Title of the talk

Seminar on “Machine Learning in Software Eningeering”

A.U.Thor

Summer 2024

Zusammenfassung

Large Language Models (LLMs) have shown impressive capabilities in code generation, yet they often struggle with complex programming problems due to rigid, single-path reasoning. This paper explores PairCoder, a novel multi-agent framework inspired by human pair programming. PairCoder consists of two LLM agents—a Navigator and a Driver—that collaborate through multi-plan exploration and feedback-driven refinement. The system dynamically generates, evaluates, and repairs code using iterative reflection and execution feedback. Experimental results on benchmarks like HumanEval and CodeContest demonstrate superior accuracy over prompt-based methods. This paper also compares PairCoder to "Guided Code Generation with LLMs,"which employs hierarchical decomposition and bottom-up synthesis to enhance LLMs' compositional reasoning and long-context management. By contrasting these frameworks, we highlight strengths and limitations in adaptive versus structured agentic code generation strategies.

Inhaltsverzeichnis

1 Introduction

Recent advancements in Large Language Models (LLMs) have revolutionized software engineering tasks such as code completion, synthesis, and debugging [4]. Despite their capabilities, LLMs often struggle to reason compositionally or manage complex dependencies across longer prompts, especially when relying on single-pass generation strategies [2].

To address these limitations, two recent agent-based frameworks have emerged. The first, *PairCoder*, models human pair programming by pairing two LLM agents: a **Navigator** and a **Driver** [4]. Through iterative plan selection, code generation, and feedback-based refinement, PairCoder dynamically adapts its strategy based on test execution results. This allows it to overcome flaws in static prompting and recover from runtime failures.

In contrast, *Guided Code Generation with LLMs* introduces a structured decomposition-based architecture in which a **Generalist Agent** hierarchically splits a problem into subtasks, which are solved by **Code Agents** and validated by **Tester Agents** [1]. This hierarchical, bottom-up tree synthesis enables improved compositionality, particularly for long-form or nested logic problems.

This seminar paper investigates the architecture, methodology, and performance of PairCoder, and compares it with Guided Code Generation. By contrasting iterative feedback refinement with structured hierarchical planning, we analyze the trade-offs in generality, scalability, and agent collaboration models for LLM-driven software engineering.

2 Background

2.1 LLMs for Code Generation

Large Language Models have become central to modern software automation. Tools like OpenAI's Codex, Code Llama, and DeepSeek-Coder translate natural language into executable code [4]. Prompt-based enhancements such as Chain-of-Thought (CoT) and Self-Debugging improve reasoning depth [2], but they still face limitations in long-range dependency tracking and complex logic execution.

2.2 Limitations of Single-Agent Approaches

Most LLM-based frameworks rely on a single agent to perform all stages of reasoning, planning, and implementation. This makes them brittle in multi-step synthesis tasks. They also lack resilience when outputs fail tests or miss edge cases, as they cannot easily backtrack, revise strategies, or shift plans without external scaffolding [4].

2.3 Rise of Multi-Agent Frameworks

To address these limitations, multi-agent approaches have emerged. *Self-Debugging* enables autonomous correction using prompt-driven self-analysis [2], while *MetaGPT* simulates full software teams with agentic roles such as product manager, architect, and engineer [3]. However, these approaches often struggle to scale to deeper reasoning or fail to provide structured decomposition.

2.4 Introducing PairCoder and Guided Coder

PairCoder focuses on iterative refinement via two agents: a Navigator that proposes clustered solution plans, and a Driver that implements and tests code based on selected plans [4]. In contrast, **Guided Code Generation with LLMs** uses a hierarchical planning tree built by a Generalist Agent, solved bottom-up by Code Agents, and verified by Tester Agents [1]. This structured orchestration helps solve deeply nested tasks that require compositional reasoning and contextual memory beyond standard prompting techniques.

3 The PairCoder Framework

3.1 System Architecture

PairCoder is inspired by human pair programming, where a "Navigatorßets direction and a "Driverëxecutes tasks. In this framework, both roles are fulfilled by large language models (e.g., GPT-3.5 or DeepSeek-Coder). The two agents operate in an iterative loop where each round involves:

- Problem reflection and plan proposal by the Navigator
- Plan selection and code generation by the Driver
- Testing and feedback from execution
- Plan reassessment or code repair based on feedback

The workflow continues until the code passes all public test cases or the iteration limit is reached.

3.2 Navigator Agent

The Navigator is responsible for high-level reasoning and strategic control. Its tasks include:

- Reflecting on the problem (e.g., input/output analysis, edge cases)
- Generating multiple possible solution plans using diverse sampling

- Clustering and selecting representative plans using k-means++
- Evaluating execution feedback to determine whether to repair or switch plans

The Navigator utilizes prompt templates to generate insights, select plans, and analyze errors. It also maintains a historical memory of attempted code and failures to avoid repeating past mistakes.

3.3 Driver Agent

The Driver receives a selected plan and is responsible for concrete implementation. Its key tasks are:

- Generating initial code based on the plan
- Executing the code on provided public test cases
- Applying repair strategies as suggested by the Navigator

The Driver remains stateless and reactive, with each action depending solely on the instructions and feedback from the Navigator.

3.4 Algorithmic Loop

The collaboration is formalized in an iterative process (Algorithm 1 in the paper). With each iteration:

1. A new plan is selected (or retained if promising)
2. Code is generated and executed
3. Feedback is classified (e.g., Pass, Wrong Answer, Runtime Error)
4. The Navigator decides the next action: repair or switch

This loop continues until the code passes all public tests or the maximum iteration count is reached. In experiments, 10 iterations were sufficient for most tasks.

PairCoder's architecture mimics the agile, back-and-forth nature of human collaboration, allowing it to dynamically shift direction and avoid local minima in problem-solving.

4 Key Techniques

4.1 Multi-Plan Exploration

A critical innovation in PairCoder is its ability to explore multiple solution strategies simultaneously [4]. Instead of committing to a single path, the Navigator generates a batch of n solution plans using high-temperature nucleus sampling. These plans vary in algorithm type, such as brute force, greedy, or dynamic programming approaches.

To ensure diversity and eliminate redundancy, the plans are clustered into k groups using semantic embeddings and the k-means++ algorithm. One representative plan from each cluster is retained, forming the candidate pool. The Navigator then evaluates each plan against selection criteria—primarily functional correctness, followed by efficiency and robustness.

This strategy mimics how human developers might brainstorm several approaches, mentally simulate outcomes, and discard weak plans before writing actual code.

4.2 Feedback-Driven Refinement

PairCoder's second major technique is its ability to adjust course based on execution feedback [4]. After the Driver executes the code, the result is categorized as one of the following:

- **Pass**: All public tests are passed
- **Wrong Answer (WA)**: Output is incorrect
- **Runtime Error (RE)**: Exceptions or failures during execution
- **Time Limit Exceeded (TLE)**: Execution exceeds time constraints

Based on this feedback, the Navigator chooses between two paths:

- Apply a targeted repair strategy (e.g., fix logic or edge case handling)
- Abandon the current plan and switch to a different one from the pool

A historical memory tracks previous attempts and errors, preventing the system from repeating ineffective strategies. This enables intelligent backtracking and encourages exploration of new plans when refinement fails.

Together, these two techniques empower PairCoder to adaptively explore a broader solution space and incrementally zero in on correct implementations—without human intervention.

5 Evaluation and Results

PairCoder was evaluated against a wide range of code generation benchmarks using two different foundation models: GPT-3.5-Turbo and DeepSeek-Coder-Instruct 33B [4]. The benchmarks include HumanEval, MBPP (Mostly Basic Programming Problems), and CodeContest, covering both simple and competition-level programming tasks.

5.1 Accuracy Comparison

The core metric used in evaluation is *pass@1*, which measures the rate at which a generated program passes all private test cases on the first attempt [4]. PairCoder achieved:

- Up to **87.80%** pass@1 on HumanEval with GPT-3.5-Turbo
- Up to **15.15%** on the challenging CodeContest benchmark
- Relative gains of **16.97%–162.43%** over direct prompting baselines

Compared to prompting strategies (e.g., CoT, Self-planning) and refinement-based baselines (e.g., Self-debugging, INTERVENOR), PairCoder consistently delivered higher accuracy. The advantage was especially pronounced in difficult tasks with logical traps and limited public test coverage.

5.2 Ablation Studies

Ablation experiments were conducted to assess the contribution of each component:

- Removing multi-plan exploration (*w/o MP*) reduced performance by up to 6–8
- Removing feedback-driven refinement (*w/o RF*) caused an even larger drop, up to 10–12

These results confirm that both exploration and refinement are essential. Multi-plan exploration broadens the search space, while refinement helps escape local optima and faulty logic paths.

5.3 Cost and Efficiency

Though PairCoder makes more API calls than single-shot methods, its token usage remains moderate. Compared to other iterative frameworks like Reflexion or Self-debugging, it strikes a better balance between cost and performance [4].

5.4 Error Analysis

An error breakdown on failed test cases shows that *Wrong Answers* dominate (60+%), especially in complex tasks. Runtime errors and timeouts are less common. This suggests that LLMs need continued improvement in logic consistency and input handling rather than syntax or runtime reliability [4].

6 Comparison with Guided Code Generation

The framework for Guided Code Generation proposed by Almorsi et al.[1] introduces a structured, multi-agent system to address limitations in LLMs' compositional reasoning and long-context handling. Instead of relying on prompt iteration or plan-switching like PairCoder, this approach decomposes complex programming tasks into a tree of atomic units, which are then solved and composed in a bottom-up fashion.

6.1 Architectural Scope

Guided Code Generation begins with a *Generalist Agent* that recursively decomposes the input problem into sub-problems, forming a tree where each leaf is a self-contained function. A *Code Agent* then solves each leaf, incorporating testing and validation through a *Tester Agent*. Solutions are composed upwards to form parent nodes, finally reaching a full implementation at the root. In contrast, PairCoder iteratively refines complete plans rather than decomposing them.

6.2 Division of Roles

Guided Code Generation employs at least three agent types: Generalist (for planning), Code (for implementation), and Tester/Critic (for validation and refinement). These agents operate hierarchically and asynchronously. In contrast, PairCoder uses two agents (Navigator and Driver) that work synchronously and collaboratively through strategic planning and reactive execution.

6.3 Performance and Focus

The Guided Code framework shows a 23.79% improvement in Pass@1 on HumanEval using Llama 3.1 8B quantized [1]. This is particularly notable given the small model size and absence of model finetuning. However, PairCoder demonstrates even stronger gains with larger models like GPT-3.5, especially on competitive programming tasks like CodeContest.

6.4 Trade-Off Summary

- **PairCoder**: Agentically agile, uses test feedback to refine holistic plans; excels in low-latency, mid-size problems.
- **Guided Code Generation**: Modular and hierarchical; excels in deep compositional logic and interpretable breakdown of large tasks.

Both frameworks show that decompositional and agentic guidance are crucial for improving LLM coding performance. Their comparison illustrates the importance of structuring agent collaboration based on task complexity, resource constraints, and model capabilities.

7 Discussion

PairCoder demonstrates that agentic collaboration can significantly enhance code generation beyond what is achievable through prompt engineering or single-pass models [4]. Its strength lies in dynamic adaptability—switching strategies when execution feedback indicates stagnation, and incorporating structured reasoning through reflection and planning. The tight feedback loop between the Navigator and Driver allows the system to avoid local minima and arrive at correct solutions more consistently.

However, PairCoder has limitations. Its success depends on the diversity and quality of initial plan generation. If the clustered plans fail to capture the correct logic structure, even refined outputs may fall short. Additionally, the iteration-heavy design introduces computational overhead that can make real-time usage challenging.

On the other hand, Guided Code Generation [1] addresses these issues with a planning tree architecture and bottom-up code synthesis. By ensuring modularity and interpretability, it offers more control over composition and verification. Yet, its complexity increases with task depth, and it may incur latency from excessive function decomposition.

Future development could merge both paradigms: using hierarchical planning from Guided Code Generation to feed robust plan clusters into PairCoder's refinement loop, or applying PairCoder's feedback model to bottom-level components in the Guided tree. Human-in-the-loop feedback and automatic test generation, as emphasized in prior work [2], remain promising extensions for both frameworks.

8 Conclusion and Future Work

PairCoder introduces a novel and effective agent-based paradigm for code generation, inspired by the dynamics of human pair programming. By coupling the strategic planning of a Navigator with the execution and feedback

response of a Driver, PairCoder effectively integrates exploration and refinement. Its key innovations—multi-plan generation, clustering, and feedback-based repair—address limitations of single-pass generation frameworks.

Empirical results show that PairCoder achieves state-of-the-art accuracy on standard benchmarks like HumanEval and CodeContest [4]. These improvements are especially pronounced in tasks requiring adaptability and logical reasoning.

Compared to Guided Code Generation [1], which decomposes tasks hierarchically, PairCoder focuses on plan-level iteration. While both frameworks demonstrate the value of agentic collaboration, their differing strategies highlight opportunities for synergy.

Future work could explore:

- **Hybridization**: Embedding PairCoder's feedback loop within Guided Coder's tree structure to improve robustness at each level of abstraction
- **Human-in-the-loop learning**: Enabling developer intervention in agent decision points [2]
- **Domain transfer**: Applying the framework to structured tasks such as theorem proving or symbolic querying
- **Test suite evolution**: Generating new test cases on failure and reusing them to inform plan selection and error diagnosis

Ultimately, PairCoder and Guided Code Generation represent complementary milestones toward collaborative, interpretable, and reliable AI software engineering systems.

A Appendix A: Example Prompt Template

ReflectPrompt: `"You are given a coding problem. Reflect on it, describe possible inputs, edge cases..."`

PlanPrompt: `"Provide up to 3 solution plans to the problem based on your reflection. Each plan should include a strategy name and high-level description."`

SelectPrompt: `"Choose the most robust and correct plan from the provided options, based on functional correctness and input coverage."`

AnalyzePrompt (Wrong Answer): `"Given the code and test failure, identify the likely cause and suggest how to fix the issue to match expected output."`

B Appendix B: Code Snippet Example

Here is an example output from the Driver agent:

```
def solve(arr):
operations = 0
for i, val in enumerate(arr):
if val > i + 1:
operations += val - (i + 1)
return operations
```

Literatur

[1] A. Almorsi, M. Ahmed, and W. Gomaa. Guided code generation with llms: A multi-agent framework for complex code tasks. *arXiv preprint arXiv:2501.06625*, 2025.

[2] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

[3] S. Hong, X. Zhang, Y. Gu, Y. Li, J. Hao, and Y. Yu. Metagpt: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[4] H. Zhang, W. Cheng, Y. Wu, and W. Hu. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1319–1331. ACM, 2024.