FOSSEE Screening Task 3 Submission: CodePanda-AI

Author: Harshit Mishra

This document provides the required deliverables for the Python Screening Task 3, based on my development of **CodePanda-AI**, a functional prototype of an intent-aware AI tutor.

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: 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. The core idea was inspired by the classic software engineering technique of **Rubber Duck Debugging**, where a developer finds a solution by explaining their code to an inanimate object. My goal was to create the ultimate interactive rubber duck—an AI that listens to a student's goal and code and then asks a single, pointed question to accelerate that moment of self-discovery.

Validation was performed through a hands-on, 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 or copying its instructions), I analyzed the failure and refined my prompt engineering strategy. The project's success was validated when the final version of the application, using a highly structured, XML-tagged prompt, could consistently produce high-quality, pedagogically useful hints for a variety of different bugs, proving the model could be effectively adapted for this educational purpose.

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. 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.

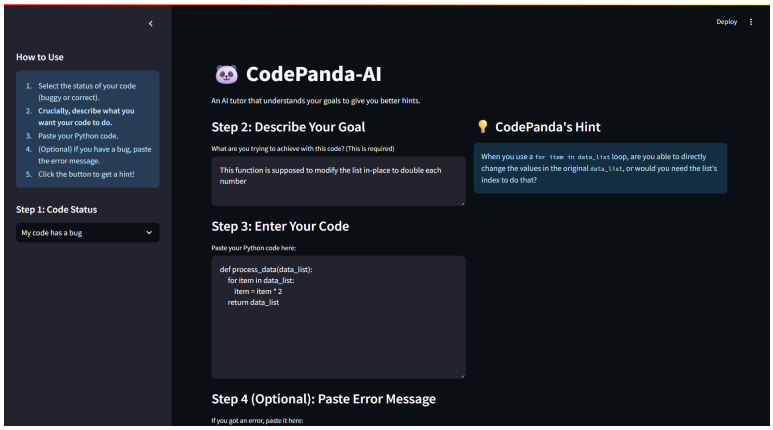
What trade-offs might exist between accuracy, interpretability, and cost?

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 FOSSEE-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

References

1) Deepseek ai
2) Llama-cpp-python
3) Streamlit