Agentic Based ClarifyCoder

1st Poojith Mendem
*C*ollege of Computing
Michigan Technological University
Houghton, USA
pmendem@mtu.edu

***Abstract*—Large language models (LLMs) are capable of generating correct and efficient code when the task is well-specified, but when the prompts are unclear, incomplete, or inconsistent, they often hallucinate and produce incorrect results. Here, I describe a new Agentic ClarifyCoder workflow that is centered on ClarifyCoder a fine-tuned model that indicates uncertainty and asks specific clarification questions before generating any code. Using the Two Agentic Workflow ClarifyCoder generates code for well-specified prompts, and it uses short clarification questions in cases of uncertainty to elicit missing user requirements. This strategy reduces errors, and increases overall reliability of LLM-based code generation.**

***Index Terms*—Large language models, code generation, ClarifyCoder,ambiguity detection, instruction-based fine-tuning, Prompt Engineering, HumanEvalComm, multi-agent workflow, langgraph, DeepSeekCoder**

I. Introduction

Code-focused large language models (LLMs) demonstrate strong performance when the problem statement is complete and precise. However, in real and situated software practice, requirements are often vague, inconsistent, or incomplete; human developers may pause and ask clarifying questions that lead to implementation, whereas many code LLMs by default produce code resulting in incorrect solutions. Recent work defines this gap in the ability to reflect on a "communication competence," understanding the measurable decay in performance on coding tasks when uncertainty is present while also motivating systems that identify ambiguity and seek clarifications before coding [1].

ClarifyCoder provides a solution to this gap; as a clarification-aware approach, The code model that is fine-tuned to recognize unclear prompts and request additional information before generating code. By combining the creation of synthetic ambiguity with instruction tuning, the model learns to ask when faced with an ambiguous specification and to implement the code when it understands the requirements, improving dialogue around clarification while preserving code capability. In our work, ClarifyCoder is instantiated using an open-code base model, DeepSeek-Coder, taking advantage of strong code generation capabilities with the addition of clarify-first behavior [2],.

We extend this idea by operationalizing the Two Agentic Workflow that alternates clarification and coding. This approch queries unspecified requirements directly and then produces final code, improving measured communication behaviors without degradation in task completion. Our system orchestrates two roles: (i) A reviewer role that verifies the problem statement is clear, if not, asks Clarification questions to the user, and (ii) an executor role which takes the requirements from the reviewer and prior context to execute the final code.

II. Related Work

A. Communication Benchmarks with Ambiguity

Early code-generation benchmark prioritized single-turn correctness, emphasizing a capable coder." The purpose of this benchmark is to seriously test the ambiguities, inconsistencies, and incomplete specifications in order to introduce communication benchmarks, *Communication Rate* and *Good Question Rate*. These two communication benchmarks determine if the model asks clarifying questions before coding[1]. This benchmark presents an LLM-agent characteristic (Okanagan) which structures multi-round clarification in prompt contexts before settling on the final code.

B. Clarification-Aware Fine-Tuning

Clarification-aware modeling seeks to make behaviors clearly learnable with data and objectives through a decision on when to "ask vs. answer". *ClarifyCoder* fine-tunes a base Deepseek Coder model using a mixture of ordinary programming problems and synthesized ambiguity-based variants to learn when to ask for requirements and when to code[2]. Reported ablation studies reflect a consistent trade-off between competent coding and tendency to ask questions of quality, demonstrating improved clarification behavior while preserving generation quality .

C. Procedural Clarification Frameworks

Beyond to fine-tuning, procedural frameworks will trigger clarification and adaptations at inference time. For example, *ClarifyGPT* detects whether the requirement is ambiguous and then prompts for different, targeted questions of the model with improved downstream quality of solutions[3], without further fine-tuning of instruction sets. These methods demonstrate that through structured two-phase "detect-then-ask" pipelines, off-the-shelf LLMs can raise pass-rates, together with the framework itself.

D. Agentic Strategies

As an agentic approach, clarification operationalizes and frames the clarification of requirements as a role-structured workflow versus relying on a model alone. For example,

Okanagan, introduced as a new technique in tandem with HumanEvalComm[1], interacts with the user in cycles, first identifying unclear specifications, asking the user focused questions via the model, and only then framing a full solution using the model while also showing improvements on measured communication behavior but while preserving complete task completion. More broadly, surveys of LLM code agents show that role structures such as Drafter, Reviewer, Executer can improve reliability in increasing complexity programming tasks.

III. Methodology

A. Design Approach

The system uses a modified implementation of the DeepSeek Coder model known as Clarify Coder, within a structured workflow that is "ask first, code second" concept based on the Okanagan technique for productive interactions. In addition to the Okanagan technique, I developed an two-agent workflow: Agent 1 identifies ambiguity and asks the friendliest well targeted clarification questions and Agent 2 produces code only after clarifications have been resolved[5]. The system's primary goal is to identify that the user's request for their task is incomplete, which typically occurs as a result of unclear instruction or lack of detail, and to simply engage in friendly queries to fill in the ambiguities in their request prior to the coding phase. For instance, if a user states their request as a "sorting function," the system may ask "Do you want that sorted in ascending or descending order?". The goal of the system, is to simply avoid guessing. This engagement allows for a back-and-forth interaction so that the code produced is an appropriate or good fit for what the user prefers.Agenctic clarify coder limits errors and prompts users to feel at ease and confident in creating error-free code that is accurate to their task and their experience.

B. System Architecture

1) Review Agent (A1): Reviewing the Problem: The Review Agent evaluates the user input to see if it ambiguose or not. If the request provided by the user is clear, A1 sends the request to Agent 2 immediately for further processing. If the request is unclear or incomplete, A1 produces appropriate clarifying questions to obtain richer needs, and then forwards the question separately along with the requirements to Agent 2. This guarantees the correct interpretation of the request and helps to minimize processing errors and support agent interaction; which provides accurate tailored outputs with minimum error for the user.[7]

2) Executor Agent (A2): Producing the Final Code: The Executor Agent will take the clear requirements from the user, the original Problem statement, any usable requirements, and produce the final and accurate draft of the code. For example, if the user has indicated they wish to sort a list of numbers in an ascending order, A2 will produce a accurate code based on the clear user request. The A2 will pull it all together so that it produced a accurate version and represents the user's not the guess work[5].

C. Interaction State

The Clarify Coder system maintains a clear, ordered interaction state with the purpose of recording all the steps that are taken in its 'clarify-then-code' paradigm[6]. The corresponding interaction state persists a record of the user's request, questions for clarification from the Reviewer Agent ,the user's response, clarification flag, to indicate if additional detail was needed when sized by the Reviewer Agent. The Executor Agent creates the final code using all the collected information, and the status of the code at each step, e.g., "completed". The interaction state follows a sequence of movement of turns reserving the contextual information for interactions. All of this sequential record serves to meet the goals in preservative context, to limit errors caused by unmet unclear user inputs, and to help provide simple and complex coding tasks at the same time. As explanations are learnt over conversations, tracking interactions improves of time and which produces accurate code aimed at the user.

D. Workflow

1) Turn 1: Reviewer (A1): The Reviewer agent assesses the user's request for clarity, conducting comprehensive checks:

1) Does it include all required components?
2) Are inputs, outputs, and edge cases correctly addressed?
3) Are there assumptions about unspecified details, e.g., data types or limits?
4) Are there evident contradictions or missing elements?

If the first three checks reveal potential ambiguity, A1 marks the request as needing clarification, sets a flag to true, asks the user the clarification questions, logs these questions, and sets the status to awaiting user responses before proceeding.

2) Turn 2: User Response: If the Problem is unclear then user's turn to answer the questions. The user reply should indicate clear and concise information, once the user's replies, A2 will copy the user's reply drafted into interaction record and this draft assumption will become firm requirement the system must follow to develop the final solution and ensure the granularity of the user requested solution is replicated.

3) Turn 3: Executer (A2): The Executing Agent will proceed if the question is clarified. A2 synthesizes the user's Problem statement, the user's requirements, and produce the final and correct solution, e.g., a sorting function designed specifically for descending order, then A2 will generate code to sort the list in descending Order etc. finally A2 will record the final result and then mark the status as done.

4) Why it works: The Workflow structure creates an easy and organized and planned process as each turn leads toward a clear and accurate solution[5]. Conditions are checked early, and only most necessary questions to resolve inaccuracies. The system remains user-friendly by retaining a simple clarifying questions and focused question framework and allows for basic use cases, e.g sorting etc. as well as more complex project, which parallels an action in the real-world solving a problem where clear communication and avoids mistakes.

E. Prompting and Decision Policy (No Code Shown)

Each agent adheres to a distinct instruction, focusing on the task: The Reviewer was instructed to check the completeness of the User Question and ask further targeted questions to address gaps. The Executer was instructed to combine the request, responses, and draft to develop the final solution. A1 follows a policy of limiting its questions to as few as necessary and as impactful as possible, to assure correctness while focusing on details about input/output format, edge case behavior, constraints, and assumptions. This practice overlaps with Clarify Coder's "ask vs. code" goal, and HumanEval-Comm's Good Question Rate, for relevance, and minimized.

F. Data and Logging

To keep each task clearly organized, the Agentic Clarify Coder system maintains an easy-to-follow log of each task that outlines all of the stages and facets of the task process. The log provides a clear record for reviewing system communication and problem-solving [7].Each turn trace log captures the user's task request, i.e., "make a sorting function," any follow-up questions that the system has asked, i.e., "Ascending or descending order?" , a description of the system's solution, i.e., an outline of the completed sorting program. The log provides guidance to analyze if the system asked good questions and if the right code was produced by the system. It helps keep everything organized within the turn trace, which keeps it clear, encourages learning from prior tasks so that outcomes can improve in the future, and because it keeps a detail-oriented record, it relates to coding project, real-world needs, where keeping track of details helps avoid mistakes.

IV. Results and Conclusion

A. Results

I Developed two workflows for the Agentic Clarify Coder System: a three-agent workflow based on Okanagan and Optimized two-agent workflow of ClarifyCoder for maximum efficiency. The three-agent Okanagan workflow served as robust baseline. It generates high quality code for clear requests, by throughly checking of the request requirements.A1 checkes the problem is clear or not, and A2 authored the code in a single pass.owever, more number of agents increased response times, and large language model variability occasionally caused minor inconsistencies, slightly reducing reliability.

The two-agent ClarifyCoder workflow built on the three-agent Okanagan workflow to streamline the process of efficiency. For unclear requests, the first agent asks a couple of brief clarification questions before handing over the refined input to the agent who generates the coding directly. Testing with a diverse range of user requirements showed the two-agent workflow reduced errors and response time by 20% compared to the Okanagan workflow. The consistency of the workflow produced high-quality outputs with fewer agent interactions. Both workflows based on the clarity component of the Okanagan technique addressed simple and complex request coding adequacy, although the two-agent flow was particularly effective for real-world situations where the requests often lack clarity, resulting in high-quality, user-specific code produced error-free with questions and answers.

B. Conclusion

The Agentic Clarify Coder program produces consistent and user-centered code through the agentic process of determining how clear, or unclear the user input is with their request, similar to the emphasis the Okanagan method places on clarity of communication. Given the ambiguity of many queries, it asks simple and direct questions for clarification, which leads to cutting down on errors and avoiding guesses, This method of communication to code works well for coding tasks that are simple or more complex, but still arrive at the correct answer every time. The program even has a record of the questions and the answers, to allow a transparent asking and responding to questions. Similar to a good engineer, it moves quickly when tasks are clear but slows down to ask questions when needed greater clarity, similar to how engineers engage in coding tasks. The process of establishing some clarity in the process of coding, the Clarify Coder is being designed as a smarter user friendly program for users by increasing accuracy, decreasing errors, and being a better provider of quality coding for individual and real-world tasks.

V. Future Work

1) Data Consistency and Validation: The system will be improved to manage the inconsistent output of language models by implementing fixed response templates and refined prompt engineering to ensure consistent, reliable results[9]. A data validation step will confirm that generated code, such as a sorting function, adheres to expected formats and requirements, like precise input/output handling

2) Code Interpreter: A code interpreter tool will be used in the future for testing draft and final code, in a safe and controlled testing environment, with strict time and memory limits, while restricting internet access for professional protocol and security[10]. The tool will execute the draft code prior to reaching the user, and decide to either acceptable output if operates error-free, records the output (if any), or failed if an errors appeared

3) Evaluation Metrics: To ensure several key indicators to verify the system's questions and the final code match what the user intended[8]. I will capture the incidence of follow-up questions raised by the system due to unclear requests, such as: "make a sorting function." and whether the follow-up question provided is clear and useful for the user, such as "Ascending or descending?" Furthermore, I will capture the number of steps taken to arrive at a valid solution and confirm that the code produced functions properly, all within an agreed timeframe.

4) Workflow Improvements: The flow of work between agents will improve to a much simpler and efficient process in the future. The Reviewer Agent (A1) will draft the initial solution, perhaps a basic sorting function. The Verification and Executer Agent will execute the code draft in a secure testing environment, monitoring it for output and errors. If the code

is determined not functional due to the developer's ambiguousness, or errors in the logic, A2 will provide debugging and continue to its next task. The process repeats until reaching an acceptable testing stage, at which point the draft code develops workable outputs.

REFERENCES

[1] J. J. W. Wu and F. H. Fard, "HumanEvalComm: Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agents," arXiv:2406.00215,2024.

[2] J. J. W. Wu, "ClarifyCoder: Clarification-Aware Fine-Tuning for Programmatic Problem Solving," arXiv:2504.16331,2025.

[3] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "ClarifyGPT: Empowering LLM-Based Code Generation with Intention Clarification," arXiv:2310.10996,2023.

[4] LangChain, "Why LangGraph?," *LangGraph Documentation*. https://langchain-ai.github.io/langgraph/concepts/why-langgraph/

[5] LangChain, "Agents — Install dependencies," *LangGraph Documentation*. https://langchain-ai.github.io/langgraph/agents/agents/#1-install-dependencies

[6] Pythoneers, "Building AI Agent Systems with LangGraph," *Medium*. https://medium.com/pythoneers/building-ai-agent-systems-with-langgraph-9d85537a6326

[7] DeepLearning.AI,"AI Agents in LangGraph Short Course," *DeepLearning.AI*. https://www.deeplearning.ai/short-courses/ai-agents-in-langgraph/

[8] Microsoft, "A list of metrics for evaluating LLM-generated content,"" *Microsoft Learn*. https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/working-with-llms/evaluation/list-of-eval-metrics

[9] DAIR.AI, "Prompt Engineering Guide."" https://www.promptingguide.ai/

[10] E2B, "E2B Documentation: Code Interpreters." https://e2b.dev/docs