



sugar labs

Project Info:

Title: Pippy Debugger

Project Length

175 hours

Difficulty

Medium

Coding Mentors

[Walter Bender](#) [Ibiam Chihurumnaya](#) [Kshitij Shah](#)

Applicant Info:

Name: Ahmad Abdul Rehman

Email: ahmadtarqx2003@gmail.com

Github: <https://github.com/ahmadtariq1>

Kaggle: <https://www.kaggle.com/ahmadtariq1>

First Language: Urdu

Location: Pakistan

Timezone: (UTC + 05:00)

University: FAST NUCES LHR

Program: Bachelors in Computer Science

About me

Contributions and Past Projects

I will be adding this shortly.

Pippy Debugger: LLM-powered Debugging Assistant for Pippy Activity

Problem Statement

“Many LLM programs for coding are almost exclusively marketed as "helping you write code for you". However, we believe that LLMs can also assist learners to *debug their code*. This project proposal is to create an LLM-powered debugger for Pippy, the Sugar Activity for creating code in Python.

The proposed Pippy Debugger integrates with the existing Pippy Activity. The LLM-powered debugger should be able to read a learner's code and offer suggestions for improvement when prompted. It should also help engage the learning in a conversation about how to discover where to look to find bugs and how to think about resolving them -- in other words, take the learning on a debugging journey as opposed to just spoon-feeding a solution. And

since we work with youth, we need to make sure that the debugger's output is age-appropriate. The Pippy interface will also be updated to expose the new feature to a user”

Project Overview

The Pippy Debugger project aims to create an innovative LLM-powered debugging assistant integrated with Pippy, the Sugar Activity for Python programming. Unlike conventional LLM coding tools that focus primarily on code generation, this debugger will emphasize educational debugging experiences, helping learners identify and fix errors while developing critical problem-solving skills. The system will guide students through the debugging process rather than simply providing solutions, making it an effective learning tool for Sugar's educational environment.

Impact on Sugar Labs

This project will significantly enhance Sugar Labs' educational toolkit by:

1. Providing an accessible debugging assistant that supports learning-by-doing
2. Making programming more approachable for young learners by reducing frustration with bugs
3. Teaching systematic debugging methodologies
4. Enabling more independent exploration and problem-solving
5. Demonstrating innovative applications of AI in educational contexts

Technologies

- **Python:** Core programming language for development
- **FastAPI:** For building the debugging API service(tentative)
- **Llama-3.1-8B:** Primary LLM model for debugging assistance
- **Docker:** For containerization and deployment consistency
- **AWS:** For hosting the model and API
- **Sugar Activity Framework:** For integration with the Pippy environment

Project Breakdown

1. Model Selection and Evaluation

Why Llama-3.1-8B? I tested multiple models including CodeLlama, CodeJemma, and few others, but Llama-3.1-8B worked the best for our debugging assistant. The other models had problems understanding logical errors, but Llama-3.1-8B had no issues understanding context and identifying errors(to my current testing). I've attached images showing sample debugging sessions with the model that demonstrates good ability to identify problems and explain them in a way that's helpful for young learners.

Code

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

# Testing the function
print(is_prime(2)) # Expected: True
print(is_prime(3)) # Expected: True
print(is_prime(4)) # Expected: False
print(is_prime(25)) # Expected: False
print(is_prime(29)) # Expected: True
```

Response

You've written a function called `is_prime(n)` that checks if a number is a prime number. You've also tested the function with some numbers.

Looking at the function, can you explain to me why the comment `# Expected: False for print(is_prime(25))` might not be what we expect?

Why do you think `25` isn't a prime number? Can you use a simple explanation, like "a prime number has only these special divisors" or "all the other numbers divide into it evenly"?

If you are thinking about divisors, then the line `for i in range(2, n)` might be important. But what does this line do? Can you use an analogy or a simple statement to explain what's happening in the loop?

Remember, I'm here to help you think about this problem step by step, rather than giving you the answer directly!

```
age = "25"
print(age + 5) # Cannot add a string and an integer
```

You've written a code that tries to print the result of adding the string "25" and the number 5.

Can you tell me what you expect the output to be? Do you think `print(age + 5)` will print "30", since that's what you get when you add the numbers 25 and 5?

Take a closer look at the code, and see what the `age` variable is equal to. Is it a number, or is it something else?

When you try to run this code, what happens? Do you get an error message, or does it print something unexpected?

Let's think about the types of things we can do with the `age` variable. We can print it out, like we're doing now. Or we can try to add two things together. Does that mean we're adding the number 5 to, um, what's in the `age` variable?

Think about it a sec - what's the problem with this line `print(age + 5)`? Why can't we add the `age` variable and the number 5 together?

```
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 1)

# Expected: 0, 1, 1, 2, 3, 5, 8
print([fibonacci(i) for i in range(7)])
```

You've written a function called `fibonacci(n)` that tries to calculate the Fibonacci sequence. You're using a technique called recursion, which is when a function calls itself.

Can you tell me what you expect the output to be when you run the code `print([fibonacci(i) for i in range(7)])`? Does it match the expected output you see there, something like 0, 1, 1, 2, 3, 5, 8?

Take a closer look at the function `fibonacci(n)`, and see if you can spot what might be wrong. Notice that if you pass in a number

greater than 1, the function calls itself twice, like `return fibonacci(n - 1) + fibonacci(n - 1)`. That doesn't seem quite right...

Remember how we always talk about debugging and trying to figure out what went wrong? Think about what would happen if we passed in, say, the number 5 to the `fibonacci(n)` function.

Notice how the Fibonacci sequence is supposed to work: 0, 1, 1, 2,

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left < right: # Should be left <= right
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

print(binary_search([1, 3, 5, 7, 9], 5))
```

You've written a function called `binary_search(arr, target)` that tries to find the index of a target value in a sorted array. You're using a technique called binary search, which is when we find the middle element and decide whether to search the first or second half.

Can you tell me what you expect the output to be when you run the code `print(binary_search([1, 3, 5, 7, 9], 5))`? Does it match the expected output you see there, something like 2?

Take a closer look at the condition in the while loop `while left < right:`. Can you spot what's wrong with it? Remember how binary search typically works?

Do you know how binary search decides which half of the array to search next? It usually checks the middle element and decides whether the target value is smaller or larger than that. If the target value is larger, it searches the second half of the array.

In your code, what happens when the target value is equal to the middle element?

Note:

- These samples are shown here to prove that the model is **sufficiently capable** of understanding logical errors.
- This is not to shown to demonstrate how debugger would respond in our project.
- It was also said in the matrix chat that sugar devs have tested llama the most.

Ibiam

The idea is for the LLM to help debug, they generally are good at pointing out any of the errors you've mentioned. We could leverage a model that has that capability rather than thinking of an additional lexer.

- It has been verified in depth that this model has no problem with identifying lexical, syntax and semantic errors. This was tested because some small models do actually fail occasionally in finding semantic errors.

Llama-3.1-8B Specifications:

- Provider: Meta

- License: Open source
- Context Window: 128k tokens
- Size: 8B
- Key Benchmarks:
 - Reasoning (MMLU): 73%
 - Coding (HumanEval): 72.6%
 - Advanced Coding (EvalPlus): 72.8%
 - Math Word Problems (GSM8K): 80.5%
 - Instruction Following (IF-Eval): 87.5%

The model is good at coding tasks (72.6% on HumanEval) which proves that it can be a good debugger tool.

2. Prompt Engineering and Testing

- Refining Prompt:
 - Adding Details for Relevance
 - Adopting a Role for Expertise(highly useful)
 - Specifying Steps for Clarity
 - Providing Examples(few-shot prompting)
- Work on creating specialized prompts that:
 - Use age-appropriate explanations for different learning levels
 - Focus on guiding instead of just giving direct answers
 - Teach good debugging methods
- Test everything thoroughly by:
 - Preparing most common codes dataset
 - Testing on extreme variation of code
 - Comparing results with top models like Claude and Gemini 2.5
 - Testing with lots of different common programming errors

Output Expectations of Debugger



walterbender

M MUJTABA AHMAD: We want to introduce the kids to the idea of debugging and help guide them. Not to do it for them.

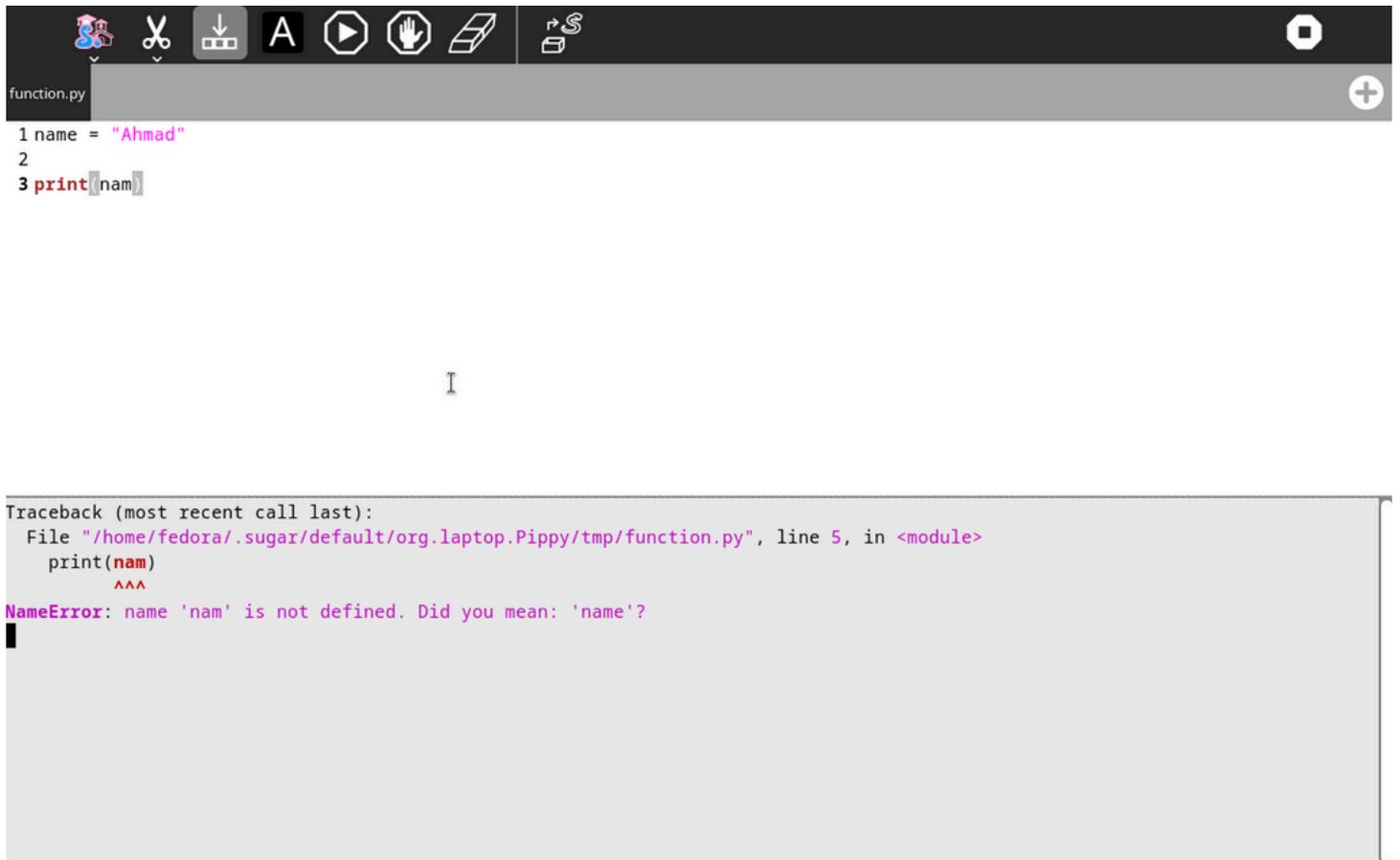
Code	Debugger Response
<pre>name = "Pippy" print("Hello " + name + "!") x = 5 + 3 tot@l = x * 2 print("Total is " + tot@l)</pre>	<p>"Hey, coder! It looks like I got a bit confused reading your program. Something in there doesn't quite fit how Python likes things to be written. Want to take a peek at your code and check for anything that looks unusual—like a misspelled word or a funny symbol? See if everything looks just right!"</p>
<pre>age = 10 if age > 5 print("You're older than 5!") else: print("You're 5 or younger!")</pre>	<p>"Oopsie! Your code tripped me up a little. It's like I couldn't figure out how to follow your instructions because something's missing or out of place. How about looking at how you've set up your 'if' part? Check if all the pieces are there to help me understand what to do!"</p>
<pre>num_cookies = 12 num_friends = 3 cookies_per_friend = num_cookies + num_friends # Meant to divide, print("Each friend gets " + str(cookies_per_friend) + " cookies!")</pre>	<p>"Hmm, your code ran smoothly, but the answer might not be what you had in mind! It's like the computer did something different from your plan. Think about what you want to happen with your numbers. Are you combining them the way you imagined? Try testing with some small numbers to see what's going on!"</p>
<pre>temperature = 25 if temperature > 20: print("It's cold outside!") else: print("It's warm outside!")</pre>	<p>"Your code worked, but the result feels a bit funny, doesn't it? It's like the computer followed your directions, but they might not match what you were thinking. Let's pretend you're the computer—walk through what happens with your 'if' part. Does it say what makes sense to you? Try a few different numbers to check!"</p>
<pre>score = 10 bonus = "5" total = score + bonus print("Your total score is " + total)</pre>	<p>"Hmm, your code tried to run, but I hit a bump! It seems like I couldn't figure out how to mix some of your pieces together. Think about what you're combining—like numbers or words. Are they playing nicely together? Try checking if everything matches up the way you want!"</p>

3. Two-Phase Debugging Architecture

Phase 1: Guided Debugging

- Help learners find problem areas through guiding questions
- Give hints without direct solutions
- Teach systematic debugging approaches
 - Verify Code Elements
 - Check Structure

- Confirm Data Fit
- Test with Examples
- Trace Logic
- Make students think critically about how their code works
- The current output terminal displays errors in a very beginner friendly manner
- Taking advantage of this, the debugger will try to relate error with debugging approach, e.g in case of NameError-->verify code elements



Phase 2: Active Assistance

"Active Assistance" starts after you try guided debugging once (tentative), steps in if you're totally lost, and helps in debugging with more ease, making the project cooler.

- Marking specific lines that might have errors with arrows
- Suggest where to add print statements to see variable values
- Recommend debugging strategies like:
 - Adding print statements in strategic places(such as before and after if-else conditions)
 - Checking variable values
 - Tracing how the program flows
 - Checking variable types

An image is attached which gives an idea of how it would look like

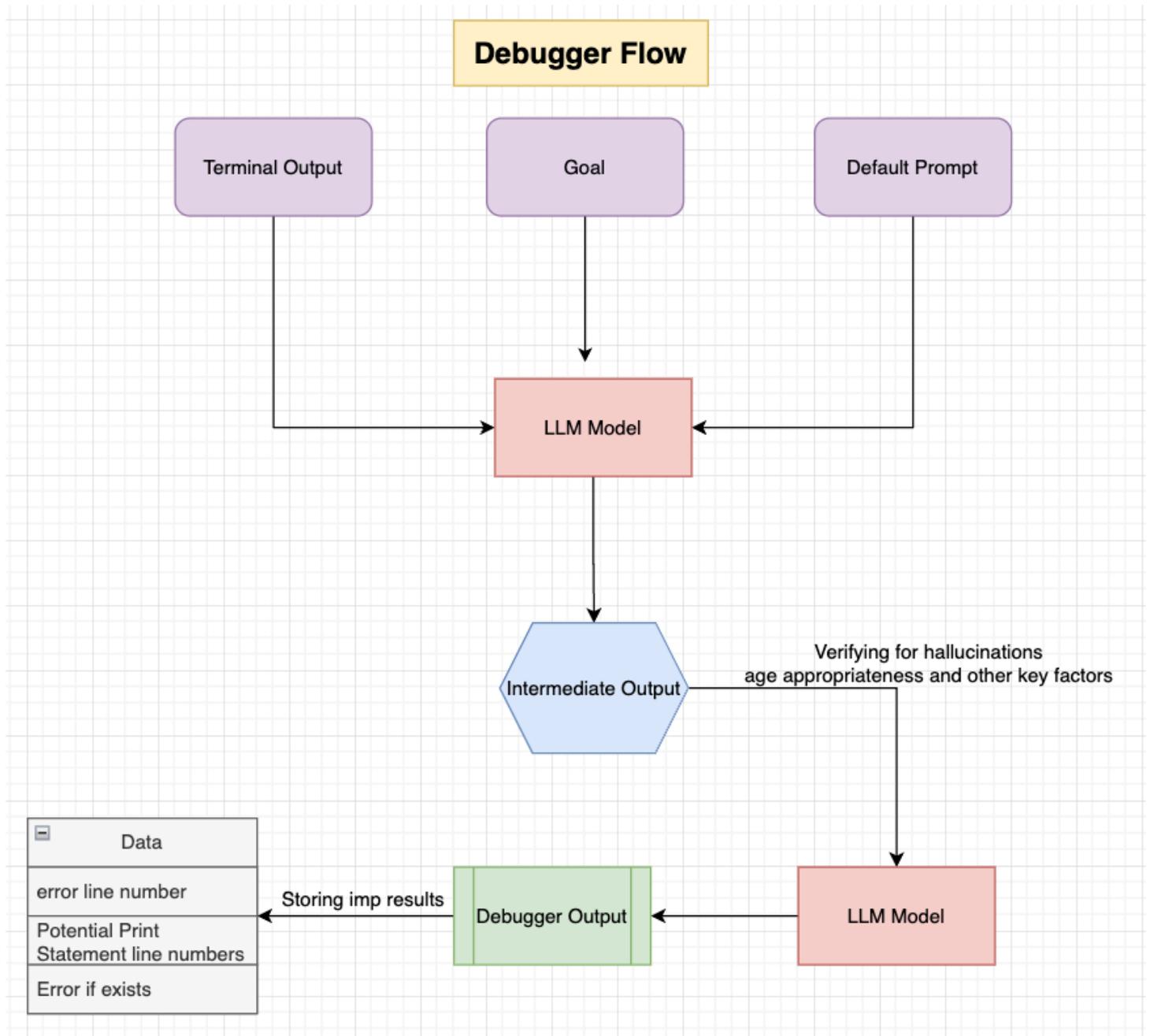


Important note:

We'll not be implementing a debugger chat bot due to the following reasons:

- It can distract students with side conversations.
- It might give direct solutions instead of guiding.
- Responses may not always be age-appropriate.
- It's hard to control what the LLM says in an open-ended chat format.
- It could overwhelm young learners with too much text.
- It's harder to keep the experience focused and on-task.

Debugger Flow Methodology



Goal statement is optional but this could really help in solving problems where there is no error but the code is not giving its intended output.

A simple example :

```
for i in range(1,5):  
    print("Hello Sugar")
```

output:

Hello World

Hello World

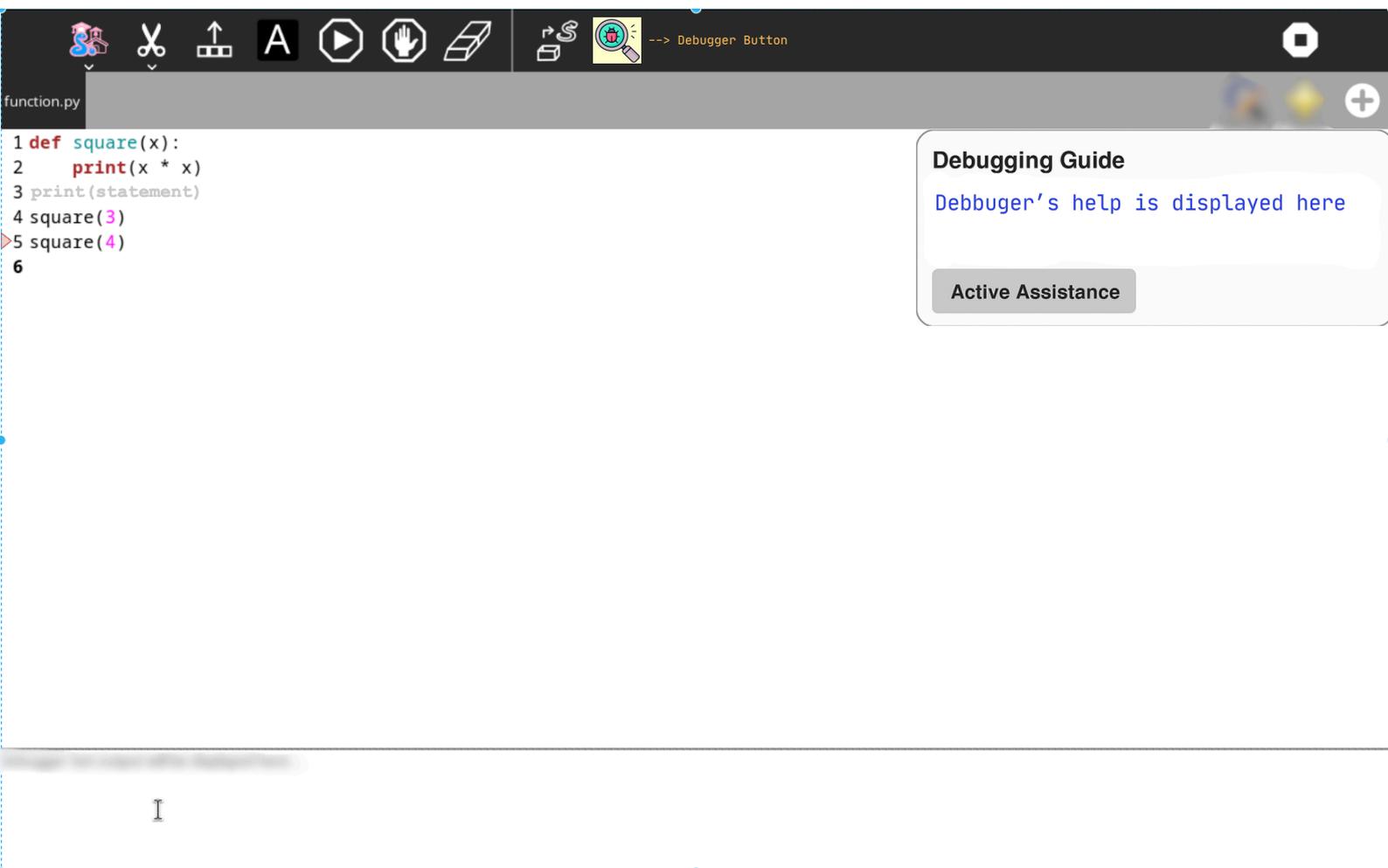
Hello World

Hello World

This code prints 4 times, but the coder intended it to print 5 times. This kind of intention or requirement is impossible for an LLM to detect on its own. To address this, we'll add a feature where users can specify their goal as context. This way, the debugger will have the necessary information about what the code is supposed to do and can provide more helpful assistance.

4. UI Integration with Pippy

- I've designed a sample debugging interface for Pippy and attached sample UI mockup showing how it could look. The interface includes:
 1. A debugging toolbar with two different debugging modes
 2. Visual indicators for problematic code
 3. A chat interface for debugging conversations
 4. A system for marking lines that might have errors
- How students will use it:
 1. Student gets an error in Pippy
 2. Clicks the Phase 1 icon (guidance mode).
 3. Is provided debugging help by the debugger about the problem in a conversational tone
 4. It is **important to note** that we'll not be implementing a chatbot because that takes away the key point of student learning. Students might ask it indirectly about the exact solution, and often when we let models operate in an open-ended way, they can hallucinate and produce messages that aren't age-appropriate. Instead, our approach will be more structured and focused specifically on debugging guidance. The two-phase system will provide targeted support within the specific context of the student's code, rather than allowing free-form questions that might let them bypass the learning process
 5. If needed, clicks the Phase 2 icon (active assistance) for specific help
 6. Tries the suggestions and learns debugging methods
 7. Print statement shown in important places similar to the style of how pycharm/colab displays in assisted coding mode. I am aware that this feature is difficult but this would make the activity look really refined.



Print Statement is not actually written but displayed

1. Line marked with arrow to indicate potential bug: (optional) because it is similar to providing the exact solution

Another style of displaying icons matching the current style tone:

5. Static and Dynamic Analysis Integration

- I'll implement static code analysis to find:
 - Syntax errors
 - Common mistake patterns
 - Style issues that could cause problems
- I'll also add dynamic analysis if static analysis isn't enough:(Most probably not needed for our debugging scope)
 - Trace how the code executes
 - Monitor variable values
 - Find runtime problems
 - Send this information to the LLM for better context

6. API Development with FastAPI

- **Endpoint Design:**
 - Create API endpoints for the debugger
 - Make endpoints for code analysis
 - Design a system to manage conversations
 - Make sure communication between Pippy and the backend is secure

- **Backend Integration:**
 - Connect the Llama-3.1-8B model
 - Build the debugging logic into the API
 - Create ways to report and analyze errors
- **Performance Optimization:**
 - Cache responses for common errors so it's faster
 - Make the model run as quickly as possible
 - Create efficient code analysis
 - Find the right balance between local processing and using the LLM

7. Deployment Architecture

- **Cloud Deployment (AWS):**
 - Set up the model on AWS
 - Make sure API access is secure
 - Add monitoring and logging
 - Making it as inexpensive as possible
 - Make it automatically scale based on usage

Note: I realized local deployment isn't feasible for this project because the model needs more resources than what will be available on x0 laptops and typical local machines, especially considering the hardware constraints of the environments where Sugar is commonly used.

8. Documentation and Educational Resources

- I'll create thorough documentation:
 - How to install and set up everything
 - API documentation
 - Usage examples and tutorials
 - Guides on debugging methods
- I'll also make educational resources:
 - Tutorial video of how to use the debugger (for teachers and parents reference)
 - Best practices for teaching with the tool

Timeline for the Project

(May 8 - September 1, 2025)

“Hofstadter's Law: It takes longer than you expect, even when you take into account Hofstadter's Law”

Time Period	Details
<p>May 8 - May 18 (total time: 16 hours) (Bonding and Recky period)</p>	<p>Connect with mentors (Walter Bender, Ibiham Chihurumnaya, Kshitij Shah) to discuss project scope. Read Sugar Labs and Pippy documentation to further understand the environment. Run initial tests with Llama-3.1-8B and gathering code samples for testing dataset.</p>
<p>May 18 - June 2 (total time: 16 hours) Semester Final Exams</p>	<p>Designing complete documentation of model system flow, and discussing it with mentor.</p>
<p>June 3 - June 7 (total time:10 hours) Exams Ongoing</p>	<p>Finalizing the design of frontend interface.</p>
<p>June 8 - June 15 (total time 25 hours) Semester ends here, fully available for GSOC 😊</p>	<p>Initially running model locally(yes my system is capable of that) and then testing Llama-3.1-8B with a variety of code samples (simple loops, logic errors, etc.) to verify its capability hold up for debugging. Compare it against the best models out by then to confirm that it outperforms them in understanding logical errors and context(debugging), as noted in our tests. Start prompt engineering/Refining: aforementioned in detail.</p>
<p>June 16 - June 23 (total time 25 hours)</p>	<p>Implementing debugger_flow method, mentioned above.Creating testing dataset, then perform edge case testing.Evaluation of results and refining.</p>
<p>June 24-June 29 (total time 25 hours)</p>	<p>Implementing two phased structure: Debugging Guide Active Assistance</p> <p>Code for debugger model should be complete by now!</p>
<p>June 30 - July 6 (total time 25 hours)</p>	<p>Developing API using FastAPI,Integrate</p>

	Llama-3.1-8B into the API. Test the API with simple sugar activity by submitting sample code and checking response speed/accuracy. At this moment, we'll not be implementing proper interface for debugger.
July 7 - July 18 (total time 35 hours)	Prepare a working prototype (debugging logic + API) for the midterm evaluation (July 14-18, submit by July 18, 18:00 UTC). Incorporate mentor feedback post-midterm to refine the system. Midterm Evaluations! 😄
July 19 - August 3 (total time 25 hours)	<ul style="list-style-type: none"> • Complete Debugger interface coding • Integrating it with pippy activity • Testing for bugs.
Aug 4 - Aug 11 (total time 25 hours)	Deployment Phase
Aug 12 - Aug 19 (total time 25 hours)	Detailed Feedback with mentors, refining based on feedback
Aug 19- Sep 1 (total time 25 hours) Semester Restarts!	Final Run! Package the project: bundle code, docs, and resources into a clear submission. Write a final report summarizing the work (two-phase system, UI integration, educational impact). Address any last-minute mentor feedback and ensure everything aligns with Sugar Labs' goals. Prepare for mentor final evaluations.

The time mentioned indicates the maximum time I'll be able to provide in the respective time period.

Expectations

1. A working LLM-powered debugging assistant integrated with Pippy

2. A two-phase debugging approach that focuses on learning
3. An improved Pippy interface with debugging features
4. Good documentation for users and developers
5. Educational resources for teaching debugging

I'm really excited about this project because it's not just about making a tool that fixes code, but one that teaches students how to debug their way out and learn to look at their code with a debugger's view. By focusing on guidance rather than just giving solutions, the Pippy Debugger will help Sugar Labs' educational mission while giving practical programming help to students worldwide.

Sidenote: Many experts think there will be a big need for people who are good at debugging, because AI is creating complicated code that often needs human help to work properly.(pretty ironic)

Post Gsoc Plans

I've been interested in SugarLabs for over a year, but setting up the development environment used to be super hard for me. The only code changes I could make without getting "failed to start" errors were just print statements 😊. I really like the concept of small specific activities because as an open source contributor, big projects can feel overwhelming, but coding complete activities gives you motivation and freedom to work on what interests you. Since I recently took an AI course, I'm excited about creating activities with cool algorithms like a 2x2 Rubik's cube solver. I've already talked about this with Sugar developers in the mailing list. Another activity I'd love to make would be to implement minimax for tic-tac-toe. These algorithms are perfect for basic activities like these.

Final Note:

A lot of the proposal details are up for discussion with the mentor, including model selection, user interface design, and deployment strategy. This document mainly aims to show a thorough understanding of the project and current best implementation approaches. More subjective elements like the interface design will be finalized at a later stage.