior bug repair studies, and using an event-driven system of our own that has not been used in LLM training, TC1. TC1 is a JAVA order processing system (~11KLOC) that stimulates e-commerce operations in which we planted 4 relatively simple bugs in the code representing typical errors and exposed them with test drivers to automate evaluation. We tested the system both with and without access to the SEEDE trace and FAIT analysis. In the latter case DIAD still provided the LLM with a detailed description of the symptom, access to the debugger to access the stack and evaluate expressions, and source information. The latter is roughly equivalent to the approach of ChatDBG or Cursor Debug and can be used as a comparison basis using the same LLM. For each test case we obtained and analyzed transcripts with the LLM including the set of agent calls that were made. The stochastic nature of the LLMs made repeatable testing somewhat difficult. We have also been using the system ourselves on itself and other systems we are working on to get some sense of its practicality and limitations.

The NanoXML problem is a parsing bug that shows up when retrieving information from the parsed structure. It is correctly repaired by fixing the parser; an incorrect but working repair fixes the retrieval method. DIAD was able to correctly diagnose and repair this problem almost all the time (LLMs are stochastic) However, when run without access to the FAIT and SEED tools, it misdiagnosed the problem about half the time and was never able to find the correct repair. This shows the advantage of using analysis tools along with the debugger.

DIAD was also able to correctly explain the four problems in TC1 much of the time. Two of them were handled correctly almost all the time, both with and without our analysis. The third was handled correctly most of the time by the full analysis and about 75% of the time without analysis. The fourth case is more problematic since an overfitted simple repair (removing a condition) seems a viable alternative to the preferred repair (negating half of the condition). In both cases, the explanation contained the preferred patch about half the time. However, the runs with analysis provided the correct repair slightly more often than the runs without analysis.

This preliminary evaluation is promising in that it shows that providing the analysis to the LLM can produce better, more accurate, results, especially in cases where the bug is not obvious and in a routine directly called from the test driver. It also shows that the LLM can do a reasonable job without the analysis in simpler cases.

Our experience when running DIAD with the Code Bubbles debugger is mixed. We have found the tool to be unobtrusive so that it does not interfere with normal debugging operations. However, it has not been particularly useful for the types of debugging we have been doing because of the mature state of the systems we are looking at. Most of the bugs we have found are either too complex (for example, timing problems involving multiple threads); involve new code that needs to be written instead of code that needs to be fixed; cannot be easily simulated (for example, involve reading an image from a web cam); did not result in the program stopping at a breakpoint; or require manually defining a somewhat complex symptom. This is the basis for future work.

We will continue to work on DIAD with an emphasis on extending its utility for everyday debugging. Once we feel that the system is mature and stable enough, we plan to do a system study that will quantify our subjective opinions about the accuracy of the approach, the time DIAD takes to assess a problem which now ranges from seconds to minutes, and the advantages of providing the additional information. We are currently working on developing a suite of non-trivial bugs of varying complexity that can be exhibited while debugging and are not part of a test suite and have not been used as input to a LLM. We also plan a user study to determine the utility of this approach to developers while they are debugging. We will provide a group of users with several bugs and have them try to repair the bugs with only the debugger, with the debugger and DIAD running without the analysis tools, and with DIAD and the analysis tools.

5 Future Work

DIAD is a continuing effort. Based on our experience using it during development, we are concentrating on making it more broadly applicable and generating more accurate results. The latter will come to a large extent with newer, more powerful LLMs.

Broader applicability involves being able to analyze additional symptom types such as a broader range of exceptions. It also means handling more complex problems such as those where the error occurred prior to our outside of the current execution stack. This includes problems with event handling where the problem occurred before the event, and problems involving multiple interacting threads.

Handling problems outside the current stack can be difficult. The set of locations that DIAD uncovers will include potential problems in this category. However, repairs for such problems cannot be validated and might have unexpected side effects. Developers might handle such problems by setting a breakpoint at a point earlier in the program and rerunning the system or hoping that a future run would provide more information. For example, the Cursor IDE debug mode adds logging for these instances [4]. DIAD has enough information to insert such breakpoints and could automate this process. SEEDE could also be used to validate a prior change by redoing the current execution with different initial values.

Another direction we are pursuing is to generate a test case that will duplicate the symptom and can be rerun easily. Rather than using non-standard serialization as others have done here, we are attempting to find a sequence of calls that will set up the appropriate initial environment to call an visible method. A side effect of a SEEDE execution is an enumeration of the initial environment that is required for this task.

6 Summary

DIAD is an attempt to use LLMs to amalgamate a variety of dynamic and static program analyses to support interactive debugging. It can be used without an extensive testing framework. Initial studies have shown the effectiveness of the approach. It is undergoing continuing development.

DIAD's contribution include its integration into an IDE, an interactive user interface that works proactively to describe the root cause of a symptom and possible fixes, and a set of agents that enable a LLM to provide an accurate analysis.

We are interested in others extending or using this work and welcome collaborations, feedback, and joint research.

7 Data Availability Statement

DIAD and the related software are available as open source. The sources can be found in https://github.com/StevenReiss/ for projects: diad, diadbb, limba, limbabb, fait, faitbb, seede, seedebb, bubbles, bubjet, lspbase, and ivy. Binary versions of these packages are part of the Code Bubbles environment or its plugins and are available from https://www.cs.brown.edu/people/spr/codebubbles. The source for TC1 and our version of NanoXML are available on zenado with DOI 10.5281/zenodo.21474418 or at

https://www.cs.brown.edu/people/spr/diadtest/.

Transcripts of our test runs and a video are also available at these locations.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* (2009), 1780–1792. doi:10.1016/j.jss.2009.06.035

[2] Elizaveta Artser, Daniil Karol, Anna Potriasaeva, Aleksei Rostovskii, Katsiaryna Dzialets, Ekaterina Koshchenko, Xiaotian Su, April Yi Wang, and Anastasiia Birillo. 2026. Enhancing Debugging Skills With AI-Powered Assistance: A Real-Time Tool for Debugging Support. *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering* (2026). doi:10.1145/3786580.3786976

[3] Andrew Bragdon, Steven P Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 455–464. doi:10.1145/1806799.1806866

[4] Cursor Debug 2026. *Debugging with Cursor.* https://cursor.com/for/debugging

[5] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proc. ACM Softw. Eng.* 1, FSE, Article 64 (July 2024), 23 pages. doi:10.1145/3660771

[6] Andrew J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158. doi:10.1145/985692.985712

[7] Kyla H Levin, Nicolas van Kempen, Emery D Berger, and Stephen N Freund. 2025. ChatDBG: Augmenting Debugging with Large Language Models. (2025). doi:10.1145/3729355

[8] Xiangyu Li, Marcelo d'Amorim, and Alexandro Orso. 2016. Iteraive user-driven fault localization. In *Hiafa Verification Conference*. Springer International Publishing, 82–98. doi:10.1007/978-3-319-49052-6_6

[9] Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. 2024. An Empirical Study on LLM-based Agents for Automated Bug Fixing. *ArXiv* abs/2411.10213 (2024). doi:10.48550/arXiv.2411.10213

[10] Steven P Reiss. 2019. Continuous Flow Analysis to Detect Security Problems. *arXiv preprint arXiv:1909.13683* (2019). doi:10.48550/arXiv.1909.13683

[11] Steven P. Reiss, Xuan Wei, Jiahao Yuan, and Qi Xin. 2025. ROSE: An IDE-Based Interactive Repair Framework for Debugging. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 112 (April 2025), 39 pages. doi:10.1145/3705306

[12] Steven P Reiss, Qi Xin, and Jeff Huang. 2018. SEEDE: simultaneous execution and editing in a development environment. In *Proceedings of 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 270–281. doi:10.1145/3238147.3238182

[13] Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, and Marc-Alexandre Côté. 2025. debug-gym: A Text-Based Environment for Interactive Debugging. arXiv:2503.21557 [cs.AI] doi:10.48550/arXiv.2503.21557

Received 2026-05-07; accepted 2026-06-19