ChatDBG: An AI-Powered Debugging Assistant

Kyla Levin*, Nicolas van Kempen*
University of Massachusetts Amherst
Amherst, MA, USA
{khlevin, nvankempen}@cs.umass.edu

Emery D. Berger†
University of Massachusetts Amherst / Amazon
Amherst, MA, USA
emery@cs.umass.edu

Stephen N. Freund
Williams College
Williamstown, MA, USA
freund@cs.williams.edu

***Abstract*—This paper presents ChatDBG, the first 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* and drive debugging by issuing commands to navigate through stacks and inspect program state; it then reports its findings and yields back control to the programmer. Our ChatDBG prototype integrates with standard debuggers including `LLDB`, `GDB`, and `WinDBG` 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 nearly 30,000 times.**

I. Introduction

Debuggers help programmers identify and fix bugs by letting them investigate program state and navigate program execution. Debuggers for mainstream languages, including GDB [1], LLDB [2], and WinDBG (for C, C++, and Rust), jdb (Java), Pdb (Python), and the Chrome or Firefox debuggers (for JavaScript), generally provide the same functionality. In particular, most debuggers 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; resuming program execution line-by-line (*single-step*) or at the granularity of function calls; and in some debuggers, stepping backward through execution via *reverse debugging*, also known as *time-travel* or *omniscient* debugging.

Debuggers can be helpful, but finding and fixing software defects remains a deeply challenging and time-consuming task. 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 **ChatDBG**, the first 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 [3], enable a debugger to leverage insights and intuition from many thousands of programs as well as the vast real-world knowledge embedded in LLMs.

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. ChatDBG then orchestrates a conversation with the LLM. A key novelty of ChatDBG is that it grants autonomy to the LLM to "take the wheel" while answering the programmer's queries. Specifically, the LLM issues "function calls" [4] 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 for use while 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 four widely used debuggers: GDB, LLDB, WinDBG, 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 87% of the time, typically at a cost of 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, the first AI-powered debugger assistant; it describes the implementation of our ChatDBG prototype; and it introduces the "take the wheel" approach to integrating large language models with developer tools. The paper also presents an evaluation of ChatDBG that demonstrates its significant

*Equal contribution.

†Work done at the University of Massachusetts Amherst.

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.** 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.

advantages over existing debugger functionality.

This paper is organized as follows. Section II illustrates the use of ChatDBG to debug a program. Section III describes key related work. Section IV describes ChatDBG's implementation. Section V presents our evaluation, and Section VI concludes with several promising directions for future work.

II. 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 [5].

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

```
$ 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...

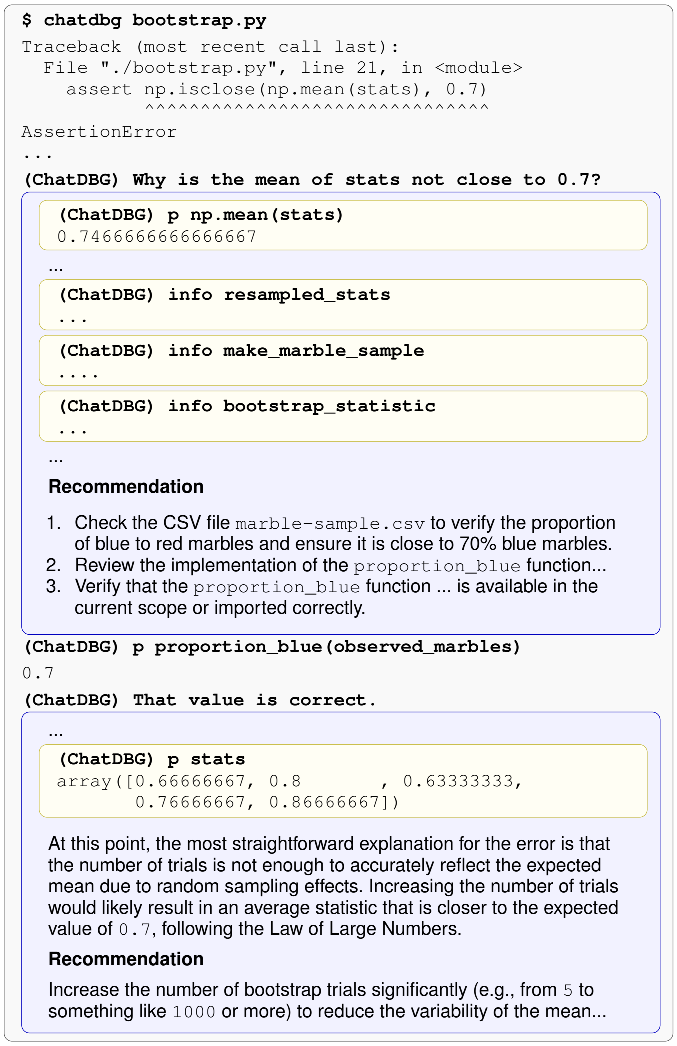
The Python debugger output shows a reference to a variable or function called `proportion_blue`, which is used as the `compute_statistic` parameter for `bootstrap_statistic`. Upon looking at the `proportion_blue` function's definition, it is found that the function simply returns the input `sample` without 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.** 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.** 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.

user-written code, and the docstrings for library code (which we assume is correct and not the root cause of any error).

ChatDBG identifies the root cause: `proportion_blue` fails to compute and return the desired statistic, and it provides a corrected version of `proportion_blue`. 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. The user asks why the mean of `stats` is not close to 0.7, and ChatDBG's first response suggests examining whether 0.7 is the appropriate expected value. The user then computes the proportion of blue marbles with a debugger command and tells ChatDBG that 0.7 is the correct value. In response, ChatDBG points to the low number of trials (5) 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.

III. Related Work

Table I presents an overview of previous interactive debuggers, together with their features and date of introduction. The first interactive debugger, DDT, introduced breakpoints, single-stepping, and stack navigation in 1961 [6]. By 1979, the Mesa debugger had most key features of modern debuggers, including source-level debugging, conditional breakpoints, tracing, and the ability to display runtime state and evaluate code [8]. Arbitrary conditional breakpoints date back at least to 1990 with DBX [9]. Watchpoints were introduced by 1991 and have been in GDB since version 4.0.1 [10]. Reverse debugging it was first implemented in EXDAMS in 1969 [7] is present in widely-used debuggers like GDB (in 2009) [11], and WinDBG (in 2017); the rr debugger, built on top of GDB, also supports reverse debugging on Linux platforms [12]. ChatDBG, by integrating into standard debuggers, inherits and extends their functionality.

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 [13]. 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.

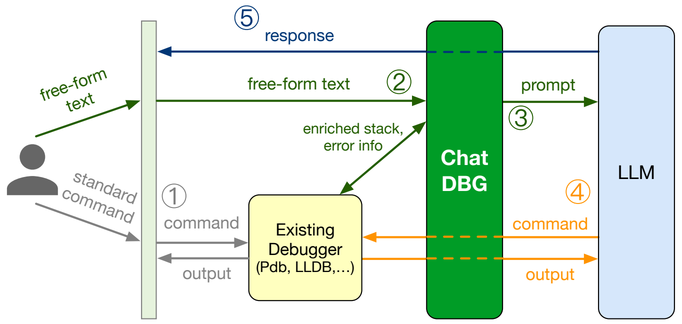
The goal of *program slicing*, introduced by Weiser in 1981 [14], is to produce a shorter version of a program limited to the source code that could have led to an error. Program slicing has been extensively studied; Weiser's paper has been cited over 5,000 times to date. As Section IV-G describes, ChatDBG 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 the root cause of a defect; Wong et al. present a survey with over 400 citations [15]. Unlike previous work, ChatDBG performs fault localization by leveraging LLM-based examination and LLM-driven exploration of source code and program state.

Automated program repair is another highly active area of software engineering research [16]; its goal is to generate source-level program patches that prevent a program from failing. Unlike past work, ChatDBG performs best-effort automated program repair—leveraging its integration within a debugger—via LLM-based examination and LLM-driven exploration of program source and state.

System & Date Introduced	Single Step	Stack Nav.	Break-points	Cond. "	Source Level	Trace	Display State	Eval. Code	Watch-points	Reverse	Explain Bugs	Propose Fixes	Open Queries
DDT [6], 1961	✓	✓	✓										
EXDAMS [7], 1969	✓	✓	✓							✓			
Mesa [8], 1979	✓	✓	✓	✓*	✓	✓	✓	✓					
DBX [9], 1981	✓	✓	✓	1990	✓	✓	✓	✓	✓				
GDB [10], 1986	✓	✓	✓	✓	✓	✓	✓	✓	1991	*2009*			
Pdb, 1992	✓	✓	✓	✓	✓	✓	✓	✓	✓				
WinDBG, ca. 1997	✓	✓	✓	✓	✓	✓	✓	✓	✓	*2017*			
LLDB, 2010	✓	✓	✓	✓	✓	✓	✓	✓	✓				
ChatDBG, 2023	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓

TABLE I: **Debugger features and their dates of introduction.** Most key features have been around for decades. By integrating into modern debuggers (GDB, LLDB, Pdb and WinDBG), 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; for example, WinDBG's reverse debugging only works on specific Intel chips.)



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.
⑤ The LLM responds to the prompt.

```
history = “”
loop
    line = INPUT()
    if ISDEBUGGERCOMMAND(line) then
        output = DOCOMMAND(line)                                  ①
        PRINT(output)
        history = history + (line + “→” + output)
    else
        if not CHATINPROGRESS() then                              ②
            prompt = MAKEPROMPT(INSTRUCTIONS(),
                                ENRICHEDSTACK(), INPUTS(),
                                ERROR(), history, line)
        else
            prompt = MAKEPROMPT(history, line)
        SEND(prompt)                                              ③
        history = “”
        while RESPONSEPENDING() do
            match RECEIVE()
                case DEBUG(cmd) ⇒                                 ④
                    output = DOCOMMAND(cmd)
                    PRINT (cmd + “→” + output)
                    SEND(output)
                case MESSAGE(text) ⇒ PRINT(text)                  ⑤
```

Fig. 4: **ChatDBG architecture and top-level command processing loop.**

IV. Implementation

A. *Using ChatDBG: Preliminaries*

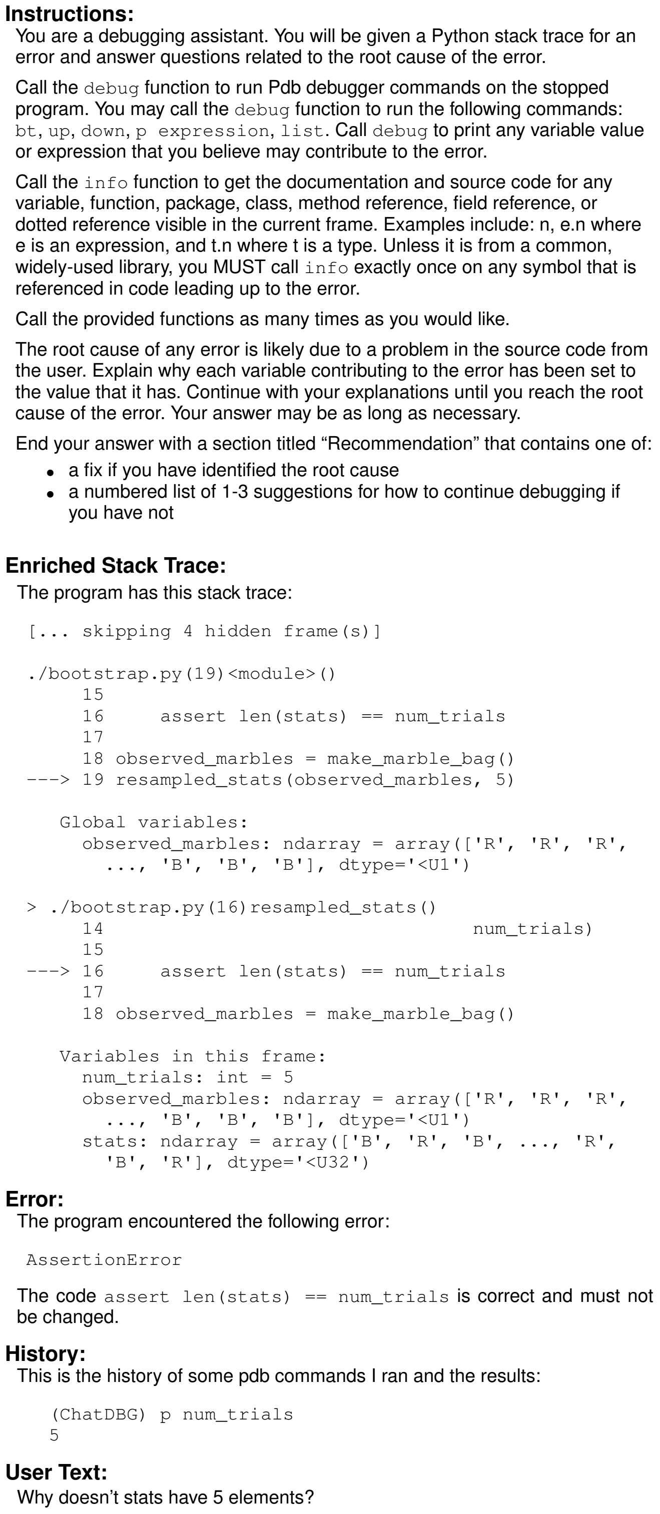
ChatDBG integrates with existing debuggers as either a plug-in or a direct extension. Our primary focus to date has been an extension to Pdb, which supports both non-interactive Python scripts and interactive sessions in IPython or Jupyter notebooks, and a plug-in for LLDB to support C/C++ code. A subset 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.

B. *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.



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.** For brevity, the enriched stack includes only five lines of source in each frame, rather than the default of 10.

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.

C. ChatDBG Architecture Overview

ChatDBG orchestrates communication between the user, the debugger, as shown in the architecture diagram and ChatDBG’s top-level command processing loop in Figure 4. The operations in the pseudocode map naturally onto debugger APIs and onto LLM APIs supporting completion and function calls [4]. ChatDBG currently utilizes OpenAI’s API [17] and GPT-4 models. We elaborate on the most salient technical innovations after this overview.

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 ouputs to the user. Once the full response has been processed, ChatDBG returns control to the user.

D. Initial Prompts and Enriched Stack Traces

In addition to including the user’s text, the initial prompt conveys instructions to LLM and also the context surrounding the error. We highlight the components comprising that context in this section, using the initial prompt in Figure 5 that was generated for the first query in Figure 2.

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 on answers that facilitates reading them and evaluating their quality. Paragraphs 2-4 of the instructions are the *take the wheel* prompt described in Section IV-E.

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

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

TABLE II: **CHATDBG command extensions.** These commands are available to not only the user but also the LLM, and they provide access to information beyond what is typical in a debugger.

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 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 just 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.

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 also 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.

E. Taking the Wheel

CHATDBG supports *take the wheel* debugging via the function call capabilities in OpenAI's API and most recent models [4]. This feature 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.

F. 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. CHATDBG extends the underlying debuggers with several new commands that are designed to help the LLM navigate through and understand the target's code. These commands are available to the LLM via function calls and are listed in Figure II.

CHATDBG augments Pdb with the `info` command, which prints the docstring for any function, class, field, method, or package. It additionally prints the source for any user-defined code. 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` 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 lo-

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 in table
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	Takes a sample from a table smaller than the sample size without replacement
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	Simulation fails to properly create a random sample under null hypothesis

TABLE III: **Python programs exhibiting a variety of common errors.** 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.

cation 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 [18].

G. 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 [19], [20]. 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:

```
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 [20], [21].

V. 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 enable 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++)?

A. Python

We applied ChatDBG to bugs in a collection of student labs from two introductory computer science courses; see Table III. 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 [22] 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.

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 also representative of mistakes often made in languages like Python. 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 ran each program ten times under the five configurations in Table IV: **Default Stack** includes standard stack traces, as generated by `ipdb` [23], 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 are intended 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.

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 [24], 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.

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. Figure 6 shows the success rate under each configuration.

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*	✓

TABLE IV: **Configurations used in the Python experiments.**

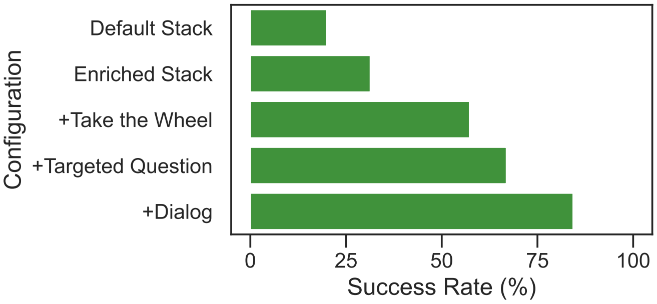


Fig. 6: **Overall ChatDBG success rate for each configuration.** ChatDBG innovations and user-provided context gradually increase effectiveness.

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 identiyfing 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, particuarly 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 taking the wheel led to significant improvement in the success rate for both crashing and semantic bugs, as shown in the **+Take the Wheel** plots.

The LLM most heavily used the `info`, `slice` (for notebooks), and `p` (print) debugging commands. That is not surprising, given that they often provide the most direct insight into the 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. 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

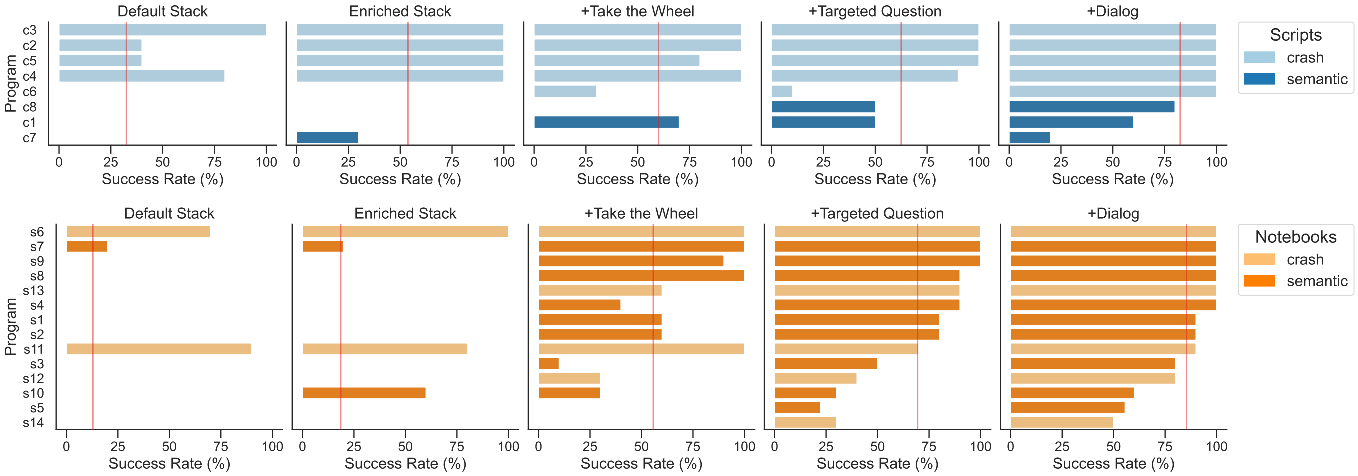


Fig. 7: **Success rate for ChatDBG for each program and configuration.** Vertical lines show the mean.

Program	LoC	Error Type	Root Cause	Proximate Cause Fix	Root Cause Fix
BC [25]	17.0k	Buffer overflow	Input from data file printed to a fixed-size buffer	Truncate on copy	Use dynamic size
GZIP [25]	8.2k	Buffer overflow	Command line argument unsafely copied to a fixed-size buffer	Truncate on copy	Check size & warn/exit
NCOM [25]	1.9k	Buffer overflow	Command line argument unsafely copied to a fixed-size buffer	Truncate on copy	Check size & warn/exit
PEG [26]	14.7k	Null dereference	Invalid input produces corrupted data structure	Check if not null	Warn/exit
POLY [25]	0.7k	Buffer overflow	Command line argument is unsafely copied to a fixed-size buffer	Truncate on copy	Check size & warn/exit
TIFF [26]	58.9k	Division by zero	Combination of command line options leads to a division by zero	Override option values	Warn/exit when invalid
YAML1 [26]	8.7k	Stack overflow	Long sequences of `{` in the input leads to deep recursion	Use iterative method	Guard recursion depth
YAML2 [26]	8.7k	Assertion failure	Specific input causes a peek request for non-existent "next" token	Replace assert	Check before peeking

TABLE V: **Bugs in unmanaged C/C++ code, and our criteria for fixing the proximate cause or the root cause of each.**

prevent it from conducting the amount of reasoning necessary to successfully 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 [27], 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. 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.

B. *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 V summarizes the programs from the BugBench [25] and BugsC++ [26] 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 studied programs do not crash at run time. We employed AddressSanitizer [28] to force a crash at the moment a memory violations occured 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

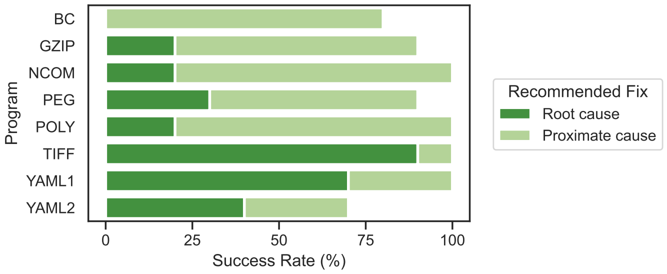


Fig. 8: **CHATDBG success rate at fixing the proximate or root cause in C/C++ programs.** CHATDBG successfully identified and fixed the root cause 36% of the time and the proximate cause an additional 55% of the time.

that information in the initial prompt.

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

We run our C/C++ experiments on an an x86 Ubuntu 22.04 server. We use Clang and LLDB 17.0.6 to compile and debug, using flags `-g -Og -fno-omit-frame-pointer`. CHATDBG used OpenAI's `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 critera outlined in Table V. 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.

C. 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 of errors, 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.

Our evaluation depends on a manual and subjective evaluation of whether CHATDBG's explanation of a bug and its proposed fix are satisfactory. We mitigated the risks of subjective evaluation 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 V. Fixes described in prose were permitted, provided that the details of all necessary changes to the code were made explicit.

VI. CONCLUSION AND FUTURE WORK

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.

We see several avenues for future work. Incorporating an existing fault localization approach into CHATDBG, rather than relying solely on the LLM's ability to explore the program's source code and state, could potentially increase its effectiveness and efficiency by allowing the LLM to focus its attention on suspicious and possibly problematic files, functions, or lines of source code. Similarly, incorporating delta debugging [29] 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 would expand its reach to exploring program state over time, letting it answer queries that cannot be answered given the current program state.

ChatDBG is available on GitHub at `github.com/plasma-umass/ChatDBG`.

References

[1] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with GDB," *Free Software Foundation*, vol. 675, 1988.

[2] T. L. Team, "LLDB debugger," https://lldb.llvm.org/, accessed: February 6, 2024.

[3] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774

[4] "Function calling - OpenAI API," accessed: February 24, 2024. [Online]. Available: https://platform.openai.com/docs/guides/function-calling

[5] B. Efron, "Bootstrap methods: Another look at the Jackknife," *The Annals of Statistics*, vol. 7, no. 1, pp. 1–26, 1979. [Online]. Available: http://www.jstor.org/stable/2958830

[6] A. Kotok, "Dec debugging tape," *Memo MIT-1 (rev.), MIT (Dec. 1961)*, 1961.

[7] R. M. Balzer, "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*, ser. AFIPS Conference Proceedings, H. W. Fuller, Ed., vol. 34. Boston, MA: AFIPS Press, May 1969, pp. 567–580. [Online]. Available: https://doi.org/10.1145/1476793.1476881

[8] Xerox, Systems Development Department, "Mesa Debugger Documentation," https://www.applefritter.com/content/mesa-debugger-documentation, Palo Alto, CA, Apr. 1979.

[9] M. A. Linton, "The evolution of dbx," in *Proceedings of the Usenix Summer 1990 Technical Conference, Anaheim, California, USA, June 1990*. USENIX Association, Jun. 1990, pp. 211–220.

[10] GDB Developers, "Debugging with GDB: New Features since GDB 3.5," Sep. 1991. [Online]. Available: https://web.mit.edu/gnu/doc/html/gdb_2.html#SEC4

[11] ——, "Reverse Debugging with GDB," 2008. [Online]. Available: https://sourceware.org/gdb/wiki/ReverseDebug

[12] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability: Extended technical report," 2017.

[13] A. J. Ko and B. A. Myers, "Extracting and answering why and why not questions about Java program output," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 2, pp. 4:1–4:36, 2010. [Online]. Available: https://doi.org/10.1145/1824760.1824761

[14] M. D. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, S. Jeffrey and L. G. Stucki, Eds. IEEE Computer Society, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=802557

[15] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[16] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

[17] "API reference - OpenAI," accessed: March 18, 2024. [Online]. Available: https://platform.openai.com/docs/api-reference

[18] Microsoft, "Language Server Protocol," 2016. [Online]. Available: https://microsoft.github.io/language-server-protocol

[19] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "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*, S. A. Brewster, G. Fitzpatrick, A. L. Cox, and V. Kostakos, Eds. ACM, 2019, p. 270. [Online]. Available: https://doi.org/10.1145/3290605.3300500

[20] S. Shankar, S. Macke, A. Chasins, A. Head, and A. Parameswaran, "Bolt-on, compact, and rapid program slicing for notebooks," *Proceedings of the VLDB Endowment*, vol. 15, no. 13, pp. 4038–4047, 2022.

[21] "IPyflow: A Next-Generation, Dataflow-Aware IPython Kernel," https://github.com/ipyflow/ipyflow, 2022.

[22] "Jupyter notebooks," accessed: March 8, 2024. [Online]. Available: https://docs.jupyter.org/en/latest/

[23] "ipdb," accessed: March 16, 2024. [Online]. Available: https://github.com/gotcha/ipdb

[24] "Pricing - OpenAI," accessed: March 8, 2024. [Online]. Available: https://openai.com/pricing

[25] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for evaluating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005. [Online]. Available: https://pages.cs.wisc.edu/~shanlu/paper/63-lu.pdf

[26] G. An, M. Kwon, K. Choi, J. Yi, and S. Yoo, "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, 2023, pp. 2034–2037. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00208

[27] J. E. Hirsch, "An index to quantify an individual's scientific research output," *Proc. Natl. Acad. Sci. USA*, vol. 102, no. 46, pp. 16 569–16 572, 2005. [Online]. Available: https://doi.org/10.1073/pnas.0507655102

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[29] A. Zeller, "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*, ser. Lecture Notes in Computer Science, O. Nierstrasz and M. Lemoine, Eds., vol. 1687. Springer, 1999, pp. 253–267. [Online]. Available: https://doi.org/10.1007/3-540-48166-4_16