ChatDBG: Augmenting Debugging with Large Language Models

KYLA H. LEVIN*, University of Massachusetts Amherst, USA
NICOLAS VAN KEMPEN*, University of Massachusetts Amherst, USA
EMERY D. BERGER†, University of Massachusetts Amherst, USA and Amazon Web Services, USA
STEPHEN N. FREUND, Williams College, USA

Debugging is a critical but challenging task for programmers. This paper proposes ChatDBG, an AI-powered debugging assistant. ChatDBG integrates large language models (LLMs) to significantly enhance the capabilities and user-friendliness of conventional debuggers. ChatDBG lets programmers engage in a collaborative dialogue with the debugger, allowing them to pose complex questions about program state, perform root cause analysis for crashes or assertion failures, and explore open-ended queries like '`why is x null`?'. To handle these queries, ChatDBG grants the LLM autonomy to *take the wheel*: it can act as an independent agent capable of querying and controlling the debugger to navigate through stacks and inspect program state. It then reports its findings and yields back control to the programmer. By leveraging the real-world knowledge embedded in LLMs, ChatDBG can diagnose issues identifiable only through the use of domain-specific reasoning. Our ChatDBG prototype integrates with standard debuggers including `LLDB` and `GDB` for native code and `Pdb` for Python. Our evaluation across a diverse set of code, including C/C++ code with known bugs and a suite of Python code including standalone scripts and Jupyter notebooks, demonstrates that ChatDBG can successfully analyze root causes, explain bugs, and generate accurate fixes for a wide range of real-world errors. For the Python programs, a single query led to an actionable bug fix 67% of the time; one additional follow-up query increased the success rate to 85%. ChatDBG has seen rapid uptake; it has already been downloaded more than 75,000 times.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Debugging, Artificial Intelligence, Software Engineering

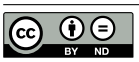
ACM Reference Format:
Kyla H. Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. 2025. ChatDBG: Augmenting Debugging with Large Language Models. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE085 (July 2025), 22 pages. https://doi.org/10.1145/3729355

1 Introduction

Debuggers help programmers identify and fix bugs by letting them investigate program state and navigate program execution. Debuggers for mainstream languages, including GDB [39] and LLDB [27] (for C, C++, and Rust), JDB (for Java), Pdb (for Python), and the Chrome or Firefox debuggers (for JavaScript), generally provide the same functionality. In particular, most debuggers

*Equal contribution.
†Work done at the University of Massachusetts Amherst.

Authors' Contact Information: Kyla H. Levin, University of Massachusetts Amherst, Amherst, MA, USA, khlevin@cs.umass.edu; Nicolas van Kempen, University of Massachusetts Amherst, Amherst, MA, USA, nvankempen@cs.umass.edu; Emery D. Berger, University of Massachusetts Amherst, Amherst, MA, USA and Amazon Web Services, Seattle, WA, USA, emery@cs.umass.edu; Stephen N. Freund, Williams College, Williamstown, MA, USA, freund@cs.williams.edu.



ACM 2994-970X/2025/7-ARTFSE085
https://doi.org/10.1145/3729355

support observing program execution via *tracing* and reporting when a program reaches a given line or function of source code; interrupting execution and returning control to the debugger when the program reaches a given line or function via *breakpoints*, when a particular condition is true via *conditional breakpoints*, or when a variable changes via *watchpoints* (a.k.a. *data breakpoints*); inspecting local variables, globals, heap objects, and *backtraces* of the call stack; and resuming program execution line-by-line (*single-step*) or at the granularity of function calls.

Debuggers can be helpful, but finding and fixing software defects remains a deeply challenging and time-consuming task [7, 20, 47]. Programmers must still reason about program behavior to ascertain what went wrong. They must formulate and test hypotheses about program execution, they must read and understand code they may have not written, and they must pore over potentially voluminous information. Such information includes lengthy executions, large amounts of program data, and many stack frames that potentially span multiple threads.

This paper introduces the **ChatDBG** AI-powered debugger assistant. ChatDBG integrates into and significantly extends the functionality of standard debuggers. ChatDBG builds on the insight that large language models (LLMs), such as OpenAI's GPT-4 [34], enable a debugger to leverage insights and intuition from the vast real-world knowledge embedded in LLMs. This knowledge enables ChatDBG to fix classes of issues that depend on logical thinking and domain-specific reasoning beyond the ability to write and debug programs. For example, Figure 2 illustrates the use of ChatDBG to debug a program by leveraging a knowledge of statistics that cannot be gleaned from the program itself.

A debugger integrated with ChatDBG continues to provide its full range of functionality but also lets programmers engage in debugging dialogs where they can ask high-level questions like '`why is x null here?`' or '`why isn't this value what I expected?`'. The question can be as simple as '`why?`' if a program has crashed or failed an assertion. To answer such queries, ChatDBG orchestrates a conversation with an LLM. A key feature of ChatDBG is that it grants autonomy to the LLM to "take the wheel" and act as an independent agent [10, 42] while answering the programmer's queries. Specifically, the LLM issues "function calls" [33] to run commands in the underlying debugger to investigate program state, execute code, or obtain source code. The results of those calls are sent back to the LLM to use in constructing its response. After answering a query, control is returned to the programmer, who may then enter additional commands or chat messages.

Our prototype of ChatDBG integrates into three widely used debuggers: GDB, LLDB, and Pdb. Our evaluation presents a range of case studies demonstrating that ChatDBG improves significantly on existing debuggers. On a suite of unpublished Python scripts and Jupyter notebooks written by undergraduate students, one or two queries is sufficient for ChatDBG to properly diagnose and fix defects 85% of the time, typically at a cost well under $0.20 USD. ChatDBG is also effective at identifying causes and providing fixes for a range of real-world bugs in C/C++ code.

This paper makes the following contributions:

- It introduces ChatDBG, an AI-powered debugger assistant that enables large language models to "take the wheel" and control the debugger via agentic reasoning.
- It describes the implementation of our ChatDBG prototype.
- It presents an evaluation of ChatDBG that demonstrates its significant advantages over existing debugger functionality.

Our evaluation shows that ChatDBG is broadly applicable to many domains and programming languages, and we expect it to be particularly useful for novice programmers, who often lack the experience to effectively use debuggers. ChatDBG is also useful for experienced programmers, who can augment debugging sessions with ChatDBG's reasoning capabilities in a conversational and interactive way.

Source code for bootstrap.py

```
1  from datascience import *
2  from ds101 import *
3
4  def make_marble_sample():
5      table = Table().read_table('marble-sample.csv')
6      return table.column('color')
7
8  def proportion_blue(sample):
9      return sample
10
11 def resampled_stats(observed_marbles, num_trials):
12     stats = bootstrap_statistic(observed_marbles,
13                                 proportion_blue,
14                                 num_trials)
15     assert len(stats) == num_trials
16     return stats
17
18 observed_marbles = make_marble_sample()
19 stats = resampled_stats(observed_marbles, 5)
20
21 assert np.isclose(np.mean(stats), 0.7)
```

Fig. 1. **An example program containing several bugs (§2).** It is supposed to create an array of marble colors, compute the proportions of blue marbles in resamples of that array, and assert that their mean is about 0.7, the proportion for the array.

2 Overview

This section illustrates ChatDBG's features and ability to assist in debugging the program in Figure 1. That program is a distillation of real errors encountered by students in an introductory data science lab. It creates an array `observed_marbles` representing the colors of marbles (red or blue) in a sample stored in a file. It then calls `bootstrap_statistic` to create same-sized resamples of that array. That function computes a statistic for each resample and returns an array of those statistics. In this case, the statistic is `proportion_blue`, the proportion of blue marbles. Given a sufficiently large number of trials, the mean of the resamples' statistics should be close to 0.7, the proportion of blue marbles in the original sample [6].

The program fails the assertion in `resampled_stats`, and Figure 2 illustrates a debugging session. To try to figure out what went wrong, the user issues the Pdb command `p num_trials` to view the value of that variable. Continuing debugging with existing tools would likely involve issuing additional commands, examining data files, source code, and examining library documentation. With ChatDBG, the user instead starts a dialog with the debugger, asking `why doesn't stats have 5 elements?` While constructing the answer (in blue), the LLM *takes the wheel* and directly issues debugger commands (yellow). These include standard Pdb commands and a ChatDBG-specific `info` command for accessing the source code and docstrings for any user-written code, as well as the docstrings for library code (which we assume is correct and not the root cause of errors).

ChatDBG identifies and corrects the root cause: `proportion_blue` incorrectly computes the desired statistic. When ChatDBG cannot identify the root cause, it suggests further debugging steps and control is returned to the user, who may continue the chat, issue further debugger commands, or both. Figure 3 illustrates this scenario, where a version of `bootstrap.py` with the corrected `proportion_blue` function fails the assertion on line 21.

```
$ chatdbg bootstrap.py
Traceback (most recent call last):
  File "/python3.11/site-packages/ipdb/__main__.py", ...
    pdb._run(stdlib_pdb._ScriptTarget(mainpyfile))
  File "/python3.11/pdb.py", line 1659, in _run
    self.run(target.code)
  File "/python3.11/bdb.py", line 600, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "./bootstrap.py", line 19, in <module>
    stats = resampled_stats(observed_marbles, 5)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "./bootstrap.py", line 15, in resampled_stats
    assert len(stats) == num_trials
           ^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
(ChatDBG) p num_trials
5
(ChatDBG) Why doesn't stats have 5 elements?
(ChatDBG) p len(stats)
150
(ChatDBG) info bootstrap_statistic
bootstrap_statistic(observed_sample, compute_statistic, num_trials)
    Creates num_trials resamples of the initial sample. Returns an array of the
    provided statistic for those samples.

    * observed_sample: the initial sample, as an array.

    * compute_statistic: a function that takes a sample as an array and returns
      the statistic for that sample.

    * num_trials: the number of bootstrap samples to create.

You MUST assume that `bootstrap_statistic` is specified and implemented correctly.
(ChatDBG) p proportion_blue
<function proportion_blue at 0x1050b00e0>
(ChatDBG) info proportion_blue
  8     def proportion_blue(sample):
  9         return sample
```

From the information we have, the `stats` variable in `resampled_stats` is supposed to have 5 elements, but it actually has 150 elements...

The Python debugger output shows a reference to a variable or function called `proportion_blue`, which is used as the `compute_statistic` parameter for `bootstrap_statistic`. Upon looking at the `proportion_blue` function's definition, it is found that the function simply returns the input `sample` without computing a statistical value. This means that for each of the `num_trials` (which is 5), `bootstrap_statistic` is adding the entire `sample` to the `stats` array without processing the sample to obtain a single statistic value...

Recommendation

To fix this issue, the `proportion_blue` function must be corrected to perform the intended computation of returning the proportion of blue marbles in the `sample`:

```
def proportion_blue(sample):
    return np.mean(sample == 'B')
```

Fig. 2. **A debugging session with ChatDBG (§2).** For brevity, we elide repetitive or unimportant parts. ChatDBG catches the assertion failure on line 15 and prompts the user to enter commands (**bold** and highlighted in gray). Commands issued by ChatDBG when it takes the wheel are highlighted in yellow, and its response is highlighted in blue.

```
$ chatdbg bootstrap.py
Traceback (most recent call last):
  File "./bootstrap.py", line 21, in <module>
    assert np.isclose(np.mean(stats), 0.7)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError
...
(ChatDBG) Why is the mean of stats not close to 0.7?
  (ChatDBG) p np.mean(stats)
  0.7466666666666667
  ...
  (ChatDBG) info resampled_stats
  ...
  (ChatDBG) info make_marble_sample
  ....
  (ChatDBG) info bootstrap_statistic
  ...
  ...
  Recommendation
  1. Check the CSV file marble-sample.csv to verify the proportion of blue to red marbles and ensure it is close to 70% blue
     marbles.
  2. Review the implementation of the proportion_blue function...
  3. Verify that the proportion_blue function ... is available in the current scope or imported correctly.
(ChatDBG) p proportion_blue(observed_marbles)
0.7
(ChatDBG) That value is correct.
  ...
  (ChatDBG) p stats
  array([0.66666667, 0.8       , 0.63333333,
         0.76666667, 0.86666667])
  At this point, the most straightforward explanation for the error is that the number of trials is not enough to accurately reflect
  the expected mean due to random sampling effects. Increasing the number of trials would likely result in an average statistic
  that is closer to the expected value of 0.7, following the Law of Large Numbers.
  Recommendation
  Increase the number of bootstrap trials significantly (e.g., from 5 to something like 1000 or more) to reduce the variability of the
  mean...
```

Fig. 3. **A debugging session demonstrating how ChatDBG incorporates real-world knowledge (§2).** After correcting `proportion_blue` in Figure 1, the program fails on line 21 because the mean proportion of blue marbles in the resamples is not the expected value. ChatDBG identifies high variance resulting from the small number of trials as the root cause.

The user asks why the mean of `stats` is not close to 0.7, and ChatDBG's initial response suggests examining whether 0.7 is the appropriate expected value. To verify this, the user then computes the proportion of blue marbles with a debugger command and tells ChatDBG that 0.7 is indeed the correct value. In its follow-up, ChatDBG points to the low number of trials (five) as the issue. The LLM drew this correct conclusion without seeing any discussion of trial size or variance in any program state, code, or documentation encountered during the chat. A powerful aspect of ChatDBG is its ability to exploit real-world knowledge in its analyses (here, the fact that bootstrapping depends on large numbers of resamples) without specific instruction or user intervention.

Table 1. **Debugger features and their dates of introduction (§3).** Most key features have been around for decades. By integrating into modern debuggers (GDB, LLDB, and Pdb), ChatDBG inherits all of their features while significantly extending them with functionality to explain bugs and their root causes, propose fixes, and answer arbitrary natural-language queries over program state. (An asterisk or *year* in italics means the feature is limited in functionality, performance, or depends on specific hardware support.)

System and Date	Single Step	Stack Navigation	Breakpoints (BPs)	Conditional BPs	Source Level	Trace	Display State	Eval. Code	Watchpoints	Explain Bugs	Propose Fixes	Open Queries
DDT [19], 1961	✓	✓	✓									
EXDAMS [3], 1969	✓	✓	✓						✓			
Mesa [43], 1979	✓	✓	✓	✓*	✓	✓	✓	✓				
Dbx [26], 1981	✓	✓	✓	1990	✓	✓	✓	✓	✓			
GDB [39], 1986	✓	✓	✓	✓	✓	✓	✓	✓	1991			
Pdb, 1992	✓	✓	✓	✓	✓	✓	✓	✓	✓			
LLDB, 2010	✓	✓	✓	✓	✓	✓	✓	✓	✓			
ChatDBG, 2023	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

3 Related Work

Table 1 presents an overview of previous interactive debuggers, together with their features. The first interactive debugger, DDT, introduced breakpoints, single-stepping, and stack navigation in 1961 [19]. By 1979, the Mesa debugger had most key features of modern debuggers, including source-level debugging, conditional breakpoints, tracing, and the ability to display run-time state and evaluate code [43]. Arbitrary conditional breakpoints date back at least to 1990 with Dbx [26]. Watchpoints were introduced by 1991 and have been in GDB since version 3.93.

In other work, Ko and Myers present Whyline, an interactive, trace-based debugger that lets programmers select from a range of queries and identifies (via static and dynamic analysis) a timeline that answers the query [18]. Programmers can only select from those queries presented by Whyline as options. In contrast, ChatDBG permits programmers to pose arbitrary queries that it answers via a dialog with an LLM. Whyline's use of traces gives it the ability to answer questions that might not be straightforward to answer with the current program state but limits its applicability to relatively short-lived executions.

The goal of *program slicing*, introduced by Weiser in 1981 [41], is to produce a shorter version of a program limited to the source code that could have led to an error. Program slicing has been extensively studied; Weiser's paper has been cited over 5,000 times to date. As Section 4.7 describes, ChatDBG performs backwards slicing to collect code spread across code cells to facilitate debugging of Jupyter notebooks.

Fault localization seeks to identify the likely location of a defect's root cause. Several prior studies have investigated the use of machine learning and LLMs for fault localization. Some of the studied techniques apply machine learning to source code features, coverage data, or other static code features to predict faulty lines of code, but they do not utilize dynamic state and run-time information. DeepFL [24], Grace [28] and DeepRL4FL [25] are examples of such systems. Similarly, LLMAO [44] employs LLMs to provide suspiciousness scores for each line of code in a given program, but only provides access to the source code. AutoFL [16] also utilizes an LLM and enables it to statically retrieve source code and coverage information about the program via function

calls. However, the system requires a failing test case as input and does not employ run-time state information.

ChatDBG improves upon these systems by providing an LLM with access to run-time program state and the ability to take control of the underlying debugger. Both features enhance the LLM's ability to provide more accurate and informative feedback to the user. We also note that other fault localization techniques can be used in tandem with ChatDBG to improve results, as suggested by Section 5.1's utilization of backwards slicing to identify code relevant to a bug in Python notebooks and Section 5.2's utilization of AddressSanitizer [37] to provide a better starting point for diagnosing and fixing memory errors in native code.

Automated program repair is another active area of software engineering research [9]. Systems for automatic program repair attempt to generate source-level program patches that prevent a program from failing. ChatDBG performs best-effort automated program repair by requesting that the LLM propose code fixes as part of its response, ultimately letting the programmer drive code changes using these suggestions. Previous research has shown that automated program repair hints can provide significant help in the debugging process and suggests that the benefits of correct advice outweigh the risk of deceptive ones [8].

3.1 Concurrent Work

Several approaches developed concurrently with ChatDBG have also integrated LLMs into automatic program repair or fault localization techniques to enhance the debugging process. Robin [2] is a chat-based debugging assistant designed to help users diagnose errors more quickly. Both Robin and ChatDBG provide a limited program context to the LLM at the beginning of a conversation. However, Robin has no direct access to any additional context about the program and execution state; the user must manually retrieve and provide these items. Robin's functionality is therefore roughly equivalent to the **Enriched Stack** configuration of ChatDBG, detailed in Section 5.1. As Figure 6 shows, ChatDBG achieves a nearly two-fold increase improvement in diagnosing errors versus the **Enriched Stack** configuration. ChatDBG's effectiveness generally increases further with targeted questions and follow-up discussions with the user.

AutoCodeRover [48] and SWE-agent [45] are complementary approaches that focus on fault localization and automatic repair, relying exclusively on issue descriptions and source code. ChatDBG additionally leverages run-time information to identify root causes and propose fixes. Section 5 demonstrates the strength of this approach over relying solely on static information. Both AutoCodeRover and SWE-agent perform an evaluation using SWE-bench [15], which was created to evaluate the efficacy of such static tools; unfortunately, this benchmark suite is not applicable to ChatDBG due to its extensive usage of run-time information.

RepairAgent [4] and AutoSD [17] are tools that employ LLMs in specific workflows that mimic standard debugging strategies in an attempt to repair pre-identified bugs. While successful in some settings, both tools rely on the user providing a failing test case and the precise location of the bug. By contrast, ChatDBG does not require this information. ChatDBG also enables a more flexible workflow that permits collaboration with the developer in addition to seamless integration into the standard debugging process.

4 Implementation

4.1 Using ChatDBG: Preliminaries

ChatDBG integrates with existing debuggers as either a plug-in or a direct extension. Our primary focus to date has been an extension to Pdb, which supports both non-interactive Python scripts

and interactive sessions in IPython or Jupyter notebooks, and a plug-in for LLDB to support C/C++ code. A subset of features has been ported to GDB and WinDBG.

Configuration for Python is minimal and limited to the installation of the `chatdbg` package with the standard package installer, plus one optional shell script command to add it as an extension to IPython. ChatDBG extends either the standard `pdb.Pdb` debugger or IPython's implementation of Pdb, depending on how it is run. Configuration for LLDB and other C/C++ debuggers is similarly straightforward. LLDB can be installed through standard package managers if it is not already present, and the ChatDBG plug-in is installed via a single shell command. Since ChatDBG leverages OpenAI's LLMs, the user must also set an environment variable to a valid OpenAI API key within their system's configuration settings.

4.2 Debugging a Target Program

For Python, debugging with ChatDBG begins by running `chatdbg` on the target program. No special preparation of the target is needed; Python's managed run time ensures that debugging information and source code is always available. Debugging is supported in IPython interactive sessions or Jupyter notebooks via the standard command-line flag `--pdb` or the Jupyter magic command `%pdb`, respectively. Control drops into the debugger when an exception occurs.

For C and C++, debugging begins by running `lldb` on the target program. The target program must be an unstripped executable generated with the `-g` compiler flag, which ensures the availability of DWARF debug information that describes the memory layout and maps the program's machine code back to the original source code. That information is essential for the effective debugging of unmanaged code.

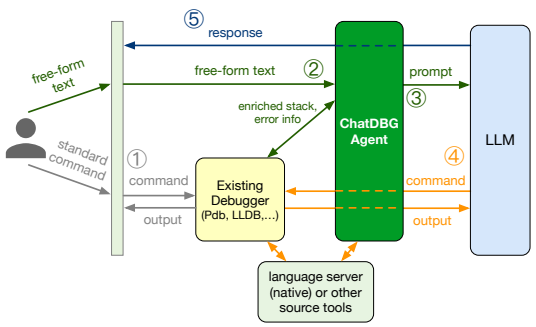
ChatDBG also handles native code generated for other languages but may require additional steps. For example, to debug a Rust target program, the `Cargo.toml` file must list ChatDBG as a dependency and the `main` function must be annotated with `#[chatdbg::main]` to ensure that error messages are visible to ChatDBG through a log file.

4.3 ChatDBG Architecture Overview

ChatDBG orchestrates communication between the user, the debugger, and the LLM, as shown in the architecture diagram in Figure 4. The operations in the command loop pseudocode map naturally onto debugger APIs and onto LLM APIs supporting completion and function calls [33]. ChatDBG currently utilizes OpenAI's API [32] and GPT-4 models. We provide a brief overview of the ChatDBG architecture and then elaborate on the most salient technical innovations below.

① ChatDBG dispatches standard commands, such as `p num_trials` in Figure 2, directly to the underlying debugger (lines 3-7). It also preserves those commands and their output in the history variable for later communication to the LLM. ② Any other text entered by the user, such as '`why doesn't stats have 5 elements?`', is directed to ChatDBG, which creates a prompt to send to the LLM. If this is the start of a chat, ChatDBG bundles basic instructions, information from the debugger about the current stack and error, program inputs, history of user commands, and the text together in an *initial prompt* (lines 9-12). Otherwise, ChatDBG bundles only the history since the last chat step and text (line 14). The MakePrompt function concatenates the prompt components into a string, respecting any length limits set by the LLM by selectively truncating parts as needed.

③ ChatDBG then sends the prompt to the LLM and processes the response stream, which includes both ④ requests to run debugger commands (lines 19-22) and ⑤ prose for the user (line 23). In Figure 2, ChatDBG runs four debugging commands, including one to print the length of the `stats` array, via this mechanism as the LLM constructs its response. ChatDBG echoes those commands and their outputs to the user. Once the full response has been processed, ChatDBG returns control to the user. As Section 4.5 discusses, ChatDBG augments the underlying debuggers



① Standard commands are handled by the existing debugger.
② ChatDBG converts free-form text into a suitable prompt.
③ ChatDBG sends the prompt.
④ The LLM takes the wheel and directly issues commands to the underlying debugger. This step may involve consulting other tools, such as a language server for native code.
⑤ The LLM responds to the prompt.

```
1:  history = “”
2:  loop
3:    line = Input()
4:    if IsDebuggerCommand(line) then
5:      output = DoCommand(line)
6:      Print(output)
7:      history = history + (line + “→” + output)
8:    else
9:      if not ChatInProgress() then
10:       prompt = MakePrompt(Instructions(),
11:                           EnrichedStack(), Inputs(),
12:                           Error(), history, line)
13:     else
14:       prompt = MakePrompt(history, line)
15:     Send(prompt)
16:     history = “”
17:     while ResponsePending() do
18:       match Receive()
19:         case Debug(cmd) ⇒
20:           output = DoCommand(cmd)
21:           Print (cmd + “→” + output)
22:           Send(output)
23:         case Message(text) ⇒ Print(text)
```

Fig. 4. **ChatDBG architecture and command processing algorithm (§4.3).**

with specialized commands for the LLM to use when taking the wheel. For example, the ChatDBG variant for native code installs debugger commands that utilize the `clangd` language server [5, 30] to retrieve source code corresponding to symbol definitions.

4.4 Initial Prompts and Enriched Stack Traces

In addition to including the user's text, the initial prompt conveys instructions to LLM and the context surrounding the error. We illustrate the components of the prompt in this section, using the initial prompt in Figure 5 that was generated for the first query in Figure 2 as a running example.

Instructions. The instructions at the top of the prompt ask the LLM to answer questions about the root cause of the error, to focus on user code, to explain values stored in variables, and to end each response with either a fix or suggestions for further debugging steps. The last item ensures a relatively consistent structure for answers that facilitates reading them and evaluating their quality. Paragraphs 2-4 of the instructions are the *take the wheel* prompt described in Section 4.5.

Enriched stack trace. ChatDBG's success at identifying and fixing errors relies critically on providing the LLM with sufficient details to reveal the cause of the error. A key source of that information is the run-time stack. Debuggers provide a way for the user to view the stack trace but often only show function names, source file locations, and possibly a couple lines of code for each stack frame. ChatDBG provides a more detailed *enriched stack trace* to the LLM. That stack trace includes the types and values of variables for each frame, as well as a larger window of at least 10 lines of code. Enriched stack traces also elide frames corresponding to library code to better focus the LLM on user-written code, which ChatDBG assumes to be the most likely cause of errors.

In Python, ChatDBG leverages Pdb's internal data structures to build enriched stack traces. When converting values to suitable string representations, ChatDBG must balance utility with the size of the string produced. For objects, ChatDBG calls the object's `__repr__` method if an

Instructions:
You are a debugging assistant. You will be given a Python stack trace for an error and answer questions related to the root cause of the error.

Call the debug function to run Pdb debugger commands on the stopped program. You may call the debug function to run the following commands: `bt`, `up`, `down`, `p expression`, `list`. Call debug to print any variable value or expression that you believe may contribute to the error.

Call the `info` function to get the documentation and source code for any variable, function, package, class, method reference, field reference, or dotted reference visible in the current frame. Examples include: n, e.n where e is an expression, and t.n where t is a type. Unless it is from a common, widely-used library, you MUST call `info` exactly once on any symbol that is referenced in code leading up to the error.

Call the provided functions as many times as you would like.

The root cause of any error is likely due to a problem in the source code from the user. Explain why each variable contributing to the error has been set to the value that it has. Continue with your explanations until you reach the root cause of the error. Your answer may be as long as necessary.

End your answer with a section titled “Recommendation” that contains one of:
– a fix if you have identified the root cause
– a numbered list of 1-3 suggestions for how to continue debugging if you have not

Enriched Stack Trace:
The program has this stack trace:

```
[... skipping 4 hidden frame(s)]

./bootstrap.py(19)<module>()
     15
     16     assert len(stats) == num_trials
     17
     18 observed_marbles = make_marble_bag()
---> 19 resampled_stats(observed_marbles, 5)

   Global variables:
     observed_marbles: ndarray = array(['R', 'R', 'R', ..., 'B',
                                       'B', 'B'], dtype='<U1')

> ./bootstrap.py(16)resampled_stats()
     14                                    num_trials)
     15
---> 16     assert len(stats) == num_trials
     17
     18 observed_marbles = make_marble_bag()

   Variables in this frame:
     num_trials: int = 5
     observed_marbles: ndarray = array(['R', 'R', 'R', ...,
                                       'B', 'B', 'B'], dtype='<U1')
     stats: ndarray = array(['B', 'R', 'B', ..., 'R', 'B',
                             'R'], dtype='<U32')
```

Error:
The program encountered the following error:

```
AssertionError
```

The code `assert len(stats) == num_trials` is correct and must not be changed.

History:
This is the history of some pdb commands I ran and the results:

```
(ChatDBG) p num_trials
5
```

User Text:
Why doesn’t stats have 5 elements?

Fig. 5. **The initial prompt for the debugging session in Figure 2 (§4.4).** For brevity, the enriched stack includes only five lines of source in each frame, rather than the default of 10.

appropriate (non-default) version exists. Otherwise, it iterates over the object's fields and recursively converts their values to strings. Similarly, ChatDBG recursively converts the values stored in aggregate structures like lists, arrays, and dictionaries to strings, but limits the number of elements shown to a small, fixed number. The rest of the elements are abbreviated with an ellipsis (...). This recursive conversion of values to strings is limited to a depth of three, at which point any remaining values are again abbreviated with ellipses. This strategy balances the need to provide the LLM with sufficient information to diagnose the error with the need to avoid overwhelming it with too much information. In cases with the elided details are important, the LLM can request them via the *take the wheel* mechanism.

ChatDBG follows roughly the same approach in LLDB, utilizing the static types embedded in the DWARF debugging information to decode the stack. In addition, any pointers are dereferenced to show the values being referred to as well; null pointers and illegal dereferences are dropped.

Inputs. The initial prompt also includes the target's command line arguments and standard input when that information is available from the underlying debugger. These are empty and elided in Figure 5.

Error. A description of the error causing execution to stop is extracted from the underlying debugger. When the error is due to an assertion failure, ChatDBG instructs the LLM to assume that the assertion is valid as written so that it will look beyond the assertion for the real problem.

History. The initial prompt also includes the history of commands already issued by the user, as well as their outputs. This builds a more complete context surrounding the user's query.

4.5 Taking the Wheel

ChatDBG supports *take the wheel* debugging via the function call capabilities in OpenAI's API and most recent models [33]. This agentic approach [10, 42] lets clients register callback functions with the LLM for obtaining additional information while constructing a response. The LLM calls these functions by sending special messages to the client as part of its response stream. The client receives those messages, computes the requested results, and sends them back to the LLM. The initial prompt describes how to use the available functions.

For example, ChatDBG registers a `debug(command)` function for running a command in the underlying debugger. The LLM calls `debug("p len(stats)")` through this mechanism in the session from Figure 2. ChatDBG then runs Pdb's command processing routine, `onecmd("p len(stats)")`, and captures the output to and send back. ChatDBG similarly uses the `SBCommandInterpreter.HandleCommand` routine in LLVM. In both cases, the command and output are printed so the user can see these steps.

The LLM has sufficient background knowledge on debuggers and requires *no additional training* to navigate up/down the stack, inspect variables and heap data, evaluate expressions, and perform other typical debugger operations.

Supporting agentic reasoning over run-time program state via function calls is a key technical innovation of ChatDBG. Without this capability, there would be no effective way to provide the LLM with a detailed view of relevant program state. A common alternative technique for handling large amounts of task-specific data in LLMs is to employ a retrieval augmented generation (RAG) model [23], which collects and stores the data in a vectorized database that is then made available to the model for retrieval. However, that approach seems less useful in this context, as program state information will be distinct for each debugging session and not easily vectorized.

Table 2. **ChatDBG command extensions (§4.6).** ChatDBG extends the underlying debuggers with several new commands to help the LLM navigate through and understand the target's code. ChatDBG provides access to them via the LLM's function call API.

Command	Debugger	Output
`info symbol`	Pdb	The source code and/or docstring for a `symbol` referring to any function, method, field, class, or package.
`slice symbol`	Pdb	The source code in the backwards slice of the global `symbol`. Interactive IPython/Notebook sessions only.
`code loc`	LLDB	The source code surrounding `loc`, where `loc` has the form `filename:lineno`.
`definition loc symbol`	LLDB	The declaration for the first occurrence of `symbol` at `loc`, where `loc` has the form `filename:lineno`.

4.6 Navigating the Code

While the LLM can often leverage pre-existing background knowledge of common Python and C/C++ standard libraries, it will likely have limited-to-no knowledge of any user-defined code or third-party library functions. Trying to include all possibly-relevant source code in the initial prompt would be infeasible and would prevent ChatDBG from scaling to larger codebases. Instead, ChatDBG extends the underlying debuggers with several new commands designed to help the LLM navigate through and understand the target's code. These commands are available to the LLM via function calls and listed in Table 2.

ChatDBG augments Pdb with the `info` command, which prints the docstring for any function, class, field, method, or package. It additionally prints the source code for any user-defined function. The `info` requests in Figure 2 demonstrate these two cases for `proportion_blue` and `bootstrap_statistic`, respectively. The command is implemented via the standard `inspect` and `pydoc` Python libraries.

The `info` command is not directly reproducible for unmanaged code in LLVM because there is no comparable existing debugger support for retrieving the source or documentation for a symbol. Instead, ChatDBG adds two other debugging commands to LLDB. The first, `code`, prints the code surrounding a source location described by a filename and line number, as in `code polymorph.c:118`. The second command, `definition`, prints the location and source code for the definition corresponding to the first occurrence of a symbol on a given line of code. For example, `definition polymorph.c:118 target` prints the location and source for the declaration of `target` corresponding to its use on that line. The `definition` implementation leverages the `clangd` language server, which supports source code queries via JSON-RPC and Microsoft's Language Server Protocol [30].

4.7 Slices for Interactive Python

ChatDBG supports debugging interactive IPython sessions and Jupyter notebooks. Interactive sessions lead to many individual code cells that are each evaluated separately. Cells may be evaluated out-of-order, override definitions from earlier cells, and communicate values to other cells through top-level global variables. Others have noted the challenges of reasoning about program behavior in this context [11, 38]. ChatDBG provides an additional `slice` debugging command to facilitate that reasoning. The `slice` command computes the backwards slice for any variable used in the current cell that was defined in previously-executed cells. It returns the code for cells in that slice. Suppose the code from `bootstrap.py` in Figure 1 were written in four notebook cells as shown below:

```
In[2]: def make_marble_sample(): ...

In[3]: def proportion_blue(sample): ...

In[4]: def resampled_stats(observed_marbles, num_trials):
           stats = bootstrap_statistic(observed_marbles,
                                       proportion_blue,
                                       num_trials)
           assert len(stats) == num_trials
           return stats

In[5]: observed_marbles = make_marble_sample()
       stats = resampled_stats(observed_marbles, 5)
```

After evaluating these cells, `slice(observed_samples)` returns the source for the cells labeled `In[2]` and `In[5]`, and `slice(stats)` returns the source for all four cells. ChatDBG uses `ipyflow` to compute slices [14, 38].

4.8 Security and Risks

It is possible for the LLM to issue debugging commands containing arbitrary code through the debug function call provided by ChatDBG. That code could, for example, delete files or execute other malicious actions on the client. ChatDBG mitigates this risk by sanitizing LLM-generated debugging commands before running them. For Python, the sanitizer ensures that any functions called in LLM-provided commands belong to a user-configurable whitelist. For native code, code provenance is harder to track and languages are more permissive, so the sanitizer rejects any commands calling functions. ChatDBG supports an `--unsafe` flag to disable sanitizing when the client system is running in an isolated environment that obviates the need for such protections.

It is also possible for the LLM to hallucinate and respond with incorrect or misleading diagnoses and fixes. ChatDBG mitigates this risk by not directly applying proposed code fixes or suggestions to the target code. Instead, ChatDBG presents them to the user, who may then vet and judge the quality of the LLM's responses and decide whether or not to follow suggested changes.

5 Evaluation

We demonstrate ChatDBG's capacity to identify the root cause of defects and provide fixes in two contexts: bugs in relatively small Python programs written by students and bugs in large C/C++ programs. The former have well-defined expected behavior that enables us to thoroughly and systematically assess ChatDBG. The latter demonstrates its effectiveness on unmanaged code when unusual corner cases trigger crashes. Our evaluation addresses the following research questions: **RQ1:** Is ChatDBG effective at diagnosing and fixing bugs in Python? **RQ2:** Which components of ChatDBG contribute to its effectiveness? **RQ3:** Is ChatDBG effective at diagnosing and fixing bugs in unmanaged code (C/C++)?

5.1 Python

We applied ChatDBG to all of the bugs in a collection of student labs from two introductory computer science courses; see Table 3. Bugs c1–c8 are in non-interactive scripts from a programming class that perform various file reading and text processing tasks. Bugs s1–s14 are in Jupyter notebooks [40] from a data science class that manipulate, visualize, and compute over arrays and tables. Some bugs were apparent to the programs' authors. Others were identified during autograding.

Table 3. **Python programs exhibiting a variety of common errors (§5.1).** Programs c1–c8 are command line scripts, and programs s1–s14 are Jupyter notebooks, which utilize two non-standard libraries consisting of 3,000 lines of code. Semantic errors reflect failed tests expressed as assertions. Crashes reflect unexpected termination due to any other type of error.

Name	LoC	Type	Reported Exception	Root Cause
c1	48	semantic	Assertion Error	Off-by-one error in an h-index computation
c2	81	crash	Name Error	Parameter not referenced properly
c3	64	crash	Value Error	Error in CSV column label leads to improper data parsing
c4	89	crash	Index Error	A class's `__str__` fails if an object's internal list is empty
c5	29	crash	Index Error	Missing one of two base cases in a recursive function
c6	72	crash	Name Error	Multiple errors related to building list of user-defined objects
c7	71	semantic	Assertion Error	Failure to convert input to lower case before processing
c8	72	semantic	Assertion Error	Missing test for lowercase words
s1	123	semantic	Assertion Error	Incorrect drop and rename operations leading to bad data
s2	124	semantic	Assertion Error	Incorrect max operation on a table
s3	124	semantic	Assertion Error	Incorrect aggregation function in pivot operation
s4	124	semantic	Assertion Error	Incorrect aggregation function in group operation
s5	162	semantic	Assertion Error	Hardcoded table data in wrong order
s6	162	crash	Name Error	Typo in variable reference
s7	45	semantic	Assertion Error	Function confuses parameter and global variable
s8	49	semantic	Assertion Error	Wrong percentile used in confidence interval construction
s9	112	semantic	Assertion Error	Wrong percentile used in confidence interval construction
s10	118	semantic	Assertion Error	Loops doesn't append to array correctly
s11	181	crash	Value Error	Creates a sample without replacement larger than the input
s12	127	crash	Value Error	Incorrect label when accessing column value for table row
s13	127	crash	Value Error	Pivot uses wrong columns for row/columns in new table
s14	65	crash	Index Error	Incorrect computation of random sample under null hypothesis

Unlike many existing bug benchmarks for Python, these programs are unpublished and thus not in the language model's training data. In addition, the programs have clear correctness criteria that lead to objective effectiveness metrics in our experiments. The bugs are representative of common mistakes because they were introduced by real humans, rather than synthetically generated. They range from scoping issues, algorithmic errors, and misuse of library functions to subtle misunderstanding of domain knowledge. They include both semantic errors leading to failed tests and crashing errors that terminate execution abruptly. Further, they reflect two important, widely-used modalities for Python programming: non-interactive scripts and interactive computational notebooks. ChatDBG supports debugging in both settings.

Programs were prepared by removing them from their automatic grading harness and replacing failed unit tests with `assert` statements that generate exceptions. We focus our evaluation on Python and perform an ablation study by progressively enabling ChatDBG features. We ran each program ten times under the five configurations in Table 4: **Default Stack** includes standard stack traces, as generated by `ipdb` [13], with 5 lines of code per frame in the initial prompt, but it does not support the LLM taking the wheel. **Enriched Stack** generates enriched stacks with ten lines of code per frame, and **+Take the Wheel** additionally permits ChatDBG to run debugger commands. These three configurations all use `why?` as the initial user text. **+Targeted Question** asks a question specific to the failure. For semantic errors, which validate the values stored in variables, these questions describe what those values should be or what they intend to represent. For crashes, the questions relate the crash to expected behavior, as in the following; we designed our questions to be "neutral" and not hint at the root cause.

Table 4. **Configurations used in the Python experiments (§5.1).**

Configuration	Stack Trace	Take the Wheel	Initial Prompt	Ask a Follow-up
Default Stack	standard		`why?`	
Enriched Stack	enriched		`why?`	
+Take the Wheel	enriched	✓	`why?`	
+Targeted Question	enriched	✓	*specialized*	
+Dialog	enriched	✓	*specialized*	✓

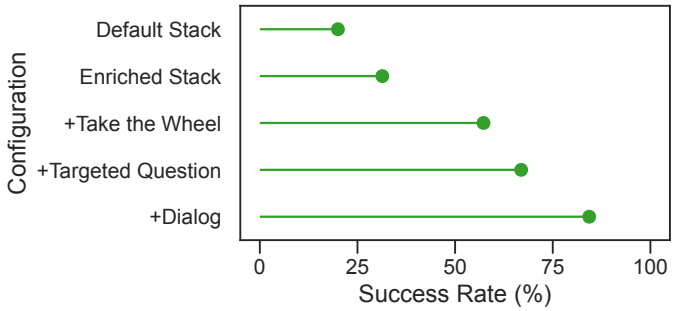


Fig. 6. **Overall ChatDBG success rate for each configuration (§5.1).** ChatDBG innovations and user-provided context gradually increase effectiveness.

c3 (Crash) Why am I not reading the CSV file correctly?
s11 (Crash) Why am I not able to sample 100 rows?
c1 (Semantic) Why am I not getting 3?
s1 (Semantic) `bill_length_mean_by_species` should be a table of the mean bill lengths of each species in our data set. Why isn't it?

The final **+Dialog** configuration is the same as **+Targeted Question** but extends the chat with a second query. All trials use the same follow up: *Continue to explain your reasoning and give me a fix to make it work as I describe.* Context-specific follow-ups work better, but we opted for consistency.

ChatDBG used the `gpt-4-1106-preview` model for these experiments. Under **+Targeted Question**, the first prompt and response led to, on average, a chat of about 10,000 tokens (7,500 words), a cost of about $0.12 USD under OpenAI's current pricing model [35], and a completion time of about 25 seconds. Subsequent steps in extended debugging dialogs incurred comparable costs. Time was highly variable and dominated by the performance of OpenAI's service. These characteristics will be different for other platforms and models and, given current trends, we expect significant reductions in both time and cost as models improve.

RQ1: Is ChatDBG effective at diagnosing and fixing bugs in Python?

Each response was manually examined and deemed a success if it included an accurate explanation of the error and an actionable fix. That fix could be either code or a prose description in which all necessary details were made explicit. To avoid bias in this assessment, explicit criteria for each program was determined prior to examining the responses.

Figure 6 shows the success rate under each configuration. The simplest configuration, **Default Stack**, provides functionality roughly equivalent to the user copying and pasting the program stack trace and basic error information into an LLM chat window and requesting a fix. We use this

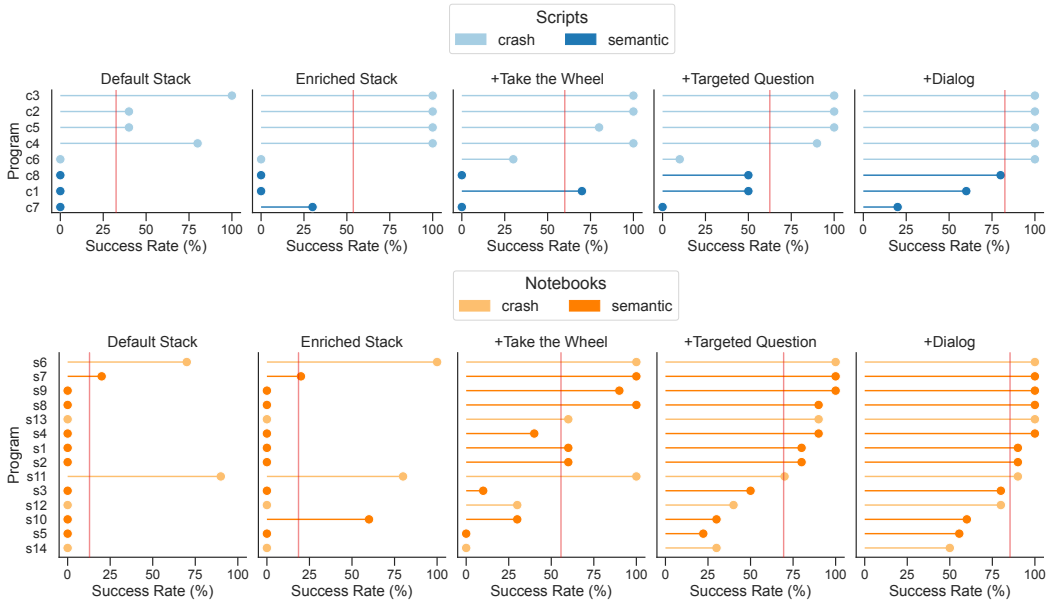


Fig. 7. **Success rate for ChatDBG for each program and configuration (§5.1).** Vertical lines show the mean.

configuration as a baseline for evaluating the impact of ChatDBG's more advanced configurations. With all features enabled, ChatDBG was successful at identifying and fixing bugs in well over half of the trials. Any time or energy expended by the user manually debugging those cases would be all but eliminated by using ChatDBG.

RQ1 Summary: Even with just the simple question `why`?, ChatDBG was successful 57% of the time. With questions specialized to the target's particular error, that number jumps to 67%, and with an additional dialog step ChatDBG succeeded in identifying and fixing the defect in 85% of the trials.

RQ2: Which components of ChatDBG contribute to its effectiveness?

Figure 7 presents the success rates for each program under each configuration. The **Enriched Stack** plots demonstrate that enriched stacks provide some benefit, particularly for crashes in which the stack contains sufficient information to diagnose the problem, but they alone do not provide much improvement for many semantic errors in which the relevant computation steps complete before failure. However, enriched stacks coupled with letting the LLM take the wheel led to significant improvement in the success rate for both crashing and semantic bugs, as shown in the **+Take the Wheel** plots.

Using the **+Take the Wheel** feature, the LLM issues from 0 to 12 debugging commands per run, most commonly calling the `info`, `slice` (for notebooks), and `p` (print) debugging commands. While all of these commands provide useful information about execution state and code, the `slice` command was critically important for notebooks. Without it, success rates rarely improved when the LLM took the wheel.

The **+Targeted Question** configuration demonstrates the impact of providing even the most modest details about expected behavior in queries. When the LLM is asked to continue its reasoning in **+Dialog**, ChatDBG's success rate improves despite the follow-up prompt providing no feedback on the contents or quality of the first response. This phenom indicates that constraints on the underlying LLM's response lengths may prevent it from conducting the amount of reasoning necessary to develop a fix in a single step. The success rates for **+Targeted Question** and **+Dialog** demonstrate the importance of continued dialogs and user input. We expect those features to be even more important to ChatDBG's success when diagnosing bugs in more complex programs.

The LLM also demonstrated its background knowledge with the responses including, for example, details of Python idioms and libraries, the definition of h-index [12], and the implementation and limitations of various statistical techniques.

Failures were generally due to the LLM not always recognizing or discovering key aspects of a program's behavior. We observe that in some cases, enabling additional features in ChatDBG decreases its success rate. We attribute this result to the fact that longer and more complex prompts can occasionally degrade the effectiveness of LLMs [22]. In general, the further the distance between the root cause of a bug and observable effect, the more challenging it was for ChatDBG (and people [47, p. 243]) to find it. In some cases, it was on the right track but did not converge on an actionable fix. In others, it suggested changes that would introduce other bugs. It also occasionally made mistakes, such as conflating proportions and percentages or failing to handle unusual corner cases. All of these could be mitigated by feedback from the user in subsequent follow-ups.

RQ2 Summary: While all features of ChatDBG contribute to its success, the technical innovations enabling it to take the wheel are critical. The most sophisticated configurations show that user-provided contextual information about behavior and engaging in multi-step dialogs are particularly good ways to improve its effectiveness.

5.2 C and C++

Programs in unmanaged languages such as C and C++ are vulnerable to memory safety errors. These memory errors can also hinder the debugging process: the crash may not occur immediately at the memory violation but instead much later on, and the crash may cause corruption of the stack and/or heap, making it challenging to recover any useful information.

Table 5 summarizes the programs extracted from the BugBench [29] and BugsC++ [1] suites used to evaluate ChatDBG's effectiveness at debugging unmanaged code. Programs used in this evaluation are all real-world applications with concrete known bugs. The four BugBench programs were selected as the only ones we could retrieve, build, and reproduce on our system. The BugsC++ suite does not include the original crash-causing inputs. However, it provides links to the original bug report, CVE identifier, and/or exploit-fixing patch, from which we manually retrieve crash reproduction information. We randomly selected and reproduced four bugs from the "memory error" category.

Some of the programs studied do not crash at run time. We employed AddressSanitizer [37] to force a crash at the moment a memory violation occurs to trigger those defects. AddressSanitizer is already capable of reporting some information about the crash when it happens. However, this information is often very dense, and typically points at the symptom of the bug, not its root cause. We did not include that information in the initial prompt.

RQ3: Is ChatDBG effective at diagnosing and fixing bugs in unmanaged code (C/C++)?

We ran our C/C++ experiments on an x86 Ubuntu 22.04 server. We used Clang and LLDB 17 to compile and debug, using flags `-g -Og -fno-omit-frame-pointer`. ChatDBG used OpenAI's

Table 5. **Bugs in unmanaged C/C++ code, and our criteria for fixing the proximate cause or the root cause of each (§5.2).**

Program	LoC	Error Type	Root Cause	Fix	
				Proximate Cause	Root Cause
BC [29]	17.0k	Buffer overflow	Input from data file printed to a fixed-size buffer	Truncate on copy	Use dynamic size
GZIP [29]	8.2k	Buffer overflow	Command line argument unsafely copied to a fixed-size buffer	Truncate on copy	Check size & warn/exit
NCOM [29]	1.9k	Buffer overflow	Command line argument unsafely copied to a fixed-size buffer	Truncate on copy	Check size & warn/exit
PEG [1]	14.7k	Null dereference	Invalid input produces corrupted data structure	Check if not null	Warn/exit
POLY [29]	0.7k	Buffer overflow	Command line argument is unsafely copied to a fixed-size buffer	Truncate on copy	Check size & warn/exit
TIFF [1]	58.9k	Division by zero	Combination of command line options leads to a division by zero	Override option values	Warn/exit when invalid
YAML1 [1]	8.7k	Stack overflow	Long sequences of { in the input leads to deep recursion	Use iterative method	Guard recursion depth
YAML2 [1]	8.7k	Assertion failure	Specific input causes a peek request for non-existent "next" token	Replace assert	Check before peeking

`gpt-4-1106-preview` model. Each program was run ten times using queries of the form `I am debugging cpp-peglib. Provide the root cause of this crash`, for PEG, followed by a request to include code in the response. Average time (27 seconds) and cost ($0.06 USD) were comparable to Python.

We manually examined each response to determine if ChatDBG successfully provided an actionable code fix for the proximate cause of the crash or for the underlying root cause. We used the criteria outlined in Table 5. While fixing root causes is the ultimate goal, fixing proximate causes can still be beneficial as fixing crashes enables further debugging steps.

Figure 8 presents ChatDBG's ability to suggest a fix for either the proximate or root cause of the bug. Generally, ChatDBG is excellent at diagnosing and explaining the reason for the crash, which in itself may be useful to programmers. For BC, GZIP, NCOM, and POLY, ChatDBG tends to suggest replacing the `strcpy` or `sprintf` call with their respective `strncpy` and `snprintf` counterparts to prevent buffer overflows. While correct, this change truncates the input silently. Validation or other measures should be added to obtain a robust fix. The root cause in BC is inside code generated from a YACC file. The `clangd` language server does not handle this case in a way that would let ChatDBG answer the LLM's `definition` requests properly.

In the case of PEG, ChatDBG correctly identifies which pointer is null but typically suggests ignoring it instead of failing immediately. This is similar to YAML2, where ChatDBG recommends replacing the assertion with a check inside a function rather than recommending that the client check that the function's preconditions are met prior to the call. ChatDBG has a relatively high root cause fix rate for YAML1 and TIFF. It often correctly suggests fixes to limit recursion depth (YAML1) and to validate input parameters (TIFF).

RQ3 Summary: ChatDBG was successful in virtually all of our trials in diagnosing and explaining the cause of the crash. It was also capable of providing relevant, actionable fixes: 36% of its suggestions addressed the root cause of the bug, while another 55% corrected the proximate cause.

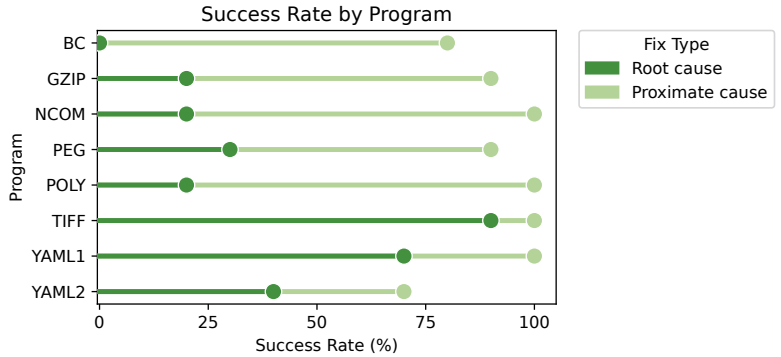


Fig. 8. **ChatDBG success rate at fixing the proximate or root cause in C/C++ programs (§5.2).** ChatDBG successfully identified and fixed the root cause 36% of the time and the proximate cause an additional 55% of the time.

5.3 Threats to Validity

This paper evaluates ChatDBG on two suites of code. The primary suite is a collection of unpublished student labs that may not be entirely representative of code written by, for example, experienced programmers. The second suite consists of real C/C++ applications and bugs drawn from the BugBench and BugsC++ suites. Unlike the Python suite, the C/C++ source code and the bug fixes for these programs are available on GitHub, which may lead to data leakage affecting the C/C++ study if those repositories were part of the training set for the LLMs we used. While the C/C++ suite consists of real-world applications, most of the errors are memory errors. Other types, such as assertion failures, concurrency errors, or other logical errors, may lead to different results.

ChatDBG depends on an LLM to analyze and drive exploration of state, and like all systems based on LLMs today, its performance is affected by prompt engineering. It is possible that ChatDBG's prompts are overfit to the specific GPT-4 models we employed; this threat is somewhat mitigated by the fact that ChatDBG was originally developed using a different model (GPT-3.5-turbo). LLMs are also inherently stochastic, and it is possible to obtain unusually good results by chance. To mitigate this threat, our evaluation runs ChatDBG on each test program at least ten times, which produced stable and repeatable results with only small variation in aggregate.

Our evaluation depends on a manual evaluation of whether ChatDBG's explanation of a bug and its proposed fix are satisfactory. We mitigated the risks of subjectivity by using precisely defined criteria decided upon in advance. Python fixes were deemed successful if the resulting code met the correctness requirements outlined in the assignment. C/C++ fixes were deemed successful at fixing the proximate or root cause using the criteria in Table 5. Fixes described in prose were permitted, provided that the details of all necessary changes to the code were made explicit.

6 Future Work

We see several promising avenues for future work. Incorporating existing fault localization approaches into ChatDBG, rather than relying solely on the LLM's ability to explore the program's code and state, could potentially increase its effectiveness and efficiency by allowing the LLM to focus its attention on suspicious files, functions, or lines of source code. Similarly, incorporating delta debugging [46] could increase the effectiveness of ChatDBG by limiting the amount of input for an LLM and providing failure-inducing events as guidance. Finally, integrating ChatDBG with a time-travel debugger [31, 36] would expand its reach to exploring program state over time, letting

it answer queries that cannot be answered given only the current program state. One challenge of integrating these more sophisticated techniques will be ensuring that the LLM can effectively utilize them, which may necessitate fine tuning or additional training on their usage.

7 Conclusion

This paper presents ChatDBG, the first AI-based debugging assistant. Our evaluation shows that engaging in a debugging dialog with ChatDBG can significantly assist in identifying root causes of errors and developing correct fixes.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 2243636. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Data-Availability Statement

ChatDBG is available on GitHub at `github.com/plasma-umass/ChatDBG`. An archived version is also available on Zenodo [21].

References

[1] Gabin An, Minhyuk Kwon, Kyunghwa Choi, Jooyong Yi, and Shin Yoo. 2023. BugsC++: A Highly Usable Real World Defect Benchmark for C/C++. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2034–2037. doi:10.1109/ASE56229.2023.00208

[2] Yasharth Bajpai, Bhavya Chopra, Param Biyani, Cagri Aslan, Dustin Coleman, Sumit Gulwani, Chris Parnin, Arjun Radhakrishna, and Gustavo Soares. 2024. Let's Fix this Together: Conversational Debugging with GitHub Copilot. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Liverpool, UK, September 2-6, 2024*. IEEE, 1–12. doi:10.1109/VL/HCC60511.2024.00011

[3] Robert M. Balzer. 1969. EXDAMS: Extendable Debugging and Monitoring System. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1969 Spring Joint Computer Conference, Boston, MA, USA, May 14-16, 1969 (AFIPS Conference Proceedings, Vol. 34)*, Harrison W. Fuller (Ed.). AFIPS Press, Boston, MA, 567–580. doi:10.1145/1476793.1476881

[4] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *CoRR* abs/2403.17134 (2024). doi:10.48550/ARXIV.2403.17134 arXiv:2403.17134

[5] LLVM 2025. *What is clangd?* LLVM. Retrieved Feburary 12, 2025 from https://clangd.llvm.org/

[6] B. Efron. 1979. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics* 7, 1 (1979), 1–26. http://www.jstor.org/stable/2958830

[7] Marc Eisenstadt. 1993. Tales of Debugging from The Front Lines. In *Empirical Studies of Programmers: Fifth Workshop*, Vol. 86. Ablex Publishing Corporation, Palo Alto, CA.

[8] Hadeel Eladawy, Claire Le Goues, and Yuriy Brun. 2024. Automated Program Repair, What Is It Good For? Not Absolutely Nothing!. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, New York, NY, USA, 84:1–84:13. doi:10.1145/3597503.3639095

[9] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019), 56–65. doi:10.1145/3318162

[10] Izzeddin Gur, Hiroki Furuta, Austin V. Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. 2024. A Real-World WebAgent with Planning, Long Context Understanding, and Program Synthesis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=9JQtrumvg8

[11] Andrew Head, Fred Hohman, Titus Barik, Steven Mark Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.). ACM, New York, NY, USA, 270. doi:10.1145/3290605.3300500

[12] Jorge E. Hirsch. 2005. An index to quantify an individual's scientific research output. *Proc. Natl. Acad. Sci. USA* 102, 46 (2005), 16569–16572. doi:10.1073/PNAS.0507655102

[13] ipdb 2007. *IPython PDB*. Retrieved March 16, 2024 from https://github.com/gotcha/ipdb

[14] IPyflow 2020. *IPyflow: A reactive Python kernel for Jupyter notebooks.* https://github.com/ipyflow/ipyflow

[15] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=VTF8yNQM66

[16] 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 (2024), 1424–1446. doi:10.1145/3660771

[17] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2025. Explainable automated debugging via large language model-driven scientific debugging. *Empir. Softw. Eng.* 30, 2 (2025), 45. doi:10.1007/S10664-024-10594-X

[18] Amy J. Ko and Brad A. Myers. 2010. Extracting and Answering Why and Why Not Questions about Java Program Output. *ACM Trans. Softw. Eng. Methodol.* 20, 2 (2010), 4:1–4:36. doi:10.1145/1824760.1824761

[19] A. Kotok. 1961. DEC Debugging Tape.

[20] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert DeLine, and Gina Venolia. 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*. IEEE Computer Society, 383–392. doi:10.1109/ESEM.2013.43

[21] Kyla Levin, Nicolas van Kempen, Emery Berger, and Stephen Freund. 2025. *Software Artifact for "ChatDBG: Augmenting Debugging with Large Language Models"*. doi:10.5281/zenodo.15185773

[22] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 15339–15353. doi:10.18653/V1/2024.ACL-LONG.818

[23] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[24] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019* (Beijing, China), Dongmei Zhang and Anders Møller (Eds.). ACM, New York, NY, USA, 169–180. doi:10.1145/3293882.3330574

[25] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 661–673. doi:10.1109/ICSE43902.2021.00067

[26] Mark A. Linton. 1990. The Evolution of Dbx. In *Proceedings of the Usenix Summer 1990 Technical Conference, Anaheim, California, USA, June 1990*. USENIX Association, Berkeley, CA, 211–220.

[27] LLVM 2010. *LLDB Debugger*. LLVM. Retrieved February 6, 2024 from https://lldb.llvm.org/

[28] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, New York, NY, USA, 664–676. doi:10.1145/3468264.3468580

[29] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. BugBench: Benchmarks for Evaluating Bug Detection Tools. https://pages.cs.wisc.edu/~shanlu/paper/63-lu.pdf

[30] Microsoft. 2016. *Language Server Protocol*. Microsoft. Retrieved September 6, 2024 from https://microsoft.github.io/language-server-protocol

[31] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[32] OpenAI. 2020. *OpenAI API*. OpenAI. Retrieved March 18, 2024 from https://openai.com/index/openai-api/

[33] OpenAI. 2023. *Function calling and other API updates*. OpenAI. Retrieved February 24, 2024 from https://openai.com/index/function-calling-and-other-api-updates

[34] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[35] OpenAI. 2024. *Pricing - OpenAI*. OpenAI. Retrieved March 8, 2024 from https://openai.com/pricing

[36] GNU Project. 2009. *Reverse Debugging with GDB*. Free Software Foundation. Retrieved September 6, 2024 from https://sourceware.org/gdb/wiki/ReverseDebug

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[38] Shreya Shankar, Stephen Macke, Andrew Chasins, Andrew Head, and Aditya Parameswaran. 2022. Bolt-on, Compact, and Rapid Program Slicing for Notebooks. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4038–4047.

[39] Richard Stallman, Roland Pesch, Stan Shebs, et al. 2011. *Debugging with GDB*. Free Software Foundation, Boston, MA.

[40] Jupyter Team. 2015. *Jupyter Notebooks*. Jupyter Team. Retrieved March 8, 2024 from https://docs.jupyter.org/en/latest/

[41] Mark D. Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, Seymour Jeffrey and Leon G. Stucki (Eds.). IEEE Computer Society, 439–449. http://dl.acm.org/citation.cfm?id=802557

[42] Michael J. Wooldridge and Nicholas R. Jennings. 1995. Intelligent agents: theory and practice. *Knowl. Eng. Rev.* 10, 2 (1995), 115–152. doi:10.1017/S0269888900008122

[43] Xerox, Systems Development Department. 1979. Mesa Debugger Documentation. https://www.applefritter.com/content/mesa-debugger-documentation

[44] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 17:1–17:12. doi:10.1145/3597503.3623342

[45] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *CoRR* abs/2405.15793 (2024). doi:10.48550/ARXIV.2405.15793 arXiv:2405.15793

[46] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1687)*, Oscar Nierstrasz and Michel Lemoine (Eds.). Springer, Berlin, Heidelberg, 253–267. doi:10.1007/3-540-48166-4_16

[47] Andreas Zeller. 2009. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann.

[48] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

Received 2024-09-12; accepted 2025-04-01