One Debugger to Teach Them All: Towards Language-Agnostic Debugging in the Browser

Author1
anon1@university.edu
Author1Institution
Author1Location

Author2
anon2@university.edu
Author2Institution
Author2Location

Author3
anon3@university.edu
Author3Institution
Author3Location

Author4
anon4@university.edu
Author4Institution
Author4Location

Author5
anon5@university.edu
Author5Institution
Author5Location

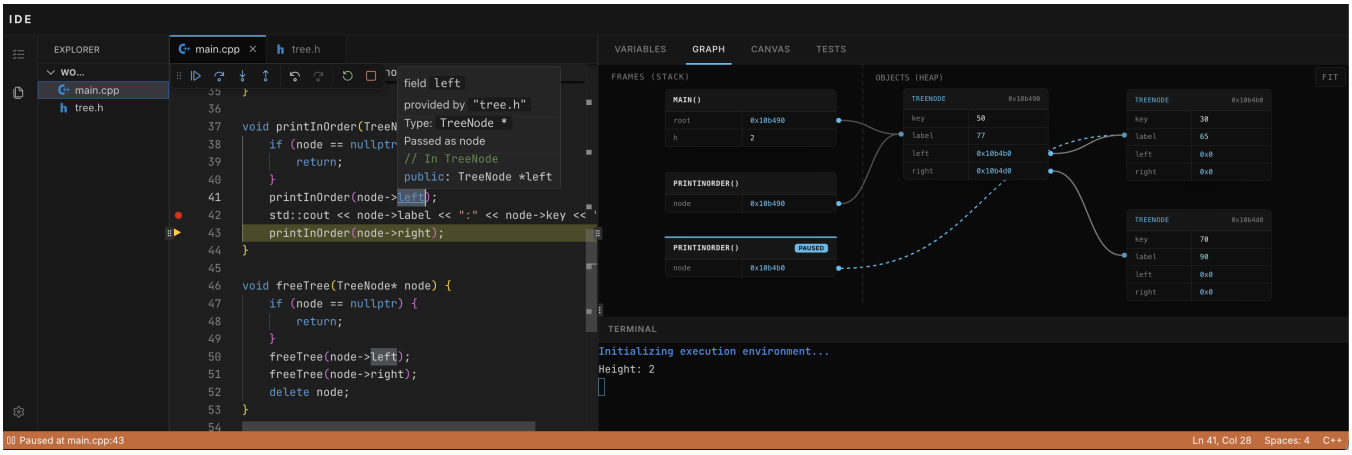

Figure 1: A fully client-side C++ IDE built using our paper's runtime library, demonstrating support for terminal input/output, breakpoints, stepping, and memory visualization, and piloted in a CS2 course at our university.

Abstract

Debugging code is foundational to computer science education; however, existing environments require a compromise between cumbersome local installations and costly server-side sandboxes. We present an open-source, headless library capable of compiling, executing, and debugging C/C++, Rust, and Python entirely in the browser via WebAssembly. Our library exposes an interface via the Debug Adapter Protocol (DAP) and features a client-side debugger that is language agnostic. To demonstrate its utility, we embedded this framework into a web-based integrated development environment (IDE), equipped with multi-file support, memory graph visualization, and a unit testing framework. We deployed an initial pilot of this IDE with a section of students in an offering of CS2 at a large private university. The library and the IDE are publicly available and open source, featuring ten self-guided lessons and guides for instructors to setup classes or self-host. This paper discusses use cases, implementation details, student and instructor experiences, and plans for future extensions.

CCS Concepts

• **Human-centered computing** → *Accessibility*; *Open source software*; **Web-based interaction**; • **Social and professional topics** → **CS1**; • **Software and its engineering** → **Software testing and debugging**.

Keywords

Debugging, WebAssembly, IDE, Python, C++, Rust, Sandboxing, Browser, CS1, CS2

ACM Reference Format:
Author1, Author2, Author3, Author4, and Author5. 2027. One Debugger to Teach Them All: Towards Language-Agnostic Debugging in the Browser. In *Proceedings of the 58th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2027), February 17–20, 2027, Sacramento, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

SIGCSE TS 2027, Sacramento, CA, USA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Running and debugging code is at the core of computer science education. Local toolchains and installations are cumbersome, and cause friction in both the teacher and student experience. Server-side execution introduces hosting costs, latency, and a maintenance burden that is oftentimes unrealistic for educators. In the ideal world, students and educators could get the responsiveness of a local toolchain without ever having to leave their browser.

WebAssembly (Wasm) [4] makes it possible to run many of these toolchains locally in the browser with minimal setup. The browser-based tools that exist today for executing code focus on monolithic applications [3, 5, 24] rather than reusable infrastructure, leaving educators who would like to adapt them for their own bespoke teaching needs to reverse-engineer and re-architect them from the ground up.

1.1 Principal Contributions

We present [anonymized tool name], an open-source, headless library that compiles, executes, and debugs C/C++, Python, and Rust entirely within a student's browser tab. Built on top of existing in-browser execution work [1, 3, 5, 22, 24], our library exposes this functionality through a single, uniform interface using the Debug Adapter Protocol (DAP) [19]. Paired with a reference IDE implementation, we hope to provide educators with the building blocks to create their own custom web-based tools without the need to re-engineer a debugging and execution stack from scratch. Our main contributions include:

(1) **A language agnostic client-side interface for running and debugging compiled and interpreted languages.** To our knowledge, we are the first to expose a single interface for source-level debugging of compiled (C++, Rust) and interpreted (Python) languages that runs in the browser. The source code for our project, along with instructions for using it, can be found at [anonymized GitHub link].
(2) **An open-source reference and a CS2 section pilot.** We release the library alongside a reference IDE which consumes it. We also report on a pilot deployment done in a CS2 course at [a large private university].
(3) **A technique for source-level debugging.** We'll discuss a novel technique for instrumenting Wasm binaries using compiler-emitted debug information to enable debugging compiled languages, as well as a technique for debugging interpreted languages without modifications to the interpreter, both of which mirror the performance of running the language toolchains locally on the student's own machine.

2 Related Work

Pedagogic Environments. Python Tutor is a seminal program visualization tool that has demonstrated the value of visualizing code for new computer science learners [3, 7]. Architecturally, the platform relies on server-side execution to generate a language-independent execution trace for front-end rendering. While computationally inexpensive [2], this architecture introduces operational overhead by requiring a remote backend to run code. Furthermore, Python Tutor is intentionally restricted in scope to support whiteboard examples, deliberately avoiding full IDE capabilities [2]. Other prominent educational platforms face similar infrastructural hurdles. For example, Harvard's CS50 initially relied on custom sandboxed execution environments [14] to overcome significant security challenges regarding malicious code. Although CS50 has since transitioned to GitHub Codespaces to handle requirements like grading and hosting of course materials, the underlying complexity of secure code execution remains a barrier for tool developers. *bcode* [24], a platform enabling students to share and collaborate on programs in-class, circumvents this challenge by supporting local execution via Wasm, but does not have support for debugging.

Execution Infrastructure. Server-side execution has historically been necessary for executing non-browser-native languages in the browser. This paradigm is shifting rapidly, however, with the introduction of Wasm, which offers first-class support for compiling and executing code to run in the browser [15]. Recent work has demonstrated that Wasm is promising for client-side code execution. PyodideU introduced a fully in-browser Python IDE [5], demonstrating that a complete IDE experience, including debugging support, is achievable without any server-side execution. Emception [22] compiles Emscripten [27], a mature C/C++ to Wasm toolchain, itself to Wasm to run in-browser, but is limited to running C/C++ programs. Rubri [1] compiles Miri [6], a well-known Rust intermediate-language interpreter, to Wasm to enable running Rust code in-browser, but lacks debugging support and, like Emception, is language-specific. We believe that building tools like PyodideU and others today remains a large engineering lift, given how much toolchain-specific knowledge is required.

WebAssembly Runtimes. There are existing environments like *webvm.io/*[10] which run Linux in the browser by compiling it to Wasm. In theory, you could then use `gcc` or other toolchains to compile, run, and even debug non-browser code in this virtualized Linux environment. In practice, this approach is prohibitively slow, and is a poor interface to build educational tools on top of, especially if they rely on a UI or interactive graphics. Browsers themselves can also debug Wasm at the source level: Chrome's DevTools steps through C++ or Rust compiled to Wasm using debug information. This capability, however, lives inside the browser's own developer tools and is intended for debugging the page's precompiled Wasm binaries rather than serving as a reusable interface that a web-based tool can drive, style, or build interfaces around.

3 A New Way to Run Code

Unlike prior in-browser tools, which are each tied to a single language, our library serves both compiled and interpreted languages through the same debugging interface. Furthermore, we provide an open-source reference IDE, [anonymized tool name], built using this library for educators to use as a starting point for their own projects and tools; a screenshot of this IDE is shown in Figure 2. We outline a few advantages of our approach below.

Ease of installation. For instructors and tool developers, adopting our library requires installing a single package into their web project, after which our reference IDE and accompanying examples serve as a template for further development. For students, first-time setup is fully automated and completes in seconds, with subsequent runs finishing in around a second (see Figure 3). Traditional IDE

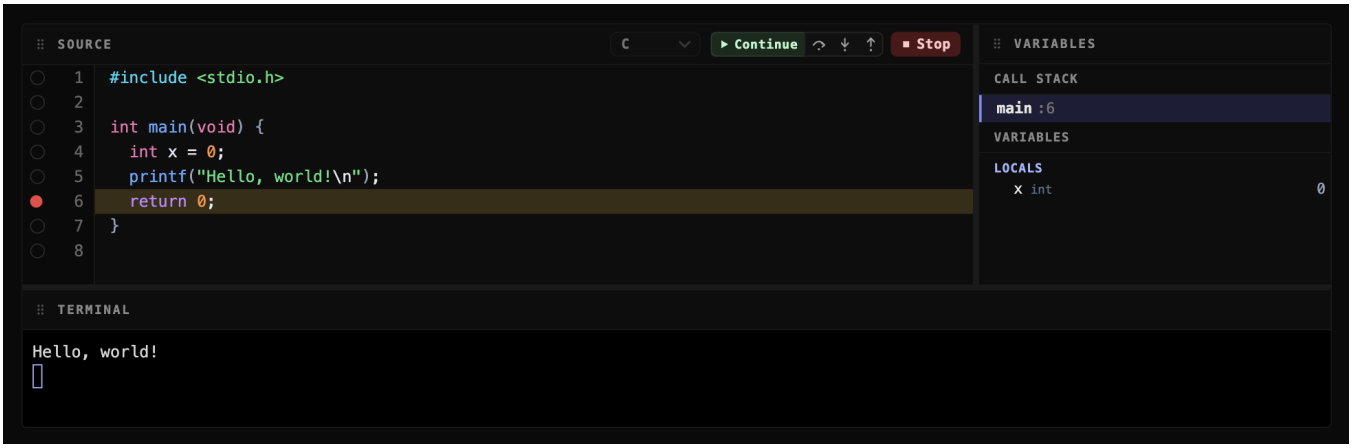

Figure 2: [anonymized tool name], a multi-language (Rust, C/C++, Python), in-browser reference IDE built using the library presented in this paper. The IDE is fully open-source, allowing educators to extend it or adapt it into their own custom tools. The source code for the IDE can be found at [anonymized GitHub link].

and environment setup in CS1/CS2 courses, on the other hand, is typically manual, error-prone, and can take anywhere from minutes to hours.

Scalability. The student brings their own compute to compile and run programs, eliminating the need for server infrastructure. This approach has been used previously to power code execution for MOOCs with thousands of learners [5]. Interactive applications like debugging become easier to implement and are more responsive when run locally.

Security. Code running through our library runs only on the student's machine. Deadlocks, infinite loops, crashes, malicious code, and attempts to tamper with the environment are all contained to the student's browser tab, largely eliminating the need to sandbox their code.

Privacy. Student code and data never leave their machine. In fact, the only activity over the network is to download Wasm binaries required for the library to run. This is particularly valuable in educational settings, where student work is protected under regulations such as FERPA.

3.1 Use Cases

As a piece of reusable software, our work unlocks the following use cases, some of which we show in our implementation of a small pilot IDE described in a later section.

Program Visualization. Many CS1/CS2 concepts require students to reason about program state that is not visible in the source code: the structure of a linked list, the current value of a pointer, or the relationship between stack variables and heap objects. By setting breakpoints in a linked list assignment, for example, an instructor can step back and forth through execution and ask students to predict how the memory state will change next. Our pilot IDE demonstrates that our library can be used to implement program visualization locally.

Debugging Assessments. We prototyped a new style of assessment in our CS2 IDE whose main task is to review, test, and fix code rather than write it. In an age of LLM-generated code, this lets instructors evaluate how a student uses the debugger, rather than relying solely on their final submission. It also teaches students to act as code critics rather than only code authors.

AI Tutors. A growing body of work pairs students with an LLM-based tutor [8, 9, 11, 13, 21], but such tutors usually reason about code without running it, leaving them to guess at runtime behavior. A tutor built on top of our library might compile a student's program, execute it, and read the actual program state before responding.

Autograders. Instructors can run tests using the same framework students see in-browser, either client-side or headlessly on a server, without requiring a separate testing infrastructure.

Course Materials. The library can be embedded directly into digital textbooks and lecture notes as live, debuggable code snippets. A reader might step through a real C++ or Rust example inline, or an interactive snippet might dynamically visualize the state of memory at a specific point in the program.

4 Deploying a Student IDE

To demonstrate what can be built on top of our library, we implemented a browser-based integrated development environment (IDE) for a typical CS2 student workflow that we piloted during a CS2 offering at our institution. The IDE was designed to remove local setup friction, make debugging a part of the default workflow, support real CS2 projects, and allow instructors to integrate it in their own classes or LMS without needing to adopt a specific backend. Ten self-paced debugger assignments are publicly available at [anonymized link].

4.1 Core Features

Our IDE combines the utility of code visualizers and professional IDEs. Visualization tools like Python Tutor [3] show the value of displaying the execution state, but are optimized for single file demonstrations. Professional IDEs and command-line debuggers support more complex projects but require debugger configuration and require students to mentally reconstruct pointer structures from flat variable panes. In our IDE, students can move from failing a unit test directly to a source-level debugging session, inspect variables and stack frames, and compare their mental model of a data structure against the actual runtime state shown by the memory graph. A screenshot of the IDE is shown in Figure 1.

Multi-File Project Support. In CS2, students often work with header files, implementation files, provided starter code, class definitions, and separate test files. The IDE includes a project file tree and an interface to create, delete, and rename files in the in-browser file system.

Breakpoints and Time Travel Debugging. Students can set breakpoints, step line by line, continue execution, and inspect variables in the stack and visually traverse pointers. This allows them to identify when program state diverges from their expectations. Students can also step backwards to replay changes to program state. Reverse stepping replays a capped history of snapshots. This approach trades memory for a simple and deterministic history that does not require re-execution.

Memory Graph Visualization. In addition to standard variable inspection, students can also view a graph that represents the current state of all stack frames and all dynamically allocated memory in the heap. The visualizer was designed for scenarios when the relevant program state is difficult to infer from variable inspection, including linked lists and trees. While actual pointer address values are shown alongside objects in the graph, pointer connections are represented with arrows—similar to Python Tutor and common whiteboarding practices in teaching dynamic memory [3]. Extending Python Tutor, the user can move through the memory graph by zooming in and out and organize nodes on an infinite canvas. Our approach allows the visualization of complex data structures like trees with an arbitrary number of elements.

Unit Testing. Students and instructors can create and run unit tests directly in the IDE. This allows unit testing to be an entry point into debugging rather than a final check for correctness after completing the assignment.

Code Intelligence. The IDE also runs a client-side, in-browser `clangd` [12] server to enable error highlighting, go-to-definitions, and context-aware refactoring. Since this feature is more resource intensive, we made it an opt-in feature to support both students who want IDE-style code assistance and those who prefer a lightweight default.

4.2 Integration in the Classroom

The deployed system includes a platform that lets instructors create their own classes and assignments, which students then open and complete. Creating an assignment is straightforward: instructors fill in their starter code, assignment description, and then publish. Both the entire UI and the underlying library (described in this paper) are available open-source. We provide setup instructions so that the IDE can be integrated into the classroom without cost or significant setup time[1]. We also provide an open-source React component that includes multi-file support, code execution, and callbacks for file saving. This allows for integration in existing LMS platforms as needed.

4.3 Student Usage and Reactions

Our CS2 IDE has been used in a small pilot of nine students in weekly sections and tutoring sessions at our institution. Students accessed the IDE through the browser and used it to complete in-class section problems, run or test their code, and use the debugger to inspect program behavior.

To understand how students used the IDE during the pilot, we logged system-level anonymous telemetry data. Across the pilot we logged 2,742 events (edit, run, set breakpoint, step, continue, test) across 176 unique sessions.

After Program Run	Count	Percentage
Entered Debug Session	162	70.43%
Other Action (edited/re-run)	68	29.57%
Total Runs	**230**	**100%**

Table 1: Transition rate from program execution to debugging. A debug session was counted if any debug action (breakpoint, step, continue) occurred within 5 minutes of a run/test event and before their next run event.

When completing standard section problems without being prompted to use the debugger, 162 of 230 (70.43%) program runs were followed by at least one debug action (setting a breakpoint, stepping, or continuing) within five minutes of the run and before their next run. We classify the remaining 68 runs (29.57%) as edit/re-run or other. This suggests that the students incorporated the debugger in their workflow rather than treating the IDE as run and check tool. However, these usage metrics just show an adoption of the workflow and not causal learning improvements.

Students and instructors who used the IDE had positive reactions to the IDE and debugging in the browser. Instructors were pleased with the ease of use, not requiring resources to troubleshoot compiler setup issues. Regarding teaching dynamic memory allocation, they noted that showing both the memory addresses and the graph visualization was helpful.

Throughout the pilot, students informally shared their feedback on the IDE during and after their section and tutoring sessions. Some of their feedback makes reference to a traditional IDE used in this offering of the course. The following is a (paraphrased) sample of their experience in sections that used the in-browser IDE and visualization tools.

> *Seeing where the pointer is when looping through the linked list in the assignment helped me understand what is actually going on.*

[1] See [anonymized GitHub link] for more information on how to set up this IDE.

> *I feel most peers who struggle with understanding what happens behind the scenes have opted for just memorizing patterns and checking if tests pass.*
>
> *I had to keep force quitting [the other IDE] because there was some issue on my computer so it was nice to not have to worry about that.*
>
> *I think many people struggle with visualizing binary tree problems (not understanding traversals + how to incorporate it in problems), where the visualizer definitely would help.*
>
> *I really liked that you can go backwards in the debugger too because I always missed something and you can't do that in [the other IDE].*

Students generally responded positively to in-browser debugging and the memory visualizer. Based on this response, we plan to launch the IDE to a larger cohort of students in a future CS2 offering, including support for assignment submissions and lecture code.

5 Building a Cross-Language Debugger

Much of the library is written in Rust and compiled to Wasm, a choice which allowed us to write performant and safe code while taking advantage of a rich Rust/Wasm ecosystem. Everything runs on the client with all of the work spread across two threads—the browser main thread and one *Web Worker* [18] thread where student code runs. The two threads communicate via a `SharedArrayBuffer` [17], a browser facility for multithreaded memory sharing. We allow clients of the library to execute and debug C, C++, Python, and Rust code. The library runs the entire compilation toolchain and user program in the browser.

Two execution contexts. The main thread is the orchestrator of the debug environment and hosts the public `Engine` API. The `Engine` acts as a handle to the running program and allows users to interact with the filesystem, standard I/O, and the debugging interface. When student code is run, the worker compiles and executes the program and communicates with the main thread via a `SharedArrayBuffer`. Furthermore, we use `SharedArrayBuffer` to pass control signals to the worker and provide the main thread with zero-copy access to the program's running memory. The use of shared memory allows us to pause the execution of the user program via the web's `Atomics` API [16] and provides efficiency gains over a message-passing system, but comes at the cost of requiring any consumer of our library to have strict cross-origin isolation enabled in their site. This restriction is intended to circumvent known security concerns of `SharedArrayBuffer`, but in practice, we have not found this to be a significant barrier to adoption.

5.1 Running Code in the Browser

To run a student's program, a compiler or interpreter is needed for that language that can run in the browser. We use custom Wasm builds of production language toolchains. Specifically, we use `CPython` 3.11 (Python), `Clang` 22.1.0 (C/C++) and a modified[3] `rustc` 1.96.0 (Rust).

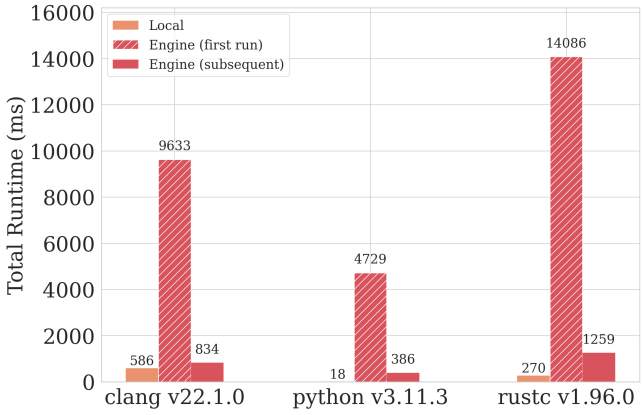

Figure 3: Total execution time (including compiling/linking) of a "Hello, World!" program in our library compared to the equivalent toolchain running locally on an M4 MacBook Pro on a 100 Mbps down/15 Mbps up network[2]. Each bar represents the average of 10 executions. Subsequent runs perform much better (and often comparably to local speeds) than the first run due to the browser's built-in HTTP caching of downloaded toolchain binaries.

These toolchains were compiled to run on and target the open-source WASI standard [26], which provides a system call interface for the filesystem, program arguments, environment variables, and other system components, that are required by each toolchain. WASI was chosen due to its widespread popularity and mature support in compilation toolchains such as LLVM. Furthermore, this approach allows us to make use of the existing production toolchains in the browser without having to run them in a server or significantly modify them ourselves to run locally. As they were not necessarily intended to be used in the browser (and in some cases, such as `rustc`, have not previously been run in the browser up to this point), some toolchains require minimal patches, e.g. to disable threading and networking that would not be supported by the environment; support for these features could be explored more in a future work. Outside of these patches, the toolchains behave as if they were running locally on the student's own machine.

Consider, for example, the process to compile and run a student's C++ project: first, all source files are loaded into an in-memory filesystem. Then, the worker invokes the Clang compiler on each source file, and finally links the object files into a single in-memory `main.wasm`. This file is then executed inside the worker, invoking it in the same manner as the compiler/linker. Executing all steps inside the worker provides isolation from the main thread, preventing the browser tab from freezing due to a long-running loop or crashing unexpectedly due to an error in the student code.

Standard output is streamed to the client as it is produced by the running program, while blocking input calls like `cin`, `scanf`, `input()` are backed by a circular buffer on a `SharedArrayBuffer`.

[2]These bandwidth figures were chosen to approximate the global average mobile network speed reported by the Speedtest Global Index [20], rather than the (often much faster) fixed broadband speeds typical of university or corporate networks.

[3]As our environment does not support multithreading, compiler patches were needed to allow `rustc` to run in a single thread.

Attempts to read data while the buffer is empty stall the worker until the client provides the next line, e.g. from a connected terminal interface. We use this to support interactive programs in a language-agnostic manner.

5.2 Program Debugging

The public debugger interface implements Microsoft's Debug Adapter Protocol (DAP) [19]. DAP was originally designed to help implement VSCode debugging extensions. It's a well-known, language-agnostic interface that standardizes core debugging operations, such as setting breakpoints, stepping through code, and inspecting variable types and values. We chose DAP because it has been extensively tested against production level debuggers like `LLDB`. It also has a large and expressive API that is self-describing, making it easy to understand and implement the requirements for a debugger. DAP's uniform JSON request/response structure made it straightforward to verify the correctness of the underlying implementation via reproducible test cases and to compare the behavior of our debugger against a reference implementation, such as `LLDB`'s implementation of DAP. In addition, we used LLMs to help implement some debugger features, with the test suite verifying the correctness of each change. Arguably the biggest appeal is that DAP provides a unified interface that bridges the gap across languages which represent fundamentally different modes of execution: C/C++ and Rust are compiled, whereas Python is interpreted.

5.2.1 Compiled Languages. To the best of our knowledge, we have developed the first in-browser, client-side debugger for multiple compiled languages. Others have attempted to compile LLDB/GDB into WASI, but failed when facing environmental constraints such as the lack of a standardized `ptrace` system call for the web. Others have tried to run GDB in a Wasm Linux container, but the startup times are unacceptable for education. Our approach avoids these concerns by moving the work of debugging into the library itself.

We compile the student program into Wasm with debug information. Using this information, we instrument each point in the compiled binary corresponding to the start of a source statement, and insert instructions that capture internal Wasm state to a shared memory *debug stack*, along with a call to an imported `bkpt` method that conditionally stops execution according to whether or not a breakpoint was set. The main thread, which holds a shared reference to the debug stack, is able to lazily reconstruct a backtrace and variable values after a breakpoint is hit. Many variables require custom formatting for them to be useful in a debugging context. For example, it's better to show the elements of a C++ `std::vector` rather than its underlying begin/end data pointers. We provide an extensible formatting interface similar to the one provided by LLDB.

5.2.2 Interpreted Languages. Debugging an interpreted language requires none of the binary instrumentation described above, since the interpreter itself maintains program state in a queryable way. For Python, we run a prebuilt CPython interpreter binary in the worker and execute the user's program inside a small Python harness built on top of `bdb` [23], a standard library in Python that underpins its built-in interactive `pdb` debugger. `bdb` provides an interface to receive a callback on every source line, function entry/exit, and uncaught exception. On each callback, the harness checks a special file backed by a `SharedArrayBuffer` to see whether a breakpoint is set at the current location, function, or exception. If so, it eagerly serializes the program state, such as call frames and variable info, and sends it to the main thread to be exposed to the client over the DAP interface. Finally, it pauses execution until the client signals to continue running.

Using a `SharedArrayBuffer`-backed file lets the main thread control the debugger without modifying the interpreter binary itself. We provide implementations for this file's `read` and `write` system calls to provide a debugger interface that is accessible to the harness: on executing a line or entering a function, it `read`s from the file to fetch the list of active breakpoints and determine if execution should stop. On a breakpoint, the harness `write`s a specific command to instruct the worker thread to pause execution. This design allows setting breakpoints and pausing execution while the program is running. A student stuck in an infinite loop, for instance, can pause execution on demand and inspect the state to find the bug without having to have set a breakpoint in advance.

6 Future Work

We are actively working on a few extensions to the debugger to make it more robust, including extended language support for commonly used classroom languages, such as Java and C#, and a richer feature set enabling the debugger's use in other pedagogical settings. For example, we aim to provide an interface for interactive graphics for CS1/CS2 offerings as well as multi-threading/concurrency for OS courses.

The use of instrumentation to debug compiled languages limits the range of possible features a debugger might support. For example, modifying variables in the debugger while the program is running, resuming execution at an arbitrary point in the program, and reverse execution (e.g. step back, reverse continue) are all difficult or impossible to implement with this approach. A future work might develop or extend [25] a lightweight Wasm interpreter that supports these use cases, and investigate the difference in overhead compared to the instrumented approach.

More broadly, we see [anonymized tool name] evolving into an open platform where educators build and share course-specific libraries, custom visualizations, and code snippets with their students via a plugin-based architecture where educators can author their own extensions to the platform. With a broader community using the platform, we plan to complete and expand our evaluation with wider deployment data and user studies across different institutions and course contexts. One concrete application we are pursuing at our own institution is using the platform to log and analyze debugging sessions from our CS1/CS2 TA hiring pipeline, with the goal of publishing a broader study of how people problem-solve and debug code when presented with a piece of unfamiliar code.

Acknowledgments

We're grateful to [anonymized name], who saw promise in the early work that eventually grew into this project. Thanks also to [anonymized name] and [anonymized name], whose willingness to support this work in our coursework gave it the foundation it needed to develop further.

References

[1] 2024. Rubri: The Rust Browser Interpreter. https://github.com/LyonSyonII/rubri. Accessed: 2026-07-01.

[2] Philip Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 1235–1251. doi:10.1145/3472749.3474819

[3] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. 579–584. doi:10.1145/2445196.2445368

[4] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. doi:10.1145/3140587.3062363

[5] Thomas Jefferson, Chris Gregg, and Chris Piech. 2024. PyodideU: Unlocking Python Entirely in a Browser for CS1. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 583–589. doi:10.1145/3626252.3630913

[6] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL, Article 48 (Jan. 2026), 29 pages. doi:10.1145/3776690

[7] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. doi:10.1145/3126594.3126632

[8] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 650, 20 pages. doi:10.1145/3613904.3642773

[9] Charles Koutcheme, Juliette Woodrow, and Chris Piech. 2026. Aligning Small Language Models for Programming Feedback: Towards Scalable Coding Support in a Massive Global Course. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1* (USA) *(SIGCSE TS 2026)*. Association for Computing Machinery, New York, NY, USA, 610–616. doi:10.1145/3770762.3772539

[10] Leaning Technologies. 2024. WebVM: Server-less x86 Virtual Machines in the Browser. https://webvm.io/. Accessed 2026-07-01.

[11] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. 2024. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. doi:10.1145/3631802.3631830

[12] LLVM Project. [n. d.]. clangd. https://clangd.llvm.org/. Accessed: 2026-07-02.

[13] Wenhan Lyu, Yimeng Wang, Tingting (Rachel) Chung, Yifan Sun, and Yixuan Zhang. 2024. Evaluating the Effectiveness of LLMs in Introductory Computer Science Education: A Semester-Long Field Study. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale* (Atlanta, GA, USA) *(L@S '24)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/3657604.3662036

[14] David J. Malan. 2013. CS50 sandbox: secure execution of untrusted code. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 141–146. doi:10.1145/2445196.2445242

[15] MDN Web Docs. 2024. Compiling a New C/C++ Module to WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/C_to_Wasm. Accessed 2026-07-01.

[16] MDN Web Docs. 2026. Atomics - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics. Accessed: 2026-07-02.

[17] MDN Web Docs. 2026. SharedArrayBuffer - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer. Accessed: 2026-07-02.

[18] MDN Web Docs. 2026. Web Workers API - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API. Accessed: 2026-07-02.

[19] Microsoft. 2024. Debug Adapter Protocol. https://microsoft.github.io/debug-adapter-protocol/. Accessed 2026-07-01.

[20] Ookla. [n. d.]. Speedtest Global Index. https://www.speedtest.net/global-index. Accessed: 2026-07-01.

[21] Tung Phung, Heeryung Choi, Mengyan Wu, Christopher Brooks, Sumit Gulwani, and Adish Singla. 2026. Closing the Loop: An Instructor-in-the-Loop AI Assistance System for Supporting Student Help-Seeking in Programming Education. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1* (USA) *(SIGCSE TS 2026)*. Association for Computing Machinery, New York, NY, USA, 852–858. doi:10.1145/3770762.3772612

[22] Jose Prendes. 2024. Emception: WebAssembly-based C++ Emulator. https://github.com/jprendes/emception. Accessed: 2026-07-01.

[23] Python Documentation. 2025. bdb — Debugger framework. https://docs.python.org/3/library/bdb.html. Accessed: 2026-07-02.

[24] Jacob Roberts-Baca, Joshua Delgadillo, and Chris Piech. 2026. Rooms of Their Own: Structured Small-Group Learning in a Realtime Browser-Based IDE. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1* (USA) *(SIGCSE TS 2026)*. Association for Computing Machinery, New York, NY, USA, 943–949. doi:10.1145/3770762.3772664

[25] Ben L. Titzer. 2022. A fast in-place interpreter for WebAssembly. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 148 (Oct. 2022), 27 pages. doi:10.1145/3563311

[26] WASI Subgroup, W3C WebAssembly Community Group. [n. d.]. WASI: WebAssembly System Interface. https://wasi.dev/. Accessed: 2026-07-02.

[27] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. https://emscripten.org. Accessed: 2026-07-01.