Implementing Debugging Support for xeus-cpp

for Google Summer of Code 25'

Abhinav Kumar

IIT Indore (2021 - 2025)

Project Mentors:

@anutosh491 @johanmabille @Vipul-Cariappa @aaronj0

Contents

Personal Information

Points of contacts and other relevant links

- Name - Abhinav Kumar
- University - Indian Institute of Technology Indore (IIT Indore)
- Degree - Bachelor of Technology (B. Tech.)
- Major - Computer Science and Engineering
- Residence - Varanasi, Uttar Pradesh, India
- Timezone - Indian Standard Time (UTC + 5:30)
- Typical Working Hours - 6:00-10:00, 19:00-24:00 (IST)
- Github Account - GitHub
- Email -
    1. cse210001002@iiti.ac.in (Institute mail)
    2. kumar.kr.abhinav@gmail.com (Github linked mail)
    3. abhinavdnpiasb@gmail.com (Personal use)
- Language spoken - English

I am Abhinav Kumar, a 4th-year Computer Science & Engineering undergraduate at IIT Indore, specializing in C++ with a deep passion for low-level programming, compilers, operating systems, and system engineering. My journey began with competitive programming, leading me to backend development and eventually transitioning to systems programming through cybersecurity and compiler research.

I am an Open Source and Software Development enthusiast and have contributed to xeus-cpp, CppInterOp, LLVM and others in the past . My goal is to explore a vast majority of Computer Science fields during my undergrad degree like Operating Systems, Database Management Systems and Compiler Design and Architecture.

Programming Background

- **General Programming Background -** I was introduced to C++ back in December of 2021 when I started doing Competitive Programming at my college. The course really showed me the power C++ carries and all its advantages. I have been coding in C++ ever since. I am also highly experienced in building scalable software applications and DevOps. During my third year, my focus shifted to cybersecurity which led me into systems programming. In my final year, I started contributing to LLVM, CppInterOp, and xeus-cpp, focusing on compiler optimizations, language interoperability, and Jupyter-based C++ kernels. These contributions reinforced my expertise in low-level software, compiler design, and tooling, completing my transition from backend development to systems programming.

- **Git and Github experience -** I have been using git for around 3 years now and I am quite familiar with all workflows involved with git and github . I understood the more tedious and error prone stuff like rebasing on other branches, hard and soft reset etc through making a decent number of mistakes on my PR's while contributing to LLVM, xeus-cpp and CppInterOp.

- **Platform Details -** I use macOS Sequoia (15.3.1) as my operating system along with Visual Studio Code as my primary editor and debugger whenever I am working on a cpp/python project.

- **Internship Experience -** I interned twice at Trilogy Innovations (Codenation) as an SDE Intern, working on:
  - **CloudfixAI** – An LLM-powered cloud cost optimization assistant with Go, AWS Lambda, and GPT-4o, integrating GitHub CI/CD to reduce latency to 2-3 seconds.
  - **Sherlock** – A graph-based problem-solving assistant leveraging AWS Lambda, Amplify, GraphQL, and OpenAI models. These experiences strengthened my software engineering, system design, and problem-solving abilities

Motivation behind participating in GSOC 25'

General - GSoC aligns perfectly with my long-term goal of building robust, high-performance system software, and I am eager to contribute and grow through this experience. I am to work with experienced mentors to contribute to impactful open-source projects.

xeus-cpp - I am passionate about compilers, operating systems, and low-level development, with C++ as my primary tech stack. xeus-cpp provides the perfect opportunity to contribute to real-world open-source projects in these areas while refining my expertise in systems programming, compiler optimizations, and OS internals.

Contributions to Compiler Research and LLVM

Here are my contributions:

Pull Requests(Merged)

- **compiler-research/CppInterOp** Added undo command for CppInterOp
- **compiler-research/xeus-cpp** Edited docs to state that tests are ON by default
- **compiler-research/xeus-cpp** Enabled file magic support for xeus-cpp-lite
- **compiler-research/xeus-cpp** Fix Inspect Request Failure in xeus-cpp-lite
- **llvm/llvm-project [libc]** Added support for fixed-points in is_signed and is_unsigned.
- **llvm/llvm-project [libc]** Enable stdfix functions for macOS arm64 targets.
- **llvm/llvm-project [clang][analyzer]** Ignore unnamed bitfields in UninitializedObjectChecker

Pull Requests(In Progress)

- **compiler-research/xeus-cpp** Added tests for xinspect

- **compiler-research/xeus-cpp** Added XEUS_SEARCH_PATH support in xeus-cpp

Issues Raised

- **llvm/llvm-project** [libc] Fixed-point types reject negative literals in constexpr context, unlike unsigned integers
- **llvm/llvm-project** [libc] std::is_signed_v<T> for any fixed-point always returns false
- **llvm/llvm-project** [libc] Ensure compatibility and proper functionality of stdfix on Darwin-based Apple systems

Overview of the Project

This proposal outlines integrating debugging into the xeus-cpp kernel for Jupyter using LLDB and its Debug Adapter Protocol (lldb-dap). Modeled after xeus-python, it leverages LLDB's Clang and JIT debugging support to enable breakpoints, variable inspection, and step-through execution. The modular design ensures compatibility with Jupyter's frontend, enhancing interactive C++ development in notebooks.

Proposed Work

In this section , I will be explaining the details of my project . I expect this structure to be considerably improved under the guidance of my mentors and fellow contributors.

The Goal

- **Enable Interactive Debugging:** Provide a seamless debugging experience for C++ code in xeus-cpp, including breakpoint management, variable inspection, and stack tracing.
- **Leverage Existing Standards:** Use the Microsoft Debug Adapter Protocol (DAP) for compatibility with Jupyter's frontend, mirroring xeus-python's approach.
- **Integrate LLDB:** Utilize LLDB's JIT debugging capabilities and Clang integration to handle xeus-cpp's dynamic code execution.

- **Ensure Kernel Stability:** Implement debugging via an external process (lldb-dap) to prevent kernel freezes during breakpoints.
- **Minimize Overhead:** Design a lightweight integration that reuses xeus infrastructure (e.g., xeus-zmq) and avoids unnecessary complexity.
- **Future-Proofing:** Support extensibility for advanced features like remote debugging (e.g., WebAssembly) and alternative debuggers (e.g., GDB).**Testing:** Implement robust testing framework for testing debugger using GoogleTest.

Background

- LLDB, part of the LLVM project, is an ideal debugger for xeus-cpp due to its native support for:
  - **JIT-Compiled Code**: LLDB can debug machine code generated by Clang's JIT, which xeus-cpp relies on (similar to Clang-Repl). This requires exposing JIT output (e.g., symbol tables, memory addresses) to LLDB.
  - **Clang Integration**: LLDB understands Clang's type system, ensuring accurate variable inspection and a native debugging experience.
  - **LLVM IR Evaluation**: LLDB can inspect LLVM Intermediate Representation (IR), providing fallback insights if JIT execution fails.
- Two integration options exist:
  - **Attach LLDB**: Attach LLDB to the JIT process externally, requiring minimal changes to xeus-cpp.
  - **Embed LLDB APIs:** Integrate LLDB's C++ APIs (e.g., lldb::SBDebugger) into xeus-cpp for tighter control, though this increases complexity.
- The simpler attachment approach, facilitated by lldb-dap, is recommended for initial implementation.

Lessons from xeus-python

- The xeus-python debugger provides a proven model:
    - DAP-Based: It uses DAP to communicate between the Jupyter frontend and debugpy, which translates requests into pydevd API calls.
    - External Process: debugpy runs separately, ensuring the kernel remains responsive during breakpoints.
    - Minimal Base Class: The xeus::xdebugger_base interface is lightweight, delegating most logic to the derived class (e.g., xpyt::debugger).
- lldb-dap as a Bridge
    - lldb-dap, originally developed for VS Code, implements DAP for LLDB. It:
    - Acts as a standalone DAP server, translating DAP messages into LLDB commands.
    - Requires no direct modification of LLDB, simplifying integration.
    - Ensures compatibility with Jupyter's DAP-based frontend.
- This makes lldb-dap a natural replacement for debugpy in the xeus-cpp architecture.

Proposed Architecture
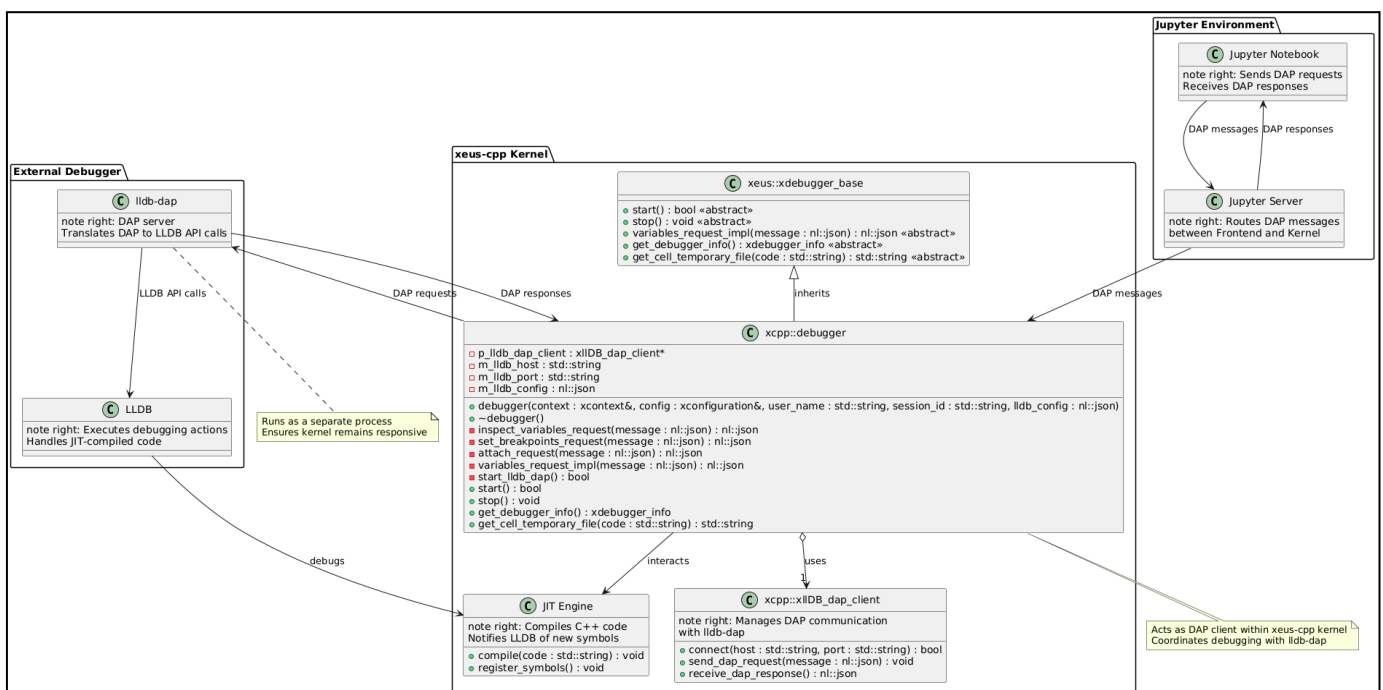
The Roadmap

- **First, we need to confirm that LLDB can attach to JIT-compiled code in Clang-Repl. I have already accomplished and demonstrated this successfully. Here's the proof:**
  - Video Demo.
  - This program takes user input (C++ code resembling Jupyter Notebook cell code) and compiles it into an executable (jitcode_with_cppinterop).
  - The input code is executed using CppInterOp's Cpp::Declare().
  - If plugin.jit-loader.gdb.enable is enabled in LLDB settings, symbols inside Cpp::Declare() are resolved, allowing debugging of the user's input code.
  - If the setting is disabled, the debugger skips over that part without resolving the symbols.
  - A temporary file (input_line_1) is created to store the original user input code, which serves as the source path.
  - For every code snippet we debug in Jupyter Notebook:
    - We need to compile the jitcode_with_cppinterop executable having multiple calls to run_code(code) function.
    - The debugger must run on this executable.
    - The source path should be retrieved using xcpp::get_cell_temporary_file(code) from xdebugger.hpp.
  - **OR**
  - We can club all the cells into one file, and pass the code in this file in the run_code(code) function. Why is this done?? It is explained in difficulty-2.

```cpp
// jitcode_with_cppinterop.cpp
#include "clang/Interpreter/CppInterOp.h"
#include <iostream>

void run_code(std::string code) {
 Cpp::Declare(code.c_str());
}

int main(int argc, char *argv[]) {
 Cpp::CreateInterpreter({"-g", "-O0"});
 std::vector<Cpp::TCppScope_t> Decls;
 std::string code = R"(
#include <iostream>
```

```cpp
void f2() {
 int a = 4;
 int b = 10;
 std::cout << b - a << std::endl;
 std::cout << "kr-2003" << std::endl;
}
void f3() {
 int a = 1;
 int b = 10;
 std::cout << b - a << std::endl;
 std::cout << "in f3 function" << std::endl;
}
f2();
f3();
int a = 100;
int b = 1000;

 )";
 std::cout << code << std::endl;
 return 0;
}
```

- **Secondly, we need to use LLDB-DAP to manage debugging. I have already set up debugging in VSCode using LLDB-DAP. The following video demo showcases how JIT-compiled code can be debugged using LLDB, LLDB-DAP, and VSCode. For our project, we need to integrate LLDB-DAP with Jupyter Notebook.**
  - Video Demo.
  - VSCode's launch.json for debugger.
  - The debugger is attached to the executable that I created above.

```json
{
 "version": "0.2.0",
 "configurations": [
     {
         "type": "lldb-dap",
         "request": "launch",
         "name": "Debug",
         "program": "/path/to/executable",
         "args": ["one", "two", "three"],
         "sourcePath" : ["${workspaceFolder}"],
         "cwd": "${workspaceFolder}",
         "initCommands": [
             "settings set plugin.jit-loader.gdb.enable on"
         ]
     },
 ]
}
```

- **LLDB-DAP Executable Running**
  - In xeus-python,

```
// call to debugpy.listen

code += "debugpy.listen((\'" + m_debugpy_host + "\'," + m_debugpy_port + "))";
```

    - debugpy runs on a specific port on the localhost.
  - I tried to replicate a similar approach using lldb-dap.
  - I was successfully able to run lldb-dap binary on a specified port and was able to perform debugging by sending debug requests which follow the DAP. Video Demo.
  - Created a python script which sends debug requests to lldb-dap server running on port 9999.

```python
init_request = {
    "seq": 1,
    "type": "request",
    "command": "initialize",
    "arguments": {
        "clientID": "manual-client",
        "adapterID": "lldb",
        "linesStartAt1": True,
        "columnsStartAt1": True
    }
}
send_dap_message(sock, init_request)
launch_request = {
    "seq": 2,
    "type": "request",
    "command": "launch",
    "arguments": {
        "program": "/Users/abhinavkumar/Desktop/Coding/Testing/test",
        "args": ["arg1", "arg2"],
        "cwd": "/Users/abhinavkumar/Desktop/Coding/Testing",
        "initCommands": [
            "settings set plugin.jit-loader.gdb.enable on",
        ]
    }
}
send_dap_message(sock, launch_request)
breakpoint_request = {
    "seq": 3,
    "type": "request",
    "command": "setBreakpoints",
    "arguments": {
        "source": {
            "name": "input_line_1",
            "path": "/Users/abhinavkumar/Desktop/Coding/Testing/input_line_1"
```

```
            },
            "breakpoints": [
                {
                    "line": 8
                }
            ],
            "lines": [8],
            "sourceModified": False,
        }
    }
    send_dap_message(sock, breakpoint_request)
    # Send configurationDone request
    config_done_request = {
        "seq": 4,
        "type": "request",
        "command": "configurationDone"
    }
    send_dap_message(sock, config_done_request)

    run_request = {
        "seq": 5,
        "type": "request",
        "command": "continue",
        "arguments": {}
    }
    send_dap_message(sock, run_request)
```

- ○ I was able to successfully apply breakpoints on specific lines, hit them and get stack-trace.
- ○ This experiment was performed on the same executable mentioned in point-1.

- **Implementation of xdebugger.cpp and xdebugger.hpp**
  - **Define debugger Class** – Inherit from xeus::xdebugger_base and declare LLDB-DAP client, configuration, and necessary functions.
  - **Implement Constructor & Destructor** – Initialize LLDB-DAP settings and ensure cleanup.
  - **Start LLDB-DAP (start_lldb_dap)** – Launch and configure LLDB-DAP for debugging. Here, we need to specify the executable on which the LLDB will run. Here, I propose spinning the lldb-dap executable on a specific port on the localhost.
  - **Handle LLDB Versioning** – Parse version (parse_lldb_version) and check feature support (check_version_features).

- **Debugger Lifecycle** – Implement start() and stop() to manage debugging sessions.
- **Utilities** – Provide debugger info (get_debugger_info), manage temporary source files (get_cell_temporary_file), and create an instance (make_cpp_debugger).
- **(Very Important)** xeus-python uses debugger::attach_request() to attach debugger to current running progress. This works for python because python-debugger doesn't need any executable to run. But to make our lldb work, we need to launch instead of attach. We need to attach lldb to a different process, i.e. our custom executable which has our JIT-compiled code. Therefore, instead of attach_request(), we need to make it launch_request(). But, this would require changes in xeus-zmq.

```
if (json_message["command"] == "attach")

{

    handle_init_sequence();

    m_wait_attach = false;

}
```

  - This is how the "attach" request is handled in xdap_tcp_client_impl::handle_control_socket().
  - We need to add "launch" request and accordingly change the handle_init_sequence().

```
register_request_handler("attach",   std::bind(&debugger::attach_request,
this, _1), true);
```

  - This is how debugger::attach_request is binded to attach request in xeus-python's debugger constructor.
  - We need to make our own custom debugger::launch_request and attach it to the "launch" request which we already have implemented in xeus-zmq.

- **Implementation of lldb-dap client(xlldb_dap_client)**
  - Define xllDB_dap_client Class – Inherit from xeus::xdap_tcp_client and handle LLDB-DAP communication.
  - Implement Constructor – Initialize the client with xeus::xcontext, xeus::xconfiguration, and DAP settings.
  - Destructor – Use default destructor for automatic cleanup.

- Handle Debug Events (handle_event) – Process LLDB-DAP messages and manage debugging state.
- Retrieve Stack Frames (get_stack_frames) – Fetch stack frames for a given thread ID and sequence number.

- **xeus_cpp_shell**
  - Create a new repository xeus_cpp_shell similar to xeus_python_shell and implement an XDebugger class. This class will manage variable tracking, debugging, and interaction with LLDB-DAP in Jupyter. It will filter and process variables, inspect them dynamically, and provide structured responses. The implementation will ensure efficient debugging support for C++ code in Jupyter Notebook.

- **Testing**
  - Creating test_debugger.cpp with debugger_client class.
  - It will mostly resemble the following structure.

```cpp
class debugger_client
{
public:

    debugger_client(xeus::xcontext& context,
                    const std::string& connection_file,
                    const std::string& log_file);

    bool test_init();
    bool test_disconnect();
    bool test_attach();
    bool test_external_set_breakpoints();
    bool test_external_next_continue();
    bool test_set_breakpoints();
    bool test_set_exception_breakpoints();
    bool test_source();
    bool test_next_continue();
    bool test_step_in();
    bool test_stack_trace();
    bool test_debug_info();
    bool test_inspect_variables();
    bool test_rich_inspect_variables();
    bool test_variables();
    bool test_copy_to_globals();
    void start();
    void shutdown();
```

```cpp
    void disconnect_debugger();

  private:

    nl::json attach();
    nl::json set_external_breakpoints();
    nl::json set_breakpoints();
    nl::json set_exception_breakpoints();

    std::string get_external_path();
    void dump_external_file();

    std::string make_code() const;
    std::string make_external_code() const;
    std::string make_external_invoker_code() const;

    bool print_code_variable(const std::string& expected, int& seq);
    void next(int& seq);
    void continue_exec(int& seq);

    bool next_continue_common();

    xeus_logger_client m_client;
};
```

Technical Difficulties

Difficulty - 1

- While performing above checks and experiments, I found that if function definition and function call are in the same cell, then the step-in call from the function call behaves normally, i.e. it steps-in directly in one call. But when they are in different cells, then it takes multiple step-in calls to step into the function definition from the function call.

- I think this is due to the fact that the execution is going through the intermediate libraries of CppInterOp/clang-repl. In vs-code, these intermediate addresses are marked as Unknown Sources.

- **When function definition and call are in the same cell**

```
1 location added to breakpoint 1

Process 93415 stopped

* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1

    frame #0: 0x00000001010841b0 JIT(0x101068000)`__stmts__9 at input_line_1:15:1
```

```
     12    std::cout << b - a << std::endl;
     13    std::cout << "in f3 function" << std::endl;
     14  }
→  15  f2(); // applying breakpoiutn here
     16  f3();
     17  int a = 100;
     18  int b = 1000;
(lldb) s
Process 93415 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step in
    frame #0: 0×000000010108400c JIT(0×101068000)`f2() at input_line_1:4:7
  1
  2    #include <iostream>
  3    void f2() {
→  4       int a = 4;
  5        int b = 10;
  6        std::cout << b - a << std::endl;
  7        std::cout << "kr-2003" << std::endl;
```

● **When function definition and call are in different cell**

```
Process 87843 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.8
    frame #0: 0×0000000100594068 JIT(0×100580000)`__stmts__0 at input_line_2:4:1
  1
  2    std::cout << "a = " << a << std::endl;
  3    std::cout << "b = " << b << std::endl;
→  4     f2();
(lldb) s
Process 87843 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step in
    frame #0: 0×00000001005940a8
→   0×1005940a8: adrp    x16, 0
    0×1005940ac: ldr     x16, [x16, #0×d8]
    0×1005940b0: br      x16
    0×1005940b4: adrp    x16, 0
(lldb) s
Process 87843 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step into
    frame #0: 0×00000001005940ac
→   0×1005940ac: ldr     x16, [x16, #0×d8]
    0×1005940b0: br      x16
    0×1005940b4: adrp    x16, 0
```

```
   0×1005940b8: ldr     x16, [x16, #0×f8]
(lldb) s
Process 87843 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step into
   frame #0: 0×00000001005940b0
→  0×1005940b0: br     x16
   0×1005940b4: adrp   x16, 0
   0×1005940b8: ldr     x16, [x16, #0×f8]
   0×1005940bc: br     x16
(lldb) s
Process 87843 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step into
   frame #0: 0×000000010057c000 JIT(0×100560000)`f2() at input_line_1:3
 1
 2     #include <iostream>
→ 3     void f2() {
 4         int a = 4;
 5         int b = 10;
 6         std::cout << b - a << std::endl;
 7         std::cout << "kr-2003" << std::endl;
```

- Here, I observed that the intermediate functions do not debug symbols, i.e. they don't have any function names. That's why the debugger in vs-code shows them as Unknown Sources.
- I tried following settings in lldb to bypass these functions, but was not successful.
  - settings set target.process.thread.step-avoid-regexp
  - settings set target.process.thread.step-avoid-libraries
  - settings set target.process.thread.step-in-avoid-nodebug true

Solving Difficulty - 1

I got to the conclusion that I want my debugger to step in/hit breakpoints only in functions that are in my code. Following are different approaches to tackle this problem.

- **Understanding Just My Code (JMC)**
  - My first proposal is to implement the "Just My Code" feature in LLDB.
  - There is an issue on LLVM regarding this, but this hasn't been implemented yet.
  - This is a Microsoft Debugger feature announced here:

https://devblogs.microsoft.com/cppblog/announcing-jmc-stepping-in-visual-studio/.

- ○ JMC is a debugging feature that helps developers focus on their own code by automatically skipping over system or library code during debugging sessions. The feature works by:
- ○ Inserting calls to a special function (__CheckForDebuggerJustMyCode) at the start of compiled functions
- ○ Using global flags to indicate whether code is "user code" (1) or "system code" (0)
- ○ Having the debugger interact with this mechanism to skip stepping into non-user code
- **Running step-in in a loop**
  - ○ When a user steps in, then we can run step-in a while loop until it hits source code.
  - ○ It can be implemented easily with a simple loop of stepIn + stackTrace messages, until the source field of the stack trace response is known
  - ○ The only thing to be careful of is the request_seq of the response, that should be set to the sequence number of the initial stepIn request.
- **Single code file for all cells**
  - ○ Packing the code of all cells into one file and then use some kind of mapping from the current file to a particular cell and its line.

For now, I am inclined towards approach-2 to solve this problem. If I get some extra time, I would love to explore approach-1.
But, we will see how approach-3 will solve this difficulty and the upcoming difficulty-2.

Difficulty - 2

Stepping-over(next) on a function call declared in previous block/cell does not work properly.

- Let's say a user defines multiple functions(f1, f2 and f3) in a single cell. These functions are called from the same cell one after another. If the step-over(next) command is used on one function, then it flawlessly goes to the next function call.
- But let's say a function call is made from another cell, then it takes multiple step-over(next) commands for the line to execute. Plus, it does not stop at the next function call. But it does execute it without any

bugs/errors.

In the given ,

- input_line_1 shows the use-case explained in point-1. Breakpoint is at f1() call.
- input_lint_2 shows the use-case explained in point-2. Breakpoint is at f3() call.

```cpp
// input_line_1

#include <iostream>
void f1() {
 std::cout << "in f1 function" << std::endl;
}
void f2() {
 std::cout << "in f2 function" << std::endl;
}
void f3() {
 std::cout << "in f3 function" << std::endl;
}
std::cout << "In codeblock 1" << std::endl;
int a = 100;
int b = 1000;
f1();
f2();
f3();
```

```cpp
// input_line_2

std::cout << "In codeblock 2" << std::endl;
std::cout << "a = " << a << std::endl;
std::cout << "b = " << b << std::endl;
void f4() {
 std::cout << "in f4 function" << std::endl;
 f2();
}
f3();
f2();
```

- But, let's say in input_line_2, if instead of using step-over(next), I continuously use step-in on f3, then it goes to the next function call(i.e. f2).
- This is shown in this .
- The problem is how stepping-in inside the last call of f3 directs it to the call of f2 in input_line_2.

Solving Difficulty - 2
- Now things are getting quite complicated for multiple cells.
- Both step-in and step-over are showing weird behaviour while debugging.
- I propose to have a single file for all the cells/blocks of code. And we have an abstract class that handles the mapping of block-line while debugging. Since, all things are working smoothly and fine for a single

cell, this seems a reasonable and solid approach.

Project Timeline & Milestones

This section provides a basic overview regarding how I plan to utilise the Pre-GSoC, GSoC contributing and the Post GSoC period .

● Pre-GSoC / Application Review Period

1. I would try to get a better grasp of the codebase , brush up on some C++, llvm and lldb essentials. I would like to surf through LLVM's documentation for the same.
2. Trying different experiments, finding out different difficulties. Also, continuously think about how these difficulties can be tackled.
3. Keep contributing to xeus-cpp and CppInterOp and get my PRs merged.

● May 8 - June 1(Community Bonding Period)

1. Regular communication with the mentors will be maintained throughout this duration to formulate a comprehensive document that outlines the implementation plan.
2. I would set up a blog post which would then be updated on a weekly basis.
3. I also intend to begin working on the project ahead of schedule, potentially during the final week of the community bonding period, to gain a head start.

● Week 1 - 3

1. Implementing the decided fix for difficulty-1, i.e. need for multiple step-ins if function call and definition are in different cells.
2. Implementing the decided fix for difficulty-2, i.e. Stepping-over(next) on a function call declared in the previous block/cell does not work properly.

● Week 4 - 6

1. Start implementing classes and functions for debugger class. For now, the whole implementation will not be needed. Just some basic templates.
2. Start implementing classes and functions for DAP client for LLDB(i.e. xlldb_dap_client).

● Week 7 - 9

1. Integration of LLDB-DAP client with jupyter debugger frontend.
2. Create a new repository xeus_cpp_shell similar to xeus_python_shell and implement an XDebugger class. This class will manage variable tracking, debugging, and interaction with LLDB-DAP in Jupyter.

● Week 10 - 12

1. Implementation of tests for debugger.
2. I would start framing a document for the final GSoC submission and complete any pending blogs from past weeks.

● Post GSoC Period

1. Complete any unfinished work on Pull Requests associated with the proposed ideas that may have been delayed due to obstacles encountered during the process.
2. Create an issue report that provides an overview of the current project status, including completed tasks, and outlines the roadmap for future development. This issue will serve as a reference for potential contributors interested in working on the project.
3. Upon the successful completion of my GSoC project, I would like to join the Compiler Research team as a full-time contributor.

● Time Commitment

1. I can positively dedicate 30 hours to my project on a weekly basis. On a need basis I wouldn't mind scaling the input hours up. I will inform my mentor in advance if any personal or miscellaneous work causes me to miss a deadline or weekly meeting. I will also provide a timeframe during which I will dedicate extra time to complete the work.
2. I plan to finish the Gsoc project with almost a week of buffer . This will give me more time to address anything left or give more importance to any issue that demands more time and debugging. or adapt to unexpected technical challenges that may necessitate slight deviations from the initial plan. It will also enable me to thoroughly document my work and rigorously test the newly added features to ensure their quality and functionality. This extra time will also enable me to thoroughly document my work and test the newly added features in their entirety.

I would like to take this opportunity to express my gratitude to those who have provided me with invaluable support and guidance over the past few months. In particular, I am deeply thankful to Anutosh (@anutosh491) for his mentorship—reviewing my PRs, helping me understand the project, and

guiding me through various challenges through our discussions over text and video calls. I would also like to thank Vipul (@Vipul-Cariappa) for assisting me with certain issues along the way.