CodeCraft: Multi-Agent Code Development Team

Mukesh Javvaji
Khoury College of Computer Sciences
Northeastern University
Boston, USA
javvaji.m@northeastern.edu

Satvika Eda
Khoury College of Computer Sciences
Northeastern University
Boston, USA
eda.s@northeastern.edu

Scott Pozder
Khoury College of Computer Sciences
Northeastern University
Boston, USA
pozder.sc@northeastern.edu

***Abstract*—This project presents a multi-agent code development system, where individual language agents specialize in chain-of-thought reasoning, code generation, debugging, and explanation. Modular design aims to improve code reliability, clarity, and debuggability by allowing each agent to focus on a specific cognitive task. To further refine these agents, the system integrates Reinforcement Learning with Human Feedback (RLHF) and Reinforcement Learning with AI Feedback (RLAIF). A reward model specifically trained to evaluate generated code is used alongside a PPO trainer to optimize the generator agent based on feedback. Preliminary results suggest that this architecture enables more accurate and interpretable outputs compared to single-agent code generation, highlighting the potential of specialization and feedback-driven fine-tuning in AI-assisted programming.**

***Index Terms*—Multi-agent systems, LLM, RLHF, RLAIF, Fine-tuning, LangGraph, StreamLit, Code Generation**

I. Introduction

Large language models have made impressive strides in generating code, making them valuable tools for developers and researchers. But most of these systems rely on a single all-in-one model to handle everything, from understanding the problem to writing and explaining the code. This one-size-fits-all approach can present challenges, especially when it comes to making the code reliable, easy to understand, and free of errors. These issues become even more noticeable when using smaller, more efficient models, which often trade off performance for lower computational costs.

A. Problem Statement

Although large language models have demonstrated strong capabilities in code generation, relying on a single model to perform multiple complex tasks such as reasoning, coding, debugging, and explaining can reduce both accuracy and clarity. This challenge is even greater when using smaller, lightweight models that lack the capacity to handle these tasks effectively on their own.

The central problem this project aims to address is to check if a collaborative system of smaller, specialized agents work together to match or exceed the performance of a larger, general-purpose language model in coding tasks.

This question is motivated by the growing need for modular, compute-efficient AI systems that can produce reliable and interpretable code, especially in resource-constrained environments.

Key challenges included:

- Designing coordination mechanisms between agents (planner, chain of thought, developer, debugger, explainer)
- Integrating UI with the agentic workflow because of compute requirements.
- Evaluating performance gains in terms of error detection, code correctness, and explanation clarity

B. Summary of Approach

To address the limitations of monolithic code generation systems, we developed a multi-agent framework where individual agents are assigned specialized roles: Planner, Chain-of-Thought (CoT), Developer, Debugger, and Explainer. Each agent is implemented using a lightweight language model optimized for its specific function. The agents communicate through a controlled message-passing system implemented using LangGraph, which supports modular task flows and persistent memory states.

The development process involves the following.

- Using a Planner Agent to decompose high-level user goals into actionable coding steps.
- Using a CoT agent to generate algorithmic steps to address coding problems.
- The Developer Agent generates code snippets based on each step.
- The Debugger Agent tests the code and flags logical or syntactical issues.
- The Explainer Agent provides human-readable justifications for generated code or error fixes.

We used HuggingFace Qwen2.5 coder models for agent tasks, choosing them for their balance between speed and task-specific accuracy.

To address challenges:

- A LangGraph state machine system to create a workflow for better coordination.
- Reward models for both RLHF and RLAIF are trained to reflect preferences for correct, readable, and logically structured code.
- Feedback loops improve performance over time by reinforcing desirable behavior and penalizing low-quality output.

We hypothesize that combining modular agent design with RLHF/RLAIF optimization will:

- Increase code accuracy and robustness
- Yield more interpretable outputs through agents fine-tuned with human-like reward signals

C. Summary of Evaluation

Our multi-agent system worked noticeably better than a typical single-model setup of a similar size. By giving each agent a clear role—like planning, chain-of-thought, coding, debugging, or explaining: we saw more organized workflows and cleaner outputs.

The Debugger Agent helped catch and fix issues that were harder to spot in a one-shot generation approach. The Explainer Agent made the outputs easier to understand, especially when things went wrong. Training these agents with human and AI feedback made a big difference. Responses felt more thoughtful and aligned with what a developer might expect.

The system consistently produced more reliable code, found bugs more effectively, and gave clearer explanations in practice.

II. Background Work

This project is inspired by and builds upon three significant advancements in AI-driven code development: AMD's Agent Laboratory, AgentCoder, and the application of Reinforcement Learning from AI Feedback (RLAIF) for code generation.

A. Agent Laboratory by AMD and Johns Hopkins University

AMD's Agent Laboratory [2] is a pioneering framework that emulates a virtual research team composed of specialized AI agents. Each agent is assigned specific roles, such as literature review, experimentation, and report compilation, mirroring the functions of human researchers. The system's core component, MLE-Solver, autonomously generates machine learning code, executes experiments, and iteratively refines outputs based on task instructions and accumulated knowledge. This modular approach has demonstrated efficiency and adaptability across various computational environments, from personal laptops to GPU clusters.

Our project adopts a similar multi-agent architecture tailored for code development tasks. By assigning distinct responsibilities like chain-of-thought reasoning, code generation, debugging, and explanation to specialized agents, we aim to enhance the reliability, interpretability, and efficiency of code generation processes. This modular design facilitates targeted optimization and scalability, aligning with the principles demonstrated in the Agent Laboratory.

B. AgentCoder: Multi-Agent Code Generation

Another relevant work is AgentCoder [1], which proposes a three-agent framework for collaborative code generation involving a programmer, test designer, and test executor. Unlike previous multi-agent systems such as MetaGPT or ChatDev, AgentCoder optimizes agent interactions by reducing token overhead and improving feedback quality. It introduces a dedicated test designer agent that independently creates diverse and objective test cases, avoiding bias from seeing the code, and a test executor agent that runs tests in a local environment to provide actionable feedback. The programmer agent then iteratively refines the code based on this feedback.

Our system adopts and extends this idea by assigning additional roles—such as a debugger and explainer agent, along with incorporating RLHF and RLAIF to continually improve model performance. This fusion allows for more granular task specialization and optimizable agent behavior, offering a novel direction beyond AgentCoder's fixed pipeline.

C. Reinforcement Learning from AI Feedback (RLAIF) in Code Generation

In the study "Applying RLAIF for Code Generation with API-usage in Lightweight LLMs" [3], the authors developed a method to generate human-like evaluation scores for model-generated code responses. This approach involved using a larger language model (e.g., GPT-3.5) to assess the quality of code outputs produced by smaller models. The evaluation focused on several key criteria:

1) Syntax Correctness: Ensuring that the code adheres to the correct syntax of the programming language.
2) Functional Correctness: Assessing whether the code performs the intended task as described in the prompt.
3) Readability and Style: Evaluating the clarity, structure, and adherence to coding standards.

The larger language model was prompted to provide a numerical score reflecting the overall quality of each code snippet, simulating a human evaluator's judgment. These scores were then used to train a reward model that guided the fine-tuning of the smaller code generation models through reinforcement learning. This methodology aimed to improve the performance of lightweight models in generating accurate and functional code.

In our system, we integrate RLAIF to train a specialized reward model for the code generation agent. This model evaluates generated code based on predefined criteria, providing feedback that informs the Proximal Policy Optimization (PPO) training process. By leveraging AI-generated feedback, we aim to refine the code generation capabilities of our agent, ensuring higher accuracy and alignment with intended functionalities.

III. Methods

Our system includes a Streamlit-based UI that allows users to interact with the backend multi-agent system. The user submits a natural language query through the UI, which is then forwarded as a structured request to the agent framework. We use Streamlit UI to capture inputs, display agent responses and collect user feedback for RLHF.

The Multi-Agent System processes the request by passing it through the Planner and other specialized agents. The final response is sent back to the Streamlit UI and displayed to the user as illustrated on Figure 1.

To improve the quality of responses from the multi-agent system, we implemented fine-tuning of the language models, structured the agent interactions using a LangGraph-based workflow, and applied reinforcement learning from human

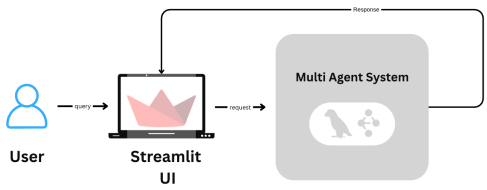

Fig. 1. Application Flow

feedback (RLHF) and reinforcement learning from AI feedback (RLAIF).

A. Fine-Tuning Agent Models

1) Fine-tuning Approach for Student Models: This research employs a teacher-student knowledge transfer paradigm to create specialized language models for a multi-agent programming environment. The approach involves fine-tuning smaller, more efficient Qwen2.5-0.5B models (students) to learn from larger, more capable Qwen2.5-7B models (teachers). This methodology enables the creation of specialized agents that can collaborate effectively within a programming workflow while maintaining computational efficiency.

Rather than training student models from scratch on raw data, we leverage the capabilities of teacher models to generate high-quality examples. This approach allows student models to benefit from the reasoning and generation capabilities of larger models while maintaining a significantly smaller parameter count for deployment efficiency.

2) Model Architecture and Training Environment: We used two separate Qwen base models for each of our teacher and student model tasks.

Teacher Models:

- Qwen2.5-7b-Instruct (for chain of thought reasoning and code explanations)
- Qwen2.5-Coder-7B-Instruct (for code generation and debugging)

Student Models:

- Qwen2.5-0.5B-Instruct (for reasoning and explanations)
- Qwen2.5-Coder-0.5B-Instruct (for code generation and debugging)

Training was conducted on NVIDIA A100-SXM4-80GB GPUs using PyTorch 1.13.1 with CUDA 11.4. The implementation utilized the Transformers library (version 4.37.0) and Accelerate (version 0.25.0) for efficient training.

3) Agent Specialization: Our system architecture consists of four specialized student models, each trained for a distinct phase of the programming workflow:

1) The **Chain-of-Thought (CoT) Agent** forms the entry point to our system, generating algorithmic strategies and problem-solving approaches. This agent translates basic python coding problems into algorithmic strategies for writing the code.
2) Following the reasoning phase, the **Developer Agent** translates algorithmic strategies into executable Python code. This agent consumes the output from the CoT agent along with the original problem statement to produce working implementations that follow the specified approach.
3) The **Debugger Agent** then analyzes and refines the code produced by the Developer Agent, identifying and fixing bugs, optimizing inefficient patterns, and ensuring adherence to best practices. This agent contributes to code quality and correctness without requiring human intervention.
4) Finally, the **Explainer Agent** creates clear, concise explanations of the code functionality. This agent translates technical implementations into accessible language, making the code more understandable to users with varying levels of programming expertise.

This pipeline architecture enables a complete programming workflow from problem formulation to understandable solution, with each agent building upon the outputs of previous stages.

4) Dataset Creation and Fine-Tuning Pipeline: The Mostly Basic Python Problems (MBPP) dataset served as the foundation for our training data. This dataset consists of 974 Python programming problems with associated test cases and solutions.

Our dataset generation process followed an iterative approach. First, teacher models generated high-quality examples for each problem in the training dataset. We utilized all 374 examples from the MBPP training set for the teacher models to generate solutions, and reserved 50 test examples for evaluation. This provided sufficient data for fine-tuning and allowed testing generalization to new coding problems in the test examples.

The data flow through our pipeline followed the logical progression of the programming workflow:

1) The CoT teacher model generated algorithmic reasoning for each problem
2) The Developer teacher model generated code implementations based on the problem and CoT output
3) The Debugger teacher model refined the code implementations
4) The Explainer teacher model created explanations of the debugged code

This sequential generation created aligned datasets where outputs from each stage became inputs for subsequent stages, ensuring coherence throughout the pipeline.

To facilitate efficient model training, we implemented a custom `CodeCraftDataset` class that handles prompt template application, tokenization, and the creation of attention masks and label masks. This scalable dataset class ensures that models focus their learning on generating appropriate outputs as training follows the fine-tuning pipeline, rather than reproducing only based on MBPP input prompts.

5) Prompt Engineering: The success of our approach relied heavily on carefully designed prompt templates tailored to each agent's specific role. These templates provided structured

guidance that created appropriate outputs from both teacher and student models.

- For the CoT Agent, we developed a template that enforces a systematic reasoning approach (Appendix A)
- The Developer Agent template leverages the CoT output to guide implementation (Appendix B)
- The Debugger Agent template focuses on code refinement without verbose explanations (Appendix C)
- The Explainer Agent template solicits clear, concise explanations (Appendix D)

These templates establish clear expectations for model outputs, ensuring consistency and quality throughout the pipeline. The precise formatting requirements and constraints help produce outputs that seamlessly integrate across agents.

6) Fine-tuning Configuration: The fine-tuning process utilized the following hyperparameters:

TABLE I
FINE-TUNING CONFIGURATION PARAMETERS

Parameter	**Value**
Learning Rate	2e-5
Batch Size	10
Number of Epochs	6
Warmup Steps	$\approx 5\%$ of training steps
Maximum Token Length	512 (150 for CoT Agent)
Generation Temperature	0.6-0.7
Optimizer	AdamW
Scheduler	OneCycleLR
Gradient clipping norm	1.0

7) Alternative Approaches Considered: During our research, we also explored knowledge distillation via logits matching as an alternative to direct fine-tuning. This approach incolved training student models to match the output probability distributions of teacher models rather than just their final text outputs.

In theory, logits matching offers advantages for transferring nuanced knowledge, as it allows the student to learn the full uncertainty profile of the teacher model rather than just the most likely tokens. However, our experiments with this approach revealed several limitations in the context of our multi-agent system.

First, logits matching required significantly more computational resources, as it necessitates storing and processing the full logit distributions across the vocabulary for each token. With vocabulary sizes of tens of thousands of tokens, this becomes memory-intensive, particularly for longer sequences.

Second, we observed less consistent results across different agent types. While logits matching showed promise for the Developer agent, it performed less reliably for the CoT and Explainer agents, which required english-only generation from its multi-lingual vocabulary.

Third, the iterative pipeline architecture of our system requires well-formed, discrete outputs that can be reliably passed between agents. The probabilistic nature of logits-based distillation sometimes produced outputs with structural inconsistencies that impaired the overall pipeline functionality. Given these considerations, we determined that direct fine-tuning on teacher-generated examples provided the most effective balance between performance, resource efficiency, and pipeline compatibility for our multi-agent programming system.

B. Multi-Agent Workflow

Our system is structured as a modular multi-agent environment, with agents communicating and collaborating through a well-defined flow controlled by a central Planner.

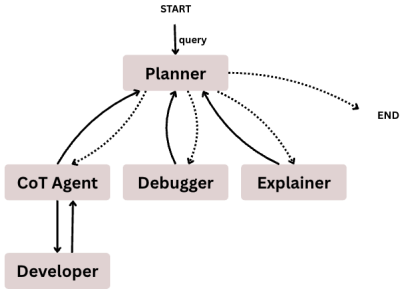

Fig. 2. LangGraph Workflow for the Multi Agent system

The core of our system is built as a modular multi-agent workflow using LangGraph, which allows the flexible composition of agents with clear state transitions and feedback loops. Each agent in the system serves a distinct reasoning, development, debugging, and explanation which is powered by fine-tuned versions of the Qwen2.5 models as mentioned in the above section.

- **Planner**: Routes the user query by classifying its intent (developer/debugger/explainer/planner). If the input falls outside code-related tasks, it finalizes the conversation with an informative response.
- **CoT Agent**: When triggered by the Planner, it breaks down the problem into a step-by-step reasoning chain before passing it to the Developer.
- **Developer**: Generates Python code based on either raw user queries or structured reasoning output from the CoT agent.
- **Debugger**: Analyzes the generated code for errors and loops back fixes through the Planner.
- **Explainer**: Produces human-readable explanations to improve interpretability and transparency.

Each agent runs independently but is linked via LangGraph as mentioned in Figure 2, which manages the flow of information and agent transitions. The dotted arrows represent optional or conditional paths, such as when the Planner routes to Debugger only if a problem is detected.

C. Reinforcement Learning with Human and AI Feedback (RLHF & RLAIF)

This project integrates both Reinforcement Learning with AI Feedback (RLAIF) and Reinforcement Learning with Human Feedback (RLHF) to improve the behavior of multiple language agents in the code development system. The following subsections outline the construction of the reward model,

the feedback collection strategy, and the application of PPO for fine-tuning the agents.

1) Reward Model Pretraining with RLAIF: To initiate reward-based optimization, we applied RLAIF to train a reward model for the code generator agent. Using prompts from the MBPP dataset, responses were generated using the Qwen/Qwen2.5-Coder-0.5B model. Each prompt-response pair was evaluated using GPT-4o, which produced a score out of 8 based on correctness, structure, and overall quality. These scores were normalized to the range [0,1] and used to train a regression-based reward model specific to code generation. This RLAIF-based reward model serves as the initial performance estimator for PPO-based training.

2) Feedback Collection (RLHF): Human feedback was collected through a custom user interface, visualized in the figures below, using two complementary mechanisms:

1) Scalar Feedback: Users were shown a single model-generated response and asked to provide a thumbs up or thumbs down, representing a binary evaluation of the response quality (Figure 3).

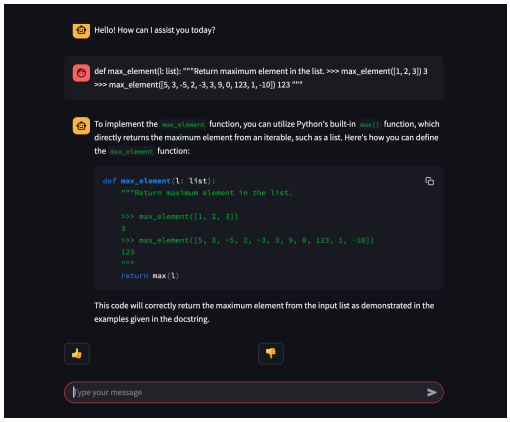

Fig. 3. Streamlit UI for scalar feedback collection

2) Preference-Based Feedback: Users were presented with two responses to the same prompt—one generated by the Qwen2.5-Coder-7B-Instruct teacher model and the other by our multi-agent system. They were asked to select the preferred response. This comparative feedback provides a strong signal for preference modeling (Figure 4).

All feedback data was stored in structured JSON format, along with metadata identifying which agent produced each response. This metadata is used to isolate agent-specific feedback during training.

3) Agent-Specific PPO Fine-Tuning: Once a sufficient amount of feedback was collected, the PPO (Proximal Policy Optimization) algorithm was used to fine-tune the agents. The planner agent plays a critical role in tracking which response came from which agent. During training, this traceability ensures that only the agent associated with the response is updated using PPO, preserving the integrity of the other agents. Figure 5 illustrates the RLHF loop.

Fig. 4. Streamlit UI for preference based feedback collection

While the PPO fine-tuning framework is generalized across all agents, only the code generation agent currently benefits from a pretrained reward model derived from RLAIF. The reward models for other agents (developer, debugger, explainer, and chain-of-thought) can be trained in a similar fashion using the same pipeline. This modular and selective training approach ensures that feedback leads to targeted improvement without introducing unnecessary parameter updates in unrelated components.

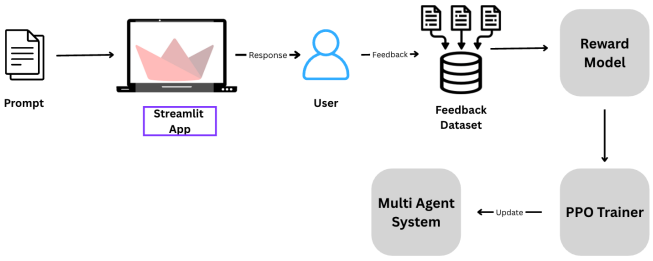

Fig. 5. RLHF Loop

IV. Experiments and Evaluation

To evaluate the performance and usability of our multi-agent code development system and RLHF integration, we conducted a small-scale user study. The aim was to collect human feedback on model-generated code responses for training reward models and assessing model preference. While the current study was conducted with a limited number of users, it provides a proof-of-concept for the feedback loop and offers insight into future experimental design.

A. Experiment Design and Task Description

Participants were asked to evaluate code responses generated by the system through two types of tasks:

1) Scalar Feedback Task: Subjects were shown a single prompt and its corresponding code output and asked to provide a thumbs up or down based on their judgment of its correctness and quality.
2) Preference-Based Task: Subjects were presented with two responses to the same prompt—one generated by

the Qwen2.5-Coder-7B-Instruct teacher model and the other by our multi-agent system—and asked to choose which response they preferred.

Both interfaces were deployed using Streamlit, a Python-based web application framework, enabling accessible and interactive feedback collection in a browser-based environment.

B. Participant Interaction and Data Recording

Initial feedback was collected from two graduate students in the Artificial Intelligence program at Northeastern University. These participants were familiar with code evaluation tasks and thus capable of providing meaningful feedback. In an ideal scenario, this experiment would be scaled to include a broader sample of 20–30 users from varied technical backgrounds, such as CS majors and practicing developers, to enhance feedback diversity and generalizability.

All interactions were logged in structured JSON format, including the prompt, response(s), user choice, and metadata about the generating agent. This data was later used to train the reward models and fine-tune the respective agents.

C. Instruments and Evaluation Metrics

The evaluation used two custom-built feedback tools implemented in Streamlit: one for scalar ratings and one for pairwise preference selection. These interfaces were designed for low cognitive overhead and rapid feedback collection.

In addition to qualitative feedback, future iterations of the experiment may incorporate quantitative usability measures such as the System Usability Scale (SUS) or NASA-TLX, as well as objective metrics like task success rate, pass@1, and execution correctness to better quantify improvements in model performance.

D. Preliminary Results and Observations

Although the evaluation was conducted with only a small number of participants, early observations indicate that the responses generated by the multi-agent system are on par with those from the much larger Qwen2.5-Coder-7B-Instruct model. Users expressed satisfaction with the overall quality and clarity of the responses, particularly in tasks involving code generation.

A notable finding is the impact of the chain-of-thought agent, which significantly enhanced the output quality of the code generator, despite having far fewer parameters. By explicitly structuring the thought process before generation, this agent helped the generator model produce more coherent and task-aligned code. Figures 6 and 7 illustrates the responses generated by the Qwen 0.5B model and our Multi Agent System.

These results suggest that modular specialization, when combined with targeted reward optimization, can empower smaller agents to deliver high-quality responses comparable to much larger models. Further studies with a broader participant base will be conducted to validate these findings and support more extensive analysis.

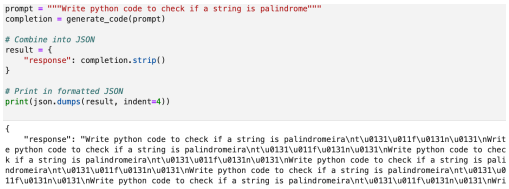

Fig. 6. Response generated by the Qwen 0.5B model

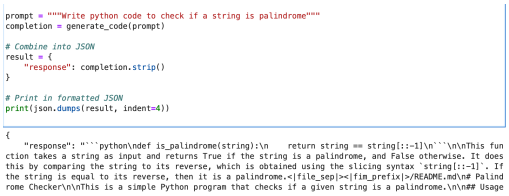

Fig. 7. Response generated by the Multi Agent System

V. Analysis and Discussion

A. Performance Analysis of Fine-tuned Agents

This section examines the performance improvements observed in our fine-tuned agents compared to their pretrained counterparts. We analyze outputs from each specialized agent in the pipeline to assess the effectiveness of our fine-tuning approach.

1) Chain-of-Thought (CoT) Agent: The CoT agent translates natural language problem statements into structured algorithmic strategies. Comparing outputs from pretrained and fine-tuned models reveals significant improvements in adherence to the desired format and reduction of redundant information.

For the problem "Write a function to find whether a given array of integers contains any duplicate element," both pretrained and fine-tuned models correctly identified a hash table-based approach. However, the pretrained model exhibited a tendency toward repetition, particularly in points 4-7 of its output:

```
4. Edge cases: An array of integers; return
   True if duplicates are found, False
   otherwise
5. Complexity: O(n) time, where n is the
   length of the array
6. Conclusion: A list of integers; return
   True if any element appears more than
   once, False otherwise
7. Note: Use a hash
```

Points 4 and 6 essentially restate the same information with slight rephrasing, while point 7 appears incomplete and repeats the desired data structure. This redundancy and formatting inconsistency were common patterns in the pretrained model's outputs.

The fine-tuned model, while maintaining the core approach, demonstrated more concise and structured reasoning. Through exposure to teacher-generated examples, it learned to eliminate

redundant information and maintain consistent formatting. This improvement is particularly valuable in the CoT agent, as cleaner reasoning structures provide better guidance for downstream agents in the pipeline.

2) Coder Agent: Our analysis of the Coder agent revealed two significant improvements through fine-tuning. First, the pretrained student model frequently failed to generate any code for programming problems, instead producing explanations or pseudo-code that lacked implementation details. After fine-tuning on teacher-generated examples that consistently included complete code implementations, the student model more reliably produced executable Python code for the given problems.

For the duplicate detection problem, both pretrained and fine-tuned models generated similar, correct implementations:

```
def has_duplicate(arr):
    seen = {}
    for num in arr:
        if num in seen:
            return True
        seen[num] = True
    return False
```

However, the consistency of code generation improved dramatically across the test suite. The pretrained model would occasionally omit critical implementation details or produce non-executable pseudo-code, particularly for more complex problems. The fine-tuned model, by contrast, consistently generated complete, executable implementations. However, both failed to produce solution code if the coding problem had an element that the models did not understand (i.e. "Write a function to get a lucid number smaller than or equal to n." where the teacher model does not know what a lucid number is).

Additionally, by prompting the Coder agent with the structured reasoning from the CoT agent, we observed improved alignment between algorithmic strategy and implementation. This integration demonstrably helped the Coder agent follow the (an often optimized) specified approach rather than defaulting to alternative implementations, particularly for problems with multiple possible solutions.

3) Debugger Agent: The Debugger agent showed perhaps the most dramatic improvement through fine-tuning. The pretrained model exhibited a strong tendency to leave code unchanged, even when presented with implementations containing obvious bugs or inefficiencies. This conservative behavior limited its utility in a practical programming pipeline.

After fine-tuning on teacher-generated examples that demonstrated appropriate code refinements, the student model became significantly more effective at identifying and fixing issues. For the duplicate detection problem, both pretrained and fine-tuned models recognized that the implementation was already correct and efficient, but for problems with actual bugs, the difference was substantial.

4) Explainer Agent: The Explainer agent's performance showed notable refinement through fine-tuning. For the duplicate detection problem, both pretrained and fine-tuned models provided explanations that covered the core functionality. However, the fine-tuned model demonstrated superior organization, clarity, and technical insight.

The pretrained model's explanation was serviceable but somewhat generic:

> The provided Python function `has_duplicate` is designed to determine whether an input array contains at least one duplicate number. It does so by utilizing a dictionary called `seen`. Each time a new number is encountered, it checks whether that number already exists within the dictionary's keys. If such a key is found, the function returns `True`, indicating that duplicates are present. Otherwise, if no duplicates are found after processing all numbers, the function concludes that there are no duplicates and returns `False`.

While accurate, this explanation lacks technical depth regarding the algorithm's efficiency or insight into why this approach is effective.

The fine-tuned model's explanation, by contrast, offered greater technical insight:

> This Python function checks if there are any duplicate numbers in an input list. It uses a simple hash table (seen) to keep track of numbers we've seen before. As we go through the list, we check if each number is already in our seen dictionary. If so, then we know there's a duplicate and we return True immediately. If not, we add the number to the seen dictionary. If no duplicates were found after checking the entire list, we return False. The key insight is that dictionaries provide average O(1) time complexity for lookups and insertions, so this approach runs very efficiently even for large lists.

The fine-tuned explanation highlights the O(1) lookup complexity advantage of dictionaries and explains why this makes the approach efficient, providing valuable context beyond merely describing what the code does. This type of insight helps users understand not just the implementation but the algorithmic principles behind it.

B. Cross-Agent Integration Analysis

Beyond individual agent improvements, our analysis revealed significant benefits from the integrated pipeline approach. The sequential flow of information—from reasoning to code generation to debugging to explanation—created a coherent programming experience with each agent building upon the work of its predecessors.

We observed that the quality of upstream agents had direct effects on downstream performance. For example, when the CoT agent provided clear, structured reasoning, the Coder agent produced more accurate implementations. Similarly, well-structured code from the Coder agent enabled more focused debugging and clearer explanations from subsequent agents.

This interdependence highlights the importance of our pipeline-aware fine-tuning approach, where each agent was trained to work with the outputs of previous agents rather than in isolation. The resulting synergy produces a programming experience that exceeds the capabilities of any individual component.

VI. Conclusion

In this project, we developed and implemented a multi-agent code development system consisting of specialized language agents for chain-of-thought reasoning, code generation, debugging, and explanation. To enhance these agents, we integrated a reinforcement learning pipeline using both human and AI-generated feedback (RLHF and RLAIF). A reward model trained using scores from GPT-4o served as the basis for optimizing the code generation agent via PPO. Feedback collection interfaces were built using Streamlit, allowing users to provide both scalar and preference-based feedback in an accessible format.

Preliminary evaluation, conducted with graduate students in AI, revealed that the multi-agent system performs comparably to the much larger Qwen2.5-Coder-7B-Instruct model. Notably, the chain-of-thought agent substantially improved the generator's output quality, despite its significantly smaller parameter size. These results demonstrate the potential of modular, feedback-driven agent architectures to match or exceed the performance of large monolithic models while maintaining efficiency and interpretability.

A. Future Work

- Feature-based Distillation: Explore using internal representations instead of output text for model distillation to retain deeper task-specific knowledge.
- Multi-Agent Architectures: Experiment with more complex communication strategies between agents, including voting, memory sharing, or parallel reasoning.
- Noise in User Feedback: Investigate strategies to reduce bias introduced by noisy or inconsistent user feedback in the RLHF loop.

VII. Acknowledgements

A. Software Used

This project was made possible through the use of several key tools and technologies:

- Python for development
- Hugging Face Transformers for loading and fine-tuning language models
- LangGraph for orchestrating the multi-agent workflow
- Streamlit for the user-facing interface
- PyTorch for model operations and inference

B. Help from Human/AI

We incorporated both human and AI feedback for evaluation surveys, and leveraged tools like ChatGPT and Claude for research, ideation, and code debugging throughout the project.

C. Credit Assessment

This was a team effort, and each member contributed significantly to different parts of the project:

- Scott Pozder: Worked extensively on the fine-tuning process for all agents and contributed to model distillation, ensuring the models remained lightweight and effective.
- Mukesh Javvaji: Designed and implemented Reinforcement Learning from Human and AI Feedback (RLHF/RLAIF). Developed the reward models and integrated PPO (Proximal Policy Optimization) for agent training
- Satvika Eda: Designed and built the LangGraph workflow to manage agent transitions. Developed the Streamlit UI for user interaction. Implemented the RLAIF design and worked on developer model distillation.

Each team member participated equally in drafting, editing, and reviewing the report.

VIII. GitHub Link

Multi Agent Code Development System Code

References

[1] Huang, D., Zhang, J. M., Luck, M., Bu, Q., Qing, Y., & Cui, H. (2024). AgentCoder: Multi-Agent Code Generation with Effective Testing and Self-optimisation. University of Hong Kong, King's College London, University of Sussex, Shanghai Jiao Tong University.

[2] Schmidgall, S., Su, Y., Wang, Z., Sun, X., Wu, J., Yu, X., Liu, J., Liu, Z., & Barsoum, E. (2024). Agent Laboratory: Using LLM Agents as Research Assistants. AMD & Johns Hopkins University.

[3] Dutta, S., Mahinder, S., Anantha, R., & Bandyopadhyay, B. (2024). Applying RLAIF for Code Generation with API-usage in Lightweight LLMs. arXiv preprint arXiv:2406.20060.

[4] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., & Chintala, S. (n.d.). Knowledge Distillation Tutorial. PyTorch Tutorials.

[5] EvalPlus. (n.d.). EvalPlus Leaderboard from https://evalplus.github.io/leaderboard.html

[6] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. OpenAI, Anthropic AI, and Zipline.

[7] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program Synthesis with Large Language Models. arXiv preprint arXiv:2108.07732.

[8] Prompting Guide. (n.d.). Chain-of-Thought Prompting from https://www.promptingguide.ai/techniques/cot

[9] LangGraph. (n.d.). Introduction to LangGraph from https://langchain-ai.github.io/langgraph/tutorials/introduction/

[10] Qwen Team. (n.d.). Qwen2.5-Coder: A Family of Code Language Models from https://qwenlm.github.io/blog/qwen2.5-coder-family/

[11] Streamlit. (n.d.). Build Conversational Apps. Streamlit Docs from https://docs.streamlit.io/develop/tutorials/chat-and-llm-apps/build-conversational-apps

Appendix

A. Chain-of-Thought (CoT) Agent Prompt

```
Provide ONE concise algorithm strategy for
this coding problem in EXACTLY 4 numbered
points:

1. Input/output: Single sentence describing
   parameters and return value
2. Approach: Name the exact algorithm/data
```

```
   structure
3. Key steps: 3-4 bullet points with
   specific algorithmic operations
4. Edge cases: 2-3 specific edge conditions,
   no explanations needed

Keep total response short. Be direct and
technical. DO NOT include pseudocode,
explanations, test cases, or implementation
details.

Problem:
{problem}

Step-by-step solution:
```

B. Coder Agent Prompt

```
Generate only the Python code
implementation for this problem.
Problem:
{problem}

Using this algorithm strategy:
{solution_cot}

STRICT REQUIREMENTS:
- Your output must begin with ```python
- Your output must end with ```
- ONLY write clean, efficient Python code
- NO text before or after the code block
- NO descriptions of what the code does

Python code:
```

C. Debugger Agent Prompt

```
Fix all bugs and inefficiencies in
this Python code.

Problem:
{problem}

Original code:
{code}

CRITICAL REQUIREMENTS:
- Output MUST start with ```python
  and end with ``` ONLY
- NO explanations before or after
  the code
- NO test cases or example output
- NO justification of your changes
- MINIMAL code changes to fix
  bugs/inefficiencies

Debugged python code:
```

D. Explainer Agent Prompt

```
Create a short, beginner-friendly
explanation of this code.

Problem:
{problem}

Code to explain:
{code}

Keep your explanation to 1-2 paragraphs.
Focus on:
- What the code accomplishes
- The core algorithm approach used
- One insightful observation
  about why it works
- Any clever tricks worth noting

Use friendly language that makes the
solution approachable.

Python code explanation:
```