Pair Programming Framework

Seminar on "Machine Learning in Software Eningeering"

Taylan Bapur, Jan Härtrich

Summer 2025

Zusammenfassung

Large Language Models (LLMs) have made notable progress in automatic code generation. However, they often fall short when faced with complex programming tasks that require flexible reasoning, iterative debugging, or adaptive planning. This seminar investigates PairCoder, a multi-agent framework inspired by the practice of human pair programming. The system integrates two LLM agents—a Navigator and a Driver—that work together through multi-plan exploration and feedback-driven refinement to iteratively generate, test, and repair code.

Through experiments on benchmarks such as HumanEval and CodeContest, PairCoder demonstrates a clear improvement in performance compared to traditional prompt-based, single-pass methods. Beyond evaluating its architecture and effectiveness, this work also compares PairCoder to two other recent frameworks: Guided Code Generation, which uses hierarchical decomposition for structured reasoning, and MapCoder, which simulates the full development lifecycle with specialized agents for planning, coding, and debugging.

By examining these frameworks side by side, this report highlights the contrasting strengths of adaptive iteration versus structured decomposition in LLM-based code generation—offering insights into how different agentic strategies handle complexity, context, and correctness in software tasks.

Inhaltsverzeichnis

1 Introduction

Recent years showed a big advancement of Large Language Models (LLMs) and how they have significantly impacted software engineering workflows, particularly in tasks such as code completion, and debugging [5]. While LLMs show strong capabilities in language understanding and pattern recognition, they often struggle in complex code generation, problem solving and deep understanding in code dependencies - especially in single-path generation strategies [2].

To minimze those limitations, H. Zhang, W. Cheng, Y. Wu, and W. Hu. have proposed a multi-agent framework that presents a collaboration between two LLM agents. *PairCoder* is a demonstration of a humanlike pair programming by using two agents: a **Navigator**, responsible for planning and strategy, and a **Driver**, which focuses on code implementation and testing [5]. Through iterative plan generation, execution, and feedback-driven refinement, Paircoder reviews and apdapts to errors and test results for a better resilience than traditional LLMs methods.

Next to the *PairCoder*, we will review two similar, yet different frameworks. *Guided Code Generation with LLMs* and *MapCoder*.

While *Guided Code Generation* focuses on **Generalist Agent**, which breaks down a problem into smaller subtasks, these are then passed by more agents **Code Agents** and to **Tester Agents** for validation using a bottom-up process [1].

On the other hand we review MapCoder, which consists of four agents, a**Retrieval Agent** which creates k new and distinct problems and passes those to the **Planning Agent** to generate multiple plans and assign confidence scores, the chosen plan is used by the **Coding Agent** to implement, then the code is tested and potentially passed to the **Debugging Agent** for code review and debugging.

in this report, we review the design and practical usage of Paircoder, comparing it with Guided Code Generation and MapCoder to better understand their strengths and weakness, by evaluating their architecture, models a performance trade offs. We aim to get an insight into how mulit-agent framework systems can improve the reliability and abilities for automated software development and 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 [5]. Prompt-based enhancements such as Chain-of-Thought (CoT) and Self-Debugging improves reasoning

depth [2], but they still reach their limits when it comes to creating complex logic and code

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 fragile in multi-step tasks. They also lack capacities when outputs fail tests or miss edge cases, as they cannot easily backtrack, revise strategies, or shift plans without external support [5].

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 [5]. 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 structure helps to solve deep tasks that require an deep understand of 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 Navigator sets direction and a Driver executes 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
- Plan selection and code generation
- 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 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 uses 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 generating 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 [5]. 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 [5]. 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 [5]. 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 [5]. 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 [5].

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 [5].

6 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 MapCoder

In this section, we present MapCoder, introduced in "MapCoder: Multi-Agent Code Generation for Competitive Problem Solving" by Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez [4]. MapCoder aims to simulate a full cycle of programming by human developers. It consists of four LLM agents, each designed to emulate a specific stage in the development process: retrieving relevant examples, planning, code generation, and debugging.

7.1 Retrieval Agent

Despite its name, the Retrieval Agent is neither based on external retrieval nor memory-based indexing. Instead, it is prompted to generate k relevant and distinct problems for a given input. The agent performs the following tasks:

- Create k new, distinct problems.
- For each generated problem:
  - Generate a step-by-step solution code.
  - Provide a problem-solving plan.
- Identify the algorithm needed to solve the original problem.

- Write a high-level tutorial on the identified algorithm (without code).

The agent uses a structured XML format in its output to ensure consistency.

Example: Retrieval Agent on Problem #133 (Sum Squares) Let's simulate how this functions using problem #133 (sum squares) from the HumanEval benchmark. The problem description is as follows:

```
"""You are given a list of numbers.
You need to return the sum of squared numbers numbers in the given list,
 round each element in the list to the upper int (Ceiling) first.
 Examples:
 For lst = [1,2,3] the output should be 14
 For lst = [1,4,9] the output should be 98
 For lst = [1,3,5,7] the output should be 84
 For lst = [1.4,4.2,0] the output should be 29
 For lst = [-2.4,1,1] the output should be 6
"""
```

A potential output for this would be:

- Find the sum of all even numbers in the list.

7.2 Planning Agent

This agent serves two functions: generating plans to solve the original problem and assigning confidence scores to those plans.

For plan generation, the LLM is prompted once per retrieved example problem, with each prompt containing:

- Description of the example problem
- The corresponding plan for the example problem
- The algorithm identified for the original problem
- Original problem statement
- Sample input/output pairs

This process results in k diverse, high-level problem-solving plans, without generating code.

In the second stage, each plan is evaluated with the LLM prompted to:

- Explain how solvable the problem is with the given plan
- Assign a confidence score between 0 and 100 reflecting solvability

7.3 Coding Agent

The Coding Agent generates code step-by-step using the chosen plan, the problem description, the identified algorithm, and sample input/output pairs. If the generated code passes all tests, it is returned as the final solution.

7.4 Debugging Agent

The Debugging Agent is invoked if the initially generated code fails to pass all test cases. In its first iteration, it receives the plan, algorithm, and code from the previous step. In subsequent iterations, it also receives and updates the modified plan and code, iteratively improving the solution.

8 Possible Performance Improvements

8.1 Amount of Plans Generated

Experiments with different values of k (number of plans) on HumanEval and CodeContest-test using GPT-3.5-Turbo revealed:

- Strong improvements up to $k = 3$
- Marginal gains beyond $k = 3$
- Over 85% of problems resolve before hitting maximum iteration (10)

Thus, increasing k or the iteration cap may yield modest improvements at higher cost.

8.2 Few-Shot Prompting

Potential improvements may be realized through few-shot prompting techniques, which could surpass zero-shot accuracy. However, the selection of demonstrations for in-context learning poses a significant challenge, which can greatly influence the behavior of LLMs [**?**].

9 Threats to Validity

While the results presented across PairCoder, Guided Code Generation, and MapCoder are promising, several potential threats to validity must be considered:

1. Dataset Overlap and Data Leakage

One known concern is data leakage due to the release timing of model checkpoints relative to benchmark datasets. For instance, DeepSeek-Coder was released after HumanEval was made public, meaning some training overlap is possible. However, as all models under comparison use similar pretrained architectures and were evaluated uniformly, the relative performance trends remain fair and valid across all baselines [5].

2. Generalization Beyond Benchmarks

Benchmarks like HumanEval and MBPP contain problems with clearly defined, relatively small inputs. These may not fully represent real-world software development challenges such as working with APIs, multi-file projects, or performance constraints. Both PairCoder and MapCoder operate within well-scoped, single-function settings. Thus, their generalization to large-scale, industry-grade tasks is untested [1].

3. Prompt Sensitivity and Tuning Bias

The performance of all systems relies heavily on prompt engineering and zero-shot or few-shot prompting quality. For example, PairCoder's multi-plan generation effectiveness can be influenced by the temperature setting and clustering strategy used during plan sampling. Guided Code Generation similarly depends on how well the hierarchical decomposition is constructed by the Generalist Agent [5, 1].

4. Benchmark Leakage and Fair Comparison

Some competitive systems evaluated against HumanEval or MBPP may have indirectly seen similar examples during pretraining (especially proprietary models like GPT-3.5 or Codex). As noted in the Self-Debugging paper, even slight overlap can disproportionately boost accuracy on small test sets. However, studies like PairCoder and MapCoder attempt to mitigate this by using multiple benchmarks (e.g., CodeContest and MBPP+) which are less prone to contamination [5, 4].

5. Evaluation Metrics and Execution Assumptions

All comparisons are made using Pass@1, which assumes correctness is binary (i.e., all test cases must pass). This metric does not reflect partial progress or human readability of code. Furthermore, some frameworks (e.g., Self-Debugging or Guided Code Generation) utilize custom test harnesses or execution environments which may affect reproducibility [1]. The absence of standardized, unified evaluation pipelines across all papers could influence reported results.

6. Model Size and Resource Constraints

Guided Code Generation was evaluated using LLaMA 3.1 8B in int4 quantized form due to resource limits, while PairCoder used GPT-3.5-Turbo and DeepSeek-Coder 33B. These differences in model scale and inference quality can confound performance attribution. A more rigorous evaluation would include ablation studies across identical model families to ensure architectural benefits (rather than scale) drive improvements [5, 1].

7. Iteration Budget and Hyperparameter Variance

PairCoder, Guided Code Generation, and MapCoder all rely on iteration caps (e.g., 10 repair loops or k plans). The choice of these hyperparameters can dramatically alter performance. The reported results often reflect optimally chosen values found through internal tuning, which may not generalize across users or unseen problem types [5, 4].

8. Autonomy vs. Oracle Feedback

In frameworks like MapCoder, all corrections are derived from internal feedback using only sample I/O. In contrast, Self-Debugging assumes access to test execution environments or simulation traces. These differences introduce variance in the robustness of model behavior under noisy or incomplete feedback conditions [1, **?**].

Overall, while PairCoder and its peers advance the field significantly, careful consideration must be given to how benchmarks, model scale, evaluation settings, and feedback assumptions shape outcomes. Future work should adopt stricter experimental protocols, larger and more diverse datasets, and ideally include human evaluation for holistic assessment.

10 Discussion

Through this seminar, we explored how PairCoder applies the principles of pair programming to large language model (LLM)-based code generation. The results and structure presented in the original paper [5] clearly demonstrate that having two collaborative agents—Navigator and Driver—working iteratively on code problems can outperform traditional single-pass methods. One of the key reasons for this success is the framework's ability to switch strategies when it detects stagnation or repeated errors, which is something that most LLMs cannot handle on their own.

The Navigator agent plays an especially important role here. By generating multiple diverse plans and clustering them, it avoids the common problem of LLMs being stuck in a single flawed reasoning path. The Driver, in turn, executes and tests these plans, and feeds back results that guide

further decision-making. This feedback loop is very similar to how human developers test, debug, and revise code based on observed behavior.

That said, PairCoder is not without its limitations. Its success heavily depends on the quality and diversity of the initial solution plans. If the Navigator fails to generate plans that cover a wide range of correct strategies, the system might still get stuck—even if the feedback loop is working correctly. Also, because the framework relies on multiple iterations (especially for complex tasks), it can be computationally expensive and slower than single-shot generation methods. This may limit its real-time applicability in some practical scenarios.

Other frameworks like Guided Code Generation [1] approach the problem differently. Instead of iteration and switching, they focus on decomposing the task into smaller parts in a tree-like structure. This helps especially in tasks that require deep reasoning and compositional logic, such as problems that involve multiple interdependent functions. However, Guided Code also has its challenges, particularly when it comes to managing the complexity of the tree and debugging at different levels of decomposition.

Another interesting alternative is MapCoder [4], which adds more structure by dividing the code generation pipeline into four agents: Retrieval, Planning, Coding, and Debugging. It adds confidence-based plan scoring and internal traversal mechanisms, which makes it more autonomous in some ways. However, like PairCoder, its performance depends a lot on the quality of generated examples and retrieved problem templates.

Finally, Self-Debugging [2] presents a more lightweight and explainability-focused approach. It allows models to reflect on their own failures using a sort of internal dialogue, which can be surprisingly effective even with smaller models. Still, it lacks the strategic planning depth of PairCoder or the structure of Guided Code.

In summary, this comparison shows that different approaches are suited to different types of code generation challenges. PairCoder's strength lies in dynamic iteration and plan refinement; Guided Code is better at structural decomposition; MapCoder is somewhere in between with modular design; and Self-Debugging emphasizes introspection and clarity. Combining these ideas could lead to even better systems in the future.

Despite the impressive results reported in the original paper, it is worth mentioning that our own attempt to test or replicate the PairCoder implementation (from the official GitHub repository) did not succeed. The system either failed to produce correct results for some test problems or encountered runtime issues during execution. This suggests that while the conceptual framework is sound, the practical implementation may require further refinement, better documentation, or additional tuning to work reliably out of the box. It also highlights a common issue in machine learning research: reproducibility remains a challenge, especially for complex multi-agent systems.

11 Conclusion and Future Work

This seminar focused on PairCoder, a novel multi-agent code generation framework inspired by the human practice of pair programming. Throughout my research and analysis, we found that the key strength of PairCoder lies in its iterative structure—especially the way it combines high-level planning (via the Navigator) with low-level implementation and testing (via the Driver). This structure enables it to overcome some of the key weaknesses of large language models when they are used in isolation, such as their inability to correct flawed logic without external help.

The experimental results presented in the original PairCoder paper [5] are quite compelling. On benchmarks like HumanEval and CodeContest, the framework shows significant improvements over standard prompting methods. These gains are particularly notable in complex tasks, where the ability to revise code based on test results is essential. The use of techniques such as multi-plan sampling, clustering, feedback memory, and error-driven repair all contribute to these improvements.

At the same time, PairCoder is not the only attempt at improving LLM-based code generation. During this seminar, I studied two other recent frameworks: Guided Code Generation [1] and MapCoder [4]. Guided Code Generation focuses more on modularity and structured planning by dividing problems into subcomponents that can be solved independently. MapCoder introduces multiple specialized agents that mimic the steps of real-world software development, such as planning, coding, and debugging.

These comparisons lead to the idea that no single framework is universally better. Each has strengths that can be useful in different settings. For instance, PairCoder is well-suited for interactive and logic-heavy tasks that require fast adaptation, while Guided Code Generation may perform better on large, multi-function programs that benefit from hierarchical structure.

Future research could combine the strengths of these different systems to create hybrid models. For example:

- **Hybrid architectures**: One idea is to embed PairCoder's feedback loop within the sub-task trees of Guided Code Generation. This would allow for robust plan-switching at each level of decomposition.
- **Human-in-the-loop learning**: Developers could be allowed to intervene at key points—such as choosing between competing plans or interpreting test results—to improve accuracy and interpretability [2].
- **Cross-domain adaptation**: These agentic frameworks could be applied to non-programming domains like theorem proving, symbolic logic, or natural language query generation.
- **Test suite evolution**: Instead of relying only on predefined public

test cases, models could learn to generate new edge cases or adversarial inputs to test their own solutions more thoroughly.

In conclusion, our report shows that the future of AI-assisted programming is likely to be built on collaboration—between multiple agents, between humans and models, and between different design philosophies. PairCoder represents a strong step in that direction, and it will be exciting to see how these ideas evolve in the years to come.

While studying PairCoder, we also attempted to test the framework ourselves using the official GitHub implementation. Unfortunately, our test did not succeed—either due to code errors, environment setup issues, or limitations in the robustness of the current version. This experience underscores the gap between theoretical innovation and practical deployment. It reinforces the importance of reproducibility and the need for clearer implementation pipelines, especially for educational and research purposes. Future versions of frameworks like PairCoder could benefit from more accessible APIs, minimal working examples, and better compatibility with common development environments.

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] M. A. Islam, M. E. Ali, and M. R. Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. 2024.

[5] 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.