MaintainCoder: Maintainable Code Generation Under Dynamic Requirements

Zhengren Wang [1,3] Rui Ling [1] Chufan Wang [1] Yongan Yu [2]
Sizhe Wang [1] Zhiyu Li [3] Feiyu Xiong [3] Wentao Zhang [1]

[1]Center for Data Science, Peking University
[2]McGill University
[3]Center for LLM, Institute for Advanced Algorithms Research, Shanghai

Outline

Outline

The Only Constant is Change

"The Only Constant in Life is Change." — Heraclitus

- **The Problem**: Modern code generation focuses on initial correctness, but overlooks **maintainability**—the ability to adapt to changing requirements.
- **The Crisis**: This leads to a "maintainability crisis."
  - 60-80% of software costs are from post-deployment maintenance.
  - Real-world disasters (e.g., Knight Capital's $440M loss) often stem from unmaintained legacy code.
- **The Gap**: Current research lacks tools and methods to address this.
  1. No benchmark to measure maintainability under evolving requirements.
  2. No method that systematically applies software engineering principles.
  3. No discussion on the interplay between correctness and maintainability.

Our Contributions

1. **Dynamic Benchmark: MaintainBench**
   The first benchmark to assess code maintainability through requirement evolution cycles.

Our Contributions

1. **Dynamic Benchmark: MaintainBench**
   The first benchmark to assess code maintainability through requirement evolution cycles.

2. **Systematic Generation: MaintainCoder**
   A multi-agent system integrating the Waterfall model, design patterns, and collaboration to produce maintainable code.

Our Contributions

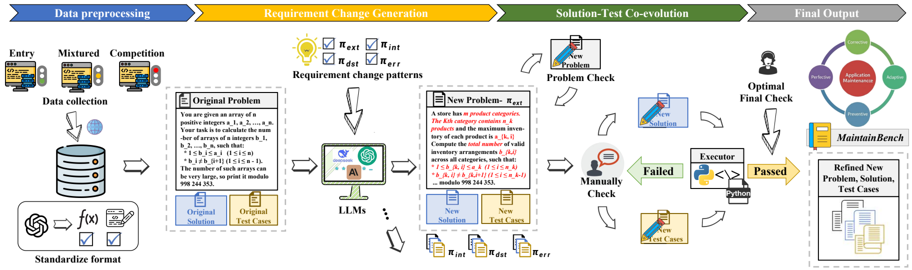
1. **Dynamic Benchmark: MaintainBench**
   The first benchmark to assess code maintainability through requirement evolution cycles.

2. **Systematic Generation: MaintainCoder**
   A multi-agent system integrating the Waterfall model, design patterns, and collaboration to produce maintainable code.

3. **Empirical Insights**
   Experiments show MaintainCoder boosts maintainability (+60%) and initial correctness, while revealing flaws in static metrics.

Outline

Systematic Construction of MaintainBench



You are given an array of n positive integers a_1, a_2, …, a_n. Your task is to calculate the num -ber of arrays of n integers b_1, b_2, …, b_n, such that:
* 1 ≤ b_i ≤ a_i (1 ≤ i ≤ n)
* b_i ≠ b_{i+1} (1 ≤ i ≤ n - 1).
The number of such arrays can be very large, so print it modulo 998 244 353.

A store has *m product categories. The Kth category contains n_k products* and the maximum inventory of each product is a_{k, i} Compute the *total number* of valid inventory arrangements *b_{k,i}* across all categories, such that:
* *1 ≤ b_{k, i} ≤ a_k (1 ≤ i ≤ n_k)*
* *b_{k, i} ≠ b_{k,i+1} (1 ≤ i ≤ n_k-1)*
… modulo 998 244 353.

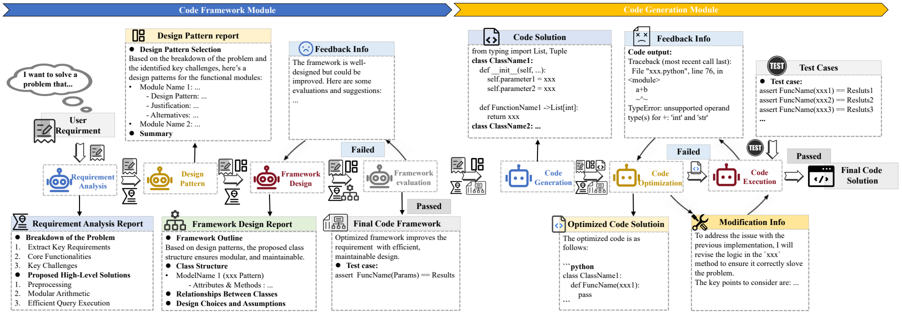
Figure: The construction of MaintainBench, from data selection to quality checks.

Key Idea: Systematically apply four types of requirement changes to existing benchmark problems:

- Functional Extensions
- Interface Modifications
- Data Structure Transformations
- Error Handling Enhancements

Outline

Overview of the MaintainCoder Workflow



Design Pattern report

- **Design Pattern Selection**

Based on the breakdown of the problem and the identified key challenges, here's a design patterns for the functional modules:

- Module Name 1: ...
  - Design Pattern: ...
  - Justification: ...
  - Alternatives: ...
- Module Name 2: ...
- **Summary**

Feedback Info

The framework is well-designed but could be improved. Here are some evaluations and suggestions: ...

Requirement Analysis Report

- **Breakdown of the Problem**
1. Extract Key Requirements
2. Core Functionalities
3. Key Challenges
- **Proposed High-Level Solutions**
1. Preprocessing
2. Modular Arithmetic
3. Efficient Query Execution

Framework Design Report

- **Framework Outline**

Based on design patterns, the proposed class structure ensures modular, and maintainable.

- **Class Structure**
- ModelName 1 (xxx Pattern)
  - Attributes & Methods : ...
- **Relationships Between Classes**
- **Design Choices and Assumptions**

Final Code Framework

Optimized framework improves the requirement with efficient, maintainable design.

- **Test case:**

assert FuncName(Params) == Results

Code Solution

```
from typing import List, Tuple
class ClassName1:
    def __init__(self, ...):
        self.parameter1 = xxx
        self.parameter2 = xxx

    def FunctionName1 ->List[int]:
        return xxx
class ClassName2: ...
```

Feedback Info

Code output:

```
Traceback (most recent call last):
  File "xxx.python", line 76, in
<module>
    a+b
    ~^~
TypeError: unsupported operand
type(s) for +: 'int' and 'str'
```

Test Cases

- **Test case:**

```
assert FuncName(xxx1) == Resluts1
assert FuncName(xxx2) == Resluts2
assert FuncName(xxx3) == Resluts3
...
```

Optimized Code Solutioin

The optimized code is as follows:

~~~
```python
class ClassName1:
    def FuncName(xxx1):
        pass
```
~~~

Modification Info

To address the issue with the previous implementation, I will revise the logic in the `xxx` method to ensure it correctly slove the problem.
The key points to consider are: ...

Figure: Overview of MaintainCoder. MaintainCoder features best practices like the Waterfall model, classical design patterns, and iterative refinement mechanisms for critical stages. 1) The Code Framework Module systematically transforms user requirements into an optimized, maintainable architectural blueprint; 2) The Code Generation Module implements the blueprint into production-ready code, rigorously adhering to both the framework specifications and initial user requirements.

How MaintainCoder Works

Module 1: Code Framework

- **Requirements Analysis Agent**: Decomposes the problem.
- **Design Pattern Agent**: Selects appropriate patterns (e.g., Strategy, Factory) to enhance modularity.
- **Framework Design Agent**: Creates a high-level blueprint.
- **Framework Evaluation Agent**: Critiques and refines the blueprint.

Module 2: Code Generation

- **Code Generation Agent**: Implements the blueprint into functional Python code.
- **Code Optimization Agent**: Executes, debugs, and refines the code against test cases until it is correct and robust.

Outline

Experimental Setup

Baselines

- **Foundation Models**: GPT-4o, DeepSeek-V3, Claude-3.7-Sonnet, etc.
- **Prompting Methods**: Chain-of-Thought (CoT), Self-Planning (Plan).
- **Multi-Agent Systems**: AgentCoder, MapCoder, etc.

Metrics

- **Static Metrics**:
  Maintainability Index (MI $\uparrow$), Cyclomatic Complexity (CC $\downarrow$).
- **Dynamic Metrics (Our Proposal)**:
  - **Post-modification Correctness (Pass@5 $\uparrow$)**
  - **Code Change Volume ($\text{Code}_{\text{diff}}$ $\downarrow$)**
  - **AST Similarity (AST_{sim} $\uparrow$)**

MaintainCoder Excels on Difficult Problems

Model (on CodeContests-Dyn)	Static		Dynamic			
	MI ↑	CC ↓	Pass@5 ↑	AST_{sim} ↑	Code_{diff}^{per} ↓	Code_{diff}^{abs} ↓
GPT-4o-mini	57.8	6.06	24.2	0.661	90.1	16.6
AgentCoder (GPT-4o-mini)	62.5	7.28	18.2	0.629	44.9	18.5
MaintainCoder (GPT-4o-mini)	**65.8**	**2.68**	**32.6**	**0.833**	**23.2**	17.4
DeepSeek-V3	55.7	6.80	26.5	0.718	87.2	17.5
AgentCoder (DeepSeek-V3)	43.8	19.6	16.7	0.670	48.4	28.0
MaintainCoder (DeepSeek-V3)	**63.1**	**3.64**	**37.9**	**0.788**	**43.2**	18.5

- Key Insight: MaintainCoder achieves the best dynamic metrics by a large margin (+60% Pass@5).
- Low Complexity: It keeps Cyclomatic Complexity (CC) extremely low ($\sim$3), indicating simpler, more modular code.
- Agent Pitfall: Other multi-agent systems like AgentCoder may actually *hurt* maintainability (lower Pass@5).

MaintainCoder Excels on Difficult Problems

Model (on CodeContests-Dyn)	Static		Dynamic			
	MI ↑	CC ↓	Pass@5 ↑	AST_{sim} ↑	Code_{diff}^{per} ↓	Code_{diff}^{abs} ↓
GPT-4o-mini	57.8	6.06	24.2	0.661	90.1	16.6
AgentCoder (GPT-4o-mini)	62.5	7.28	18.2	0.629	44.9	18.5
MaintainCoder (GPT-4o-mini)	**65.8**	**2.68**	**32.6**	**0.833**	**23.2**	**17.4**
DeepSeek-V3	55.7	6.80	26.5	0.718	87.2	17.5
AgentCoder (DeepSeek-V3)	43.8	19.6	16.7	0.670	48.4	28.0
MaintainCoder (DeepSeek-V3)	**63.1**	**3.64**	**37.9**	**0.788**	**43.2**	**18.5**

- Key Insight: MaintainCoder achieves the best dynamic metrics by a large margin (+60% Pass@5).
- Low Complexity: It keeps Cyclomatic Complexity (CC) extremely low (∼3), indicating simpler, more modular code.
- Agent Pitfall: Other multi-agent systems like AgentCoder may actually *hurt* maintainability (lower Pass@5).

MaintainCoder Excels on Difficult Problems

Model (on CodeContests-Dyn)	Static		Dynamic			
	MI ↑	CC ↓	Pass@5 ↑	AST_{sim} ↑	Code^{per}_{diff} ↓	Code^{abs}_{diff} ↓
GPT-4o-mini	57.8	6.06	24.2	0.661	90.1	16.6
AgentCoder (GPT-4o-mini)	62.5	7.28	18.2	0.629	44.9	18.5
MaintainCoder (GPT-4o-mini)	**65.8**	**2.68**	**32.6**	**0.833**	**23.2**	**17.4**
DeepSeek-V3	55.7	6.80	26.5	0.718	87.2	17.5
AgentCoder (DeepSeek-V3)	43.8	19.6	16.7	0.670	48.4	28.0
MaintainCoder (DeepSeek-V3)	**63.1**	**3.64**	**37.9**	**0.788**	**43.2**	**18.5**

- Key Insight: MaintainCoder achieves the best dynamic metrics by a large margin (+60% Pass@5).
- Low Complexity: It keeps Cyclomatic Complexity (CC) extremely low (∼3), indicating simpler, more modular code.
- Agent Pitfall: Other multi-agent systems like AgentCoder may actually *hurt* maintainability (lower Pass@5).

Good Design Improves Both

Pass@5 on Original Datasets

Model	APPS	CodeContests	xCodeEval
GPT-4o-mini	44%	18%	46%
MaintainCoder	**48%**	**23%**	**57%**
DeepSeek-V3	66%	48%	75%
MaintainCoder	**69%**	**51%**	**77%**

- **Hypothesis Confirmed**: A focus on maintainability also improves the correctness of the *initial* code. The structured MaintainCoder leads to more robust and correct solutions from the start.
- The benefit is larger for complex problems (e.g., +11% on xCodeEval).

Good Design Improves Both

Pass@5 on Original Datasets

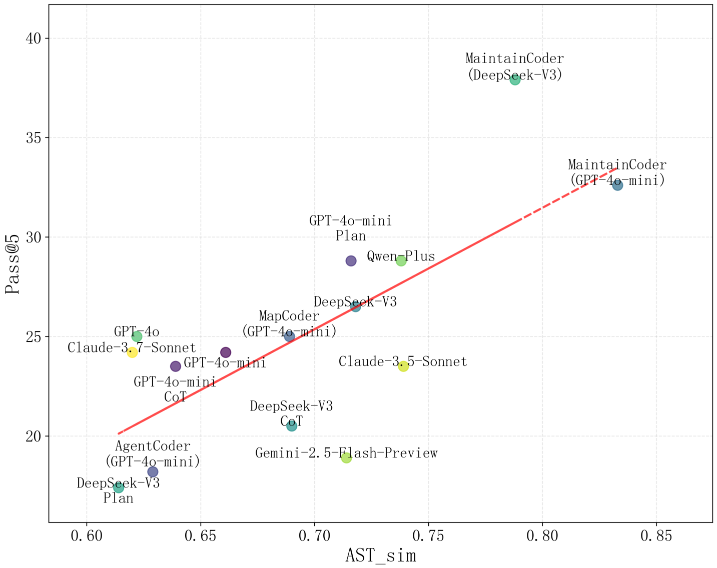
Model	APPS	CodeContests	xCodeEval
GPT-4o-mini	44%	18%	46%
MaintainCoder	**48%**	**23%**	**57%**
DeepSeek-V3	66%	48%	75%
MaintainCoder	**69%**	**51%**	**77%**

- **Hypothesis Confirmed**: A focus on maintainability also improves the correctness of the *initial* code. The structured MaintainCoder leads to more robust and correct solutions from the start.
- The benefit is larger for complex problems (e.g., +11% on xCodeEval).

Static vs. Dynamic Metrics

Our experiments show static metrics (MI, CC) can be inconsistent and misleading. In contrast, our proposed dynamic metrics (Pass@k, Code_{diff}, AST_{sim}) are highly consistent and better reflect maintenance effort.

Static vs. Dynamic Metrics



Outline

Conclusion and Future Work

Summary

- We introduced **MaintainBench**, the first benchmark for evaluating code maintainability under dynamic requirements.
- We proposed **MaintainCoder**, a multi-agent system that leverages software engineering principles to generate highly maintainable code.
- Experiments show MaintainCoder significantly outperforms existing methods in both maintainability and initial correctness.

Future Work

- Expand MaintainBench with more diverse modification types.
- Extend to more complex, repository-level tasks (e.g., SWE-Bench).
- Explore the trade-offs between initial computational cost and long-term maintenance savings.

Thank You!

Resources available at:
`https://github.com/IAAR-Shanghai/MaintainCoder`