CodePanda-AI: A Case Study in Developing an Intent-Aware Tutor using Open-Source LLMs

Author: Harshit Mishra

1. Abstract

With the increase in demand of general purpose AI assistant tools presents a huge paradox, while these models are capable of providing instant solutions, they often undermine the development of the human brain in critical problem solving situations. These, while not showing signs in the short term, lead to devastating results in the long term growth of a person. This paper directly tackles this issue within the domain of programming education, where the goal is not just to obtain a correct answer, but to develop the skills required to produce one. For this end, I document the design and implementation of `CodePanda-AI`, an intent-aware tutor engineered to foster, rather than degrade, the learning process. My approach rejects the 'solution-first' paradigm, instead leveraging a local, open-source Large Language Model (LLM) to create a tool that is accessible, private, and easily controllable. The methodology centered on building a functional prototype and employing an prompt engineering process to constrain the model's behavior, transforming it from a solution-provider into a Socratic guide. The key result is a functional Streamlit application that successfully analyzes a student's Python code against their stated programming goal, consistently providing high-quality hints that empower them to discover the solution on their own, thereby demonstrating a viable and superior approach to integrating AI into education.

2. Introduction: The Case for Open-Source AI in Education

2.1 The Problem with Conventional Tutoring

The rise of powerful, large-scale AI models, such as OpenAI's GPT-4 and Anthropic's Claude, has changed the landscape of software development. Integrated into code editors and offered as standalone personal assistants, these tools can generate code, identify bugs, and explain complex algorithms with remarkable proficiency. However, when applied to an

educational context, their core design philosophy is optimizing for the fastest path to a correct solution which in hindsight creates a significant academic deficit. For a student learning to code, the process of tackling a bug, tracing the code's core logic, and experimenting with solutions is not an obstacle to be bypassed, it is the primary mechanism through which deep, resilient understanding is built.

Conventional AI tutors, by providing complete, corrected code blocks,unintentionally encourage a pattern of dependency on these models. Students may begin to depend on AI, leading them to bypass the cognitive effort necessary for acquiring skills. This "solution-first" approach prevents them from developing the crucial self aware skills of debugging, forming hypotheses about errors, testing those hypotheses, and learning to recognize common issues. The frustration built in this process is not a flaw in the learning journey but a feature. By removing it, AI tools risk producing programmers who can use solutions but cannot create them independently when faced with novel problems.

2.2 The Open-Source (FOSS) Solution

A better approach for building educational AI tools comes from the principles of Free and Open Source Software (FOSS), which is the core mission of FOSSEE. Using a local, open-source AI model has three major advantages over using a tool like ChatGPT.

1. **Accessibility:** An open-source model can run on a person's own computer. This means it works without an internet connection and doesn't require expensive API keys. This makes learning available to everyone, regardless of their financial situation.
2. **Privacy:** When you use a commercial AI tool, you have to send your code to their servers. A local model keeps all the student's work private on their own machine, which is much more secure. No need for you to risk your data.
3. **Control:** This is the most important advantage. Proprietary models are like "black boxes", where you can't change how they work. With an open-source model, I can have complete control. I can carefully engineer its behavior to make it a Socratic tutor that asks guiding questions, instead of a simple assistant that just gives answers.

2.3 Project Introduction: CodePanda-AI as Proof of Concept

This paper introduces "`CodePanda-AI`", a working application I built to prove that this FOSS-aligned approach is not just a theory, but something that can be practically achieved. This project is a case study showing how I took a standard open-source AI model and, through a careful process, turned it into an effective educational tool for students to study with.

3. My Method: Building and Refining the Tutor

The idea behind building this tutor was inspired by a classic, time-tested technique in software engineering: **Rubber Duck Debugging**.

> *""The principle is simple yet profound: a developer, when stuck on a problem, explains their code line-by-line to an inanimate object, like a rubber duck. In the process of articulating the problem, they often discover the solution themselves.""*

My project is built on this very foundation. **`CodePanda-AI` is designed to be the ultimate interactive rubber duck.** It doesn't just sit there silently, it listens to the student's goal and code, and then asks a single on point question to guide their thought process, accelerating that moment of self-discovery. This methodology required a hands-on approach of rapid prototyping, testing, and iterative refinement to bring that concept to life.

3.1 The Starting Point: Hypothesis and Tools

My core idea was simple: I believed that a code-focused open-source model like **`DeepSeek Coder 6.7B`** could be controlled through smart instructions (prompt engineering) to act as a helpful tutor, The main goal was to make it compare a student's code to what they *intended* to do, and then ask a good question.

To build this quickly and effectively, I chose the used tools:

- **Python:** The natural choice for any AI project.
- **Streamlit:** I picked this for the user interface because it allowed me to build a clean web app very quickly without needing to write complex web code. This let me focus on the AI itself.
- **llama-cpp-python:** To run the AI model on my own computer, I used this library because it's highly optimized and makes it possible to run powerful models on normal hardware.

3.2 The Key Feature: Understanding the User's Goal

The most important design decision I made was to make the user's goal a **required input**. A standard code checker can tell you if your code has a syntax error, but it can't tell you if your code is *logically* wrong. For example, if your goal is to write a function that modifies a list, but your function accidentally creates a new list instead, the code will run without errors, but it will still be wrong because it didn't achieve your goal.

By forcing the user to state their intent, `CodePanda-AI` can perform a much deeper analysis and find these more important logical and conceptual errors.

3.3 Debugging the AI: A Story of Prompt Engineering

Getting the AI to behave correctly was the biggest challenge. Its default behavior was to be "too helpful." My journey to fix this involved several steps.

- **Failure #1: The AI Gives Away the Answer.** My first set of instructions was simple: "Give a hint, but don't give the answer." The model completely ignored this. Because it was trained on websites like Stack Overflow, where providing the full, correct code is normal, it would almost always just output the solution. This taught me that simple instructions are not enough to override the model's basic training.
- **Failure #2: The AI Starts Copying Its Instructions.** My next idea was to give the AI an example of a good hint. In my instructions, I wrote an "Internal Analysis" section to show the AI how to think. This led to a new, strange problem: the AI started literally copying my "Internal Analysis" text into its response to the user. It saw the example as a template to be copied, not a process to be learned.
- **The Solution: A Highly Structured Prompt.** The final breakthrough came when I completely restructured the prompt using clear, XML-like tags such as `<INSTRUCTIONS>` and `<EXAMPLE>`. This created a clear boundary between the rules the AI had to follow, the example it should learn from, and the actual problem it needed to solve. This structured approach finally worked, giving me consistent and reliable control over the AI's behavior.

4. Results: How CodePanda-AI Performs

The final application is a simple but powerful tool that achieves the project's original goals.

4.1 How it Works

The workflow is straightforward:

1. The user describes what they want their code to do in the "Your Goal" box.
2. They paste their Python code in the "Your Code" box.
3. When they click the button, the app sends their goal and code to the local AI model.
4. The model analyzes everything and generates a single, Socratic question, which is then displayed to the user.

4.2 Test Cases: Seeing it in Action

The success of the system is best shown with a few tests.

- **Test Case 1: The In-Place Modification Bug**
    - **User's Goal:** "This function is supposed to modify the list in-place to double each number."
    - **Buggy Code:**

```
def process_data(data_list):
    for item in data_list:
        item = item * 2
    return data_list “
```

- **CodePanda-AI's Hint:**

  “” *When you use a 'for item in data_list' loop, are you able to directly change the values in the original data_list, or would you need the list's index to do that?* “”

- **Analysis:** This is a perfect hint. It doesn't give the answer “ (for i in range(len(data_list)))”. Instead, it asks a question that forces the student to think about the core concept they are missing: the difference between iterating by value and iterating by index.

- **Test Case 2: The Infinite Recursion Bug**
  - **User's Goal:** "I want to write a Python function that counts down from a given number to 1."
  - **Buggy Code:**

```
def countdown(n):
    print(n)
    countdown(n - 1) # Missing a base case to stop
```

- **CodePanda-AI's Hint:**

  “” *A recursive function needs a 'base case' to tell it when to stop calling itself. What condition in your function prevents it from counting down forever into negative numbers?* “”

- **Analysis:** Again, the hint is excellent. It introduces the key vocabulary ("base case") and asks a question that makes the student think about their code's control flow, guiding them to the solution on their own.

5. FOSSEE Submission Requirements

This section contains the formal answers to the questions from the screening task, based on my experience building this project.

5.1 The Research Plan

My research plan was to evaluate the suitability of the open-source **DeepSeek Coder 6.7B** model for the task of providing Socratic hints. I chose a practical methodology which is to build a functional prototype. This allowed me to test the model's capabilities in a real-world scenario, focusing on its ability to understand user intent and follow complex, restrictive instructions.

Validation was performed through an iterative process. I created a set of test cases with common Python bugs and used them to evaluate the model's responses. When the model failed (for example, by giving away the answer), I analyzed the failure and refined my prompt engineering strategy. The project's success was validated when the final version of the application could consistently produce high-quality, pedagogically useful hints for a variety of different bugs.

5.2 Reasoning

- **What makes a model suitable for high-level competence analysis?**
  Through my experience building `CodePanda-AI`, I found that a model's suitability for this kind of teaching task isn't just about its size, but about three specific qualities. First, it needs to be excellent at **instruction-following**. My early tests showed that a model will default to giving away the answer unless it can precisely follow complex rules, especially "negative" rules like "do not write code." Second, it needs the ability to **reason about user intent**, which is why I made the user's goal a required input. The model had to be capable of comparing a plain English sentence to a Python script and identifying where they logically diverged. Finally, and most importantly, it must be highly **controllable**. I learned that this control wasn't a default feature, but something I had to build myself through a very structured and rigid prompt design.
- **How would you test whether a model generates meaningful prompts?**
  My method for testing was a practical, hands-on feedback loop, not a theoretical one. I started by defining "meaningful" as a prompt that was Socratic (it had to be a question) and that guided the student to the root cause of their error without giving them the solution. I then developed a small suite of test cases with different kinds of common bugs—syntax errors, logical errors, and deeper conceptual mistakes. I fed these to the model and evaluated its output against my definition. When a hint was not meaningful (like when it asked a generic question or gave away the answer), I treated it as a bug in my own system. I would analyze why the AI failed and then refine my instructions to it, making them stricter and clearer. This process of testing, analyzing the failure, and improving the prompt was how I systematically increased the quality and reliability of the hints.

- **What trade-offs might exist between accuracy, interpretability, and cost?**
  I encountered a few key trade-offs during this project. The most interesting one was between giving the AI a creative "persona" and its **accuracy** in following rules. My initial attempts to make the AI act like a cute panda resulted in unreliable and often incorrect responses. A more direct, rule-based prompt was far more reliable, so I had to trade the creative persona for accuracy and control. There was also a clear **cost** trade-off. By choosing to run a model locally, I made a conscious trade-off: I accepted the one-time hardware requirement for running the model on my machine in exchange for zero ongoing API costs. This also provided the huge benefits of privacy and the ability for the app to work completely offline, which is critical for an accessible educational tool.
- **Why did you choose the model you evaluated, and what are its strengths or limitations?**
  My choice of `DeepSeek Coder 6.7B Instruct` was strategic and based on three main factors that made it perfect for a FOSS-aligned educational project.

1. **It's Lightweight and Accessible:** As a 6.7B parameter model in a GGUF quantized format, it is lightweight enough to run effectively on consumer-grade hardware. This was a non-negotiable requirement for me, as it ensures the final application is accessible to students without needing powerful, expensive machines.
2. **It's Specialized for Code:** The model is pre-trained on a massive amount of source code, giving it a strong, built-in understanding of programming logic and syntax.
3. **It's an "Instruct" Model:** It is specifically fine-tuned to follow complex commands, which was the entire foundation of my prompt engineering strategy.

Its main limitation, which I discovered during testing, was its susceptibility to **"prompt leakage,"** where it would literally copy parts of the examples from my instructions instead of learning the process. However, this limitation became an opportunity. By redesigning my prompt to use a more rigid, XML-like structure, I was able to successfully overcome this issue, which in itself was a valuable engineering lesson in how to control and work with modern LLMs.

6. Conclusion and Future Work

6.1 Conclusion

The **CodePanda-AI** project is a successful proof-of-concept. It shows that local, open-source AI models can be carefully engineered to become powerful and effective educational tools. The key takeaway from my work is that through a process of iterative refinement and structured prompt engineering, a model's default "helpful" behavior can be controlled and shaped to serve a specific pedagogical goal, in this case, fostering independent problem-solving skills in students.

6.2 Future Work

While the project is successful, there are several ways it could be improved in the future:

- **Implement a Chat History:** Allowing the user to ask follow-up questions would make the interaction feel more like a real conversation.
- **Automatic Bug Classification:** The AI could first try to classify the bug (e.g., "Syntax Error," "Logical Error") and then use that classification to inform its hint.
- **IDE Integration:** The most powerful version of this tool would be as a plugin for a code editor like VS Code, providing hints directly in the student's development environment.

7. Appendix: How to Run Locally

To run the CodePanda-AI application on your own machine, please follow these steps.

Prerequisites

- Python 3.8 or newer
- Git

Step 1: Clone the Repository

```
git clone
cd CodePanda-AI
```

Step 2: Set up a Virtual Environment

On Windows:

```
python -m venv venv
venv\Scripts\activate
```

On macOS/Linux:

```
python3 -m venv venv
source venv/bin/activate
```

Step 3: Install Dependencies

```
pip install -r requirements.txt
```

Step 4: Download the Model

1. Download the `deepseek-coder-6.7b-instruct.Q4_K_S.gguf` model file from its Hugging Face repository.
2. Place the downloaded `.gguf` file in the root of the `CodePanda-AI` project folder.

Step 5: Run the Application

```
streamlit run app.py
```