CWM: An Open-Weights LLM for Research on Code Generation with World Models

Meta FAIR CodeGen Team (Sep, 2025)

CS8803 LLM - Paper Presentation

En-Chao Liu, Rakshit Nemakallu, Tanvi Shanbhag

Why Coding with LLMs Matters

- Large language models are already widely integrated into software development workflows, where they assist developers with generating code, debugging programs, and navigating large repositories.
- Despite these capabilities, current models still struggle with tasks that require deeper reasoning about how programs execute and how changes to code affect outputs.
- This paper explores whether teaching models about program execution dynamics can improve their ability to reason about code.

Growth of LLM Coding Systems

- Coding models have rapidly evolved from simple autocomplete systems to complex agents capable of editing repositories.
- Systems like Copilot-style assistants can generate entire code blocks and suggest fixes for bugs.
- Despite these advances, models frequently produce code that fails tests or does not run correctly.

Persistent Problems in Code LLMs

- Benchmarks consistently reveal that language models struggle with tasks that require deeper reasoning about program behavior.
- Models often generate code that appears correct syntactically but produces incorrect outputs during execution.
- This suggests that models lack a strong understanding of the semantics of programs.

Root Cause of the Problem

- Most language models treat code the same way they treat natural language, which means they learn by predicting the next token in a sequence.
- This training objective allows models to learn syntax patterns and common coding structures, but it does not explicitly teach them how programs behave during execution.
- As a result, models may produce code that looks correct but fails when it is compiled, tested, or executed.

How Human Programmers Think

- Human programmers reason about code by thinking about how each line changes the state of the program and how different parts of the system interact.
- Developers often simulate program execution mentally or use debugging tools to understand how variables and program states evolve.
- The authors argue that language models should learn similar reasoning abilities about execution rather than relying only on textual patterns.

Key Insight of the Paper

• The authors argue that effective code generation requires models to learn how programs interact with computational environments.

• To achieve this, models must learn both the **syntax of code** and the **dynamics of program execution**.

• This motivates the concept of a **code world model**.

What Is a Code World Model?

• A world model is a system that learns how actions affect the state of an environment.

• In robotics, world models simulate how physical actions affect the physical world.

• In this paper, the environment is a **program execution environment**, and the actions are lines of code.

Overview of the Proposed Model

• The paper introduces **CWM**, a 32-billion-parameter language model designed specifically for reasoning about code execution.

• The model is trained using datasets that explicitly capture program execution and software engineering workflows.

• The goal is to enable the model to simulate and reason about code behavior.

High-Level Contributions

- Introduces a new training paradigm that incorporates program execution traces into language model training.
- Builds large datasets of execution trajectories and agent interactions with software repositories.
- Demonstrates that this training approach improves performance on coding and reasoning benchmarks.

Training Pipeline Overview

- The training process for CWM consists of three main stages: general pretraining, code world modeling mid-training, and post-training with supervised fine-tuning and reinforcement learning.
- General pretraining exposes the model to large amounts of text and code data.
- Mid-training introduces execution traces and agent interaction data that teach the model how programs behave in computational environments.

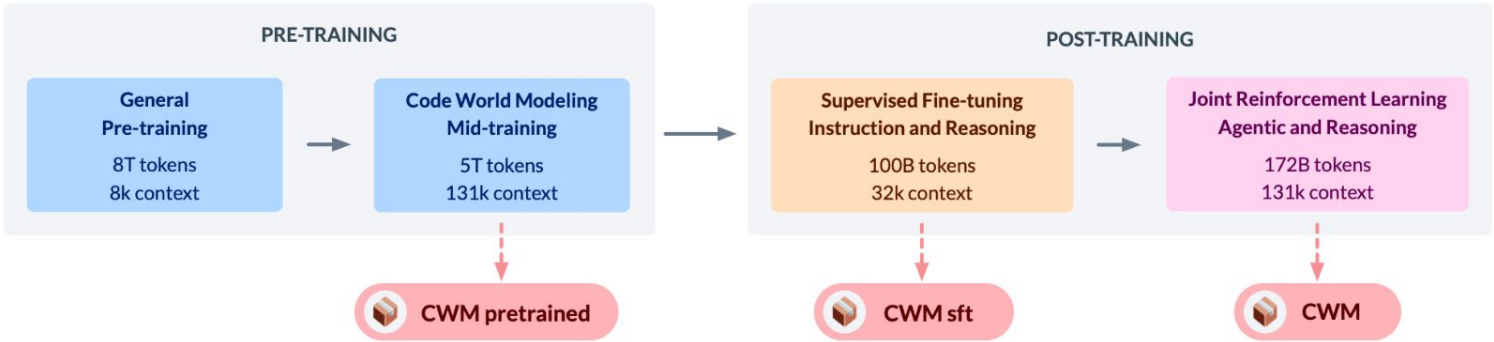


Figure 1 Overview of the CWM training stages and the model checkpoints that we release. We generally report performance of the final CWM (instruct, RL trained) model, except where otherwise stated.

General Pretraining

The model is initially pretrained on approximately **8 trillion tokens** from a mixture of text and code sources.

• Around 30% of the training data consists of programming-related content.

• This stage provides the model with general programming knowledge and language understanding.

Why Mid-Training Is Needed

• Standard pretraining does not teach models how programs behave when executed.

• To address this gap, the authors introduce a mid-training stage focused on **code world modeling datasets**.

• This stage is designed to teach the model the relationship between code actions and program states.

CWM Mid-Training

• During mid-training, the model is trained on about **5 trillion tokens** of specialized datasets.

• These datasets emphasize program execution, debugging workflows, and software engineering tasks.

Post-Training

• After mid-training, the model undergoes supervised fine-tuning and reinforcement learning to improve reasoning and problem-solving abilities.

• These stages help the model learn to solve complex programming tasks and interact with tools in coding environments.

• Reinforcement learning further encourages the model to develop multi-step reasoning strategies.

Model Architecture

• CWM is a dense decoder-only transformer model with approximately **32 billion parameters**.

• The architecture includes grouped-query attention and sliding window attention mechanisms to efficiently handle long contexts.

• The model supports context windows of up to **131,000 tokens**, allowing it to reason over large codebases.

Long Context Attention Design

- The model alternates between layers that use local attention and layers that use global attention.
- Local attention focuses on nearby tokens, which improves computational efficiency when processing long inputs.
- Global attention layers allow the model to reason across large sections of code or repository context.

Execution Trace Training

- A central idea in the paper is to train the model using execution traces that capture how programs run step by step.
- Each trace records the state of program variables before and after executing individual lines of code.
- These traces provide a structured representation of program behavior that the model can learn from.

Python Execution Traces

• The authors generate execution traces by running Python programs and recording interpreter states during execution.

• Each trace forms a sequence of observation-action pairs describing how code modifies program state.

• The dataset contains more than **120 million traced Python functions**, providing large-scale training data for execution reasoning.

```
<|begin_of_text|><|trace_context_start|>
def f(nums):
    a = 0
    for i in range(len(nums)):
        nums.insert(i, nums[a])
        a += 1
    return nums

def main(): # << START_OF_TRACE
    return f([1, 3, -1, 1, -2, 6])
<|frame_sep|><|call_sep|>{}<|action_sep|>def main(): # << START_OF_TRACE
<|frame_sep|>
---END OF PROMPT---
<|line_sep|>{}<|action_sep|> return f([1, 3, -1, 1, -2, 6])
<|frame_sep|><|call_sep|>{"nums": "[1, 3, -1, 1, -2, 6]"}<|action_sep|>def f(nums):
<|frame_sep|><|line_sep|>{"nums": ".."}<|action_sep|> a = 0
<|frame_sep|><|line_sep|>{"nums": "..", "a": "0"}<|action_sep|> for i in range(len(nums)):
<|frame_sep|><|line_sep|>{"nums": "..", "a": "..", "i": "0"}<|action_sep|> nums.insert(i, nums[a])
<|frame_sep|><|line_sep|>{"nums": "[1, 1, 3, -1, 1, -2, 6]", "a": "..", "i": ".."}<|action_sep|> a += 1
<|frame_sep|><|line_sep|>{"nums": "..", "a": "1", "i": ".."}<|action_sep|> for i in range(len(nums)):
<|frame_sep|><|line_sep|>{"nums": "..", "a": "..", "i": "1"}<|action_sep|> nums.insert(i, nums[a])
<|frame_sep|><|line_sep|>{"nums": "[1, 1, 1, 3, -1, 1, -2, 6]", "a": "..", "i": ".."}<|action_sep|> a += 1
<|frame_sep|><|line_sep|>{"nums": "..", "a": "2", "i": ".."}<|action_sep|> for i in range(len(nums)):
<|frame_sep|><|line_sep|>{"nums": "..", "a": "..", "i": "2"}<|action_sep|> nums.insert(i, nums[a])
<|frame_sep|><|line_sep|>{"nums": "[1, 1, 1, 1, 3, -1, 1, -2, 6]", "a": "..", "i": ".."}<|action_sep|> a += 1
<|frame_sep|><|line_sep|>{"nums": "..", "a": "3", "i": ".."}<|action_sep|> for i in range(len(nums)):
<|frame_sep|><|line_sep|>{"nums": "..", "a": "..", "i": "3"}<|action_sep|> nums.insert(i, nums[a])
<|frame_sep|><|line_sep|>{"nums": "[1, 1, 1, 1, 1, 3, -1, 1, -2, 6]", "a": "..", "i": ".."}<|action_sep|> a += 1
<|frame_sep|><|line_sep|>{"nums": "..", "a": "4", "i": ".."}<|action_sep|> for i in range(len(nums)):
<|frame_sep|><|line_sep|>{"nums": "..", "a": "..", "i": "4"}<|action_sep|> nums.insert(i, nums[a])
<|frame_sep|><|line_sep|>{"nums": "[1, 1, 1, 1, 1, 1, 3, -1, 1, -2, 6]", "a": "..", "i": ".."}<|action_sep|> a += 1
<|frame_sep|><|line_sep|>{"nums": "..", "a": "5", "i": ".."}<|action_sep|> for i in range(len(nums)):
<|frame_sep|><|line_sep|>{"nums": "..", "a": "..", "i": "5"}<|action_sep|> nums.insert(i, nums[a])
<|frame_sep|><|line_sep|>{"nums": "[1, 1, 1, 1, 1, 1, 1, 3, -1, 1, -2, 6]", "a": "..", "i": ".."}<|action_sep|> a += 1
<|frame_sep|><|line_sep|>{"nums": "..", "a": "6", "i": ".."}<|action_sep|> for i in range(len(nums)):
<|frame_sep|><|line_sep|>{"nums": "..", "a": "..", "i": ".."}<|action_sep|> return nums
<|frame_sep|><|return_sep|><|action_sep|> return nums<|arg_sep|>"[1, 1, 1, 1, 1, 1, 1, 3, -1, 1, -2, 6]"
<|frame_sep|><|return_sep|><|action_sep|> return f([1, 3, -1, 1, -2, 6])<|arg_sep|>"[1, 1, 1, 1, 1, 1, 1, 3, -1, 1, -2, 6]"<|frame_sep|>
```

(a) CWM Execution Trace Prediction (correct prediction).

```
Given a python code function and an assert statement containing a specific input, provide the assertion with the exact literal output that the function returns with that input. Do not
       include any mathematical expressions or function calls-only the final literal value. Your response should be solely the assertion, enclosed within [ANSWER] and [/ANSWER] tags.
[SKIPPED EXAMPLE FOR ILLUSTRATION PURPOSES]
Python function:
'''
def f(nums):
    a = 0
    for i in range(len(nums)):
        nums.insert(i, nums[a])
        a += 1
    return nums
assert f([1, 3, -1, 1, -2, 6]) == ??
'''
----END OF PROMPT---
Okay, let's see. I need to figure out what the function f does when given the list [1, 3, -1, 1, -2, 6]. The function starts by initializing a variable a to 0. Then it loops over the
       range of the length of nums. But wait, the length of nums is 6 initially. However, inside the loop, it's inserting elements into nums, which will change the length of the list
       as the loop progresses. That might affect how many times the loop runs.
Let me break it down step by step. The initial list is [1, 3, -1, 1, -2, 6]. The variable a starts at 0. The loop runs for i in range(len(nums)), which is range(6) initially. So i will
       take values 0 through 5.
First iteration (i=0):
- nums.insert(i, nums[a]) -> insert nums[0] (which is 1) at position 0. The list becomes [1, 1, 3, -1, 1, -2, 6]. Then a increments to 1.
Second iteration (i=1):
- Now, the list has 7 elements. But the loop was set up with range(6), so i is still 1. We insert nums[a] (nums[1] is 1) at position 1. The list becomes [1, 1, 1, 3, -1, 1, -2, 6]. a
       becomes 2.
Third iteration (i=2):
- Insert nums[2] (which is 1) at position 2. The list is now [1, 1, 1, 1, 3, -1, 1, -2, 6]. a is 3.
Fourth iteration (i=3):
- Insert nums[3] (3) at position 3. The list becomes [1, 1, 1, 1, 3, 3, -1, 1, -2, 6]. a is 4.
Fifth iteration (i=4):
- Insert nums[4] (-1) at position 4. The list is [1, 1, 1, 1, 3, -1, 3, -1, 1, -2, 6]. a is 5.
Sixth iteration (i=5):
- Insert nums[5] (1) at position 5. The list becomes [1, 1, 1, 1, 3, 1, -1, 3, -1, 1, -2, 6]. a is 6.
After these six iterations, the loop ends because i has gone through 0 to 5. The final list is [1, 1, 1, 1, 3, 1, -1, 3, -1, 1, -2, 6]. Wait, but let me double-check each step to make
       sure I didn't make a mistake.
Wait a minute, when we insert elements, the original elements shift to the right. Let me re-examine each step carefully.
Original list: [1, 3, -1, 1, -2, 6]
First iteration (i=0, a=0):
- Insert nums[0] (1) at position 0. The list becomes [1, 1, 3, -1, 1, -2, 6]. a becomes 1.
Second iteration (i=1, a=1):
- Insert nums[1] (1) at position 1. The list is now [1, 1, 1, 3, -1, 1, -2, 6]. a becomes 2.
Third iteration (i=2, a=2):
- Insert nums[2] (1) at position 2. The list becomes [1, 1, 1, 1, 3, -1, 1, -2, 6]. a becomes 3.
Fourth iteration (i=3, a=3):
- Insert nums[3] (3) at position 3. The list is [1, 1, 1, 1, 3, 3, -1, 1, -2, 6]. a becomes 4.
Fifth iteration (i=4, a=4):
- Insert nums[4] (-1) at position 4. The list becomes [1, 1, 1, 1, 3, -1, 3, -1, 1, -2, 6]. a becomes 5.
Sixth iteration (i=5, a=5):
- Insert nums[5] (1) at position 5. The list is [1, 1, 1, 1, 3, 1, -1, 3, -1, 1, -2, 6]. a becomes 6.
Yes, that seems correct. So the final list returned by the function is [1, 1, 1, 1, 3, 1, -1, 3, -1, 1, -2, 6].
</think>
[ANSWER]
assert f([1, 3, -1, 1, -2, 6]) == [1, 1, 1, 1, 3, 1, -1, 3, -1, 1, -2, 6]
[/ANSWER]
```

(b) CWM Reasoning Trace (wrong prediction).

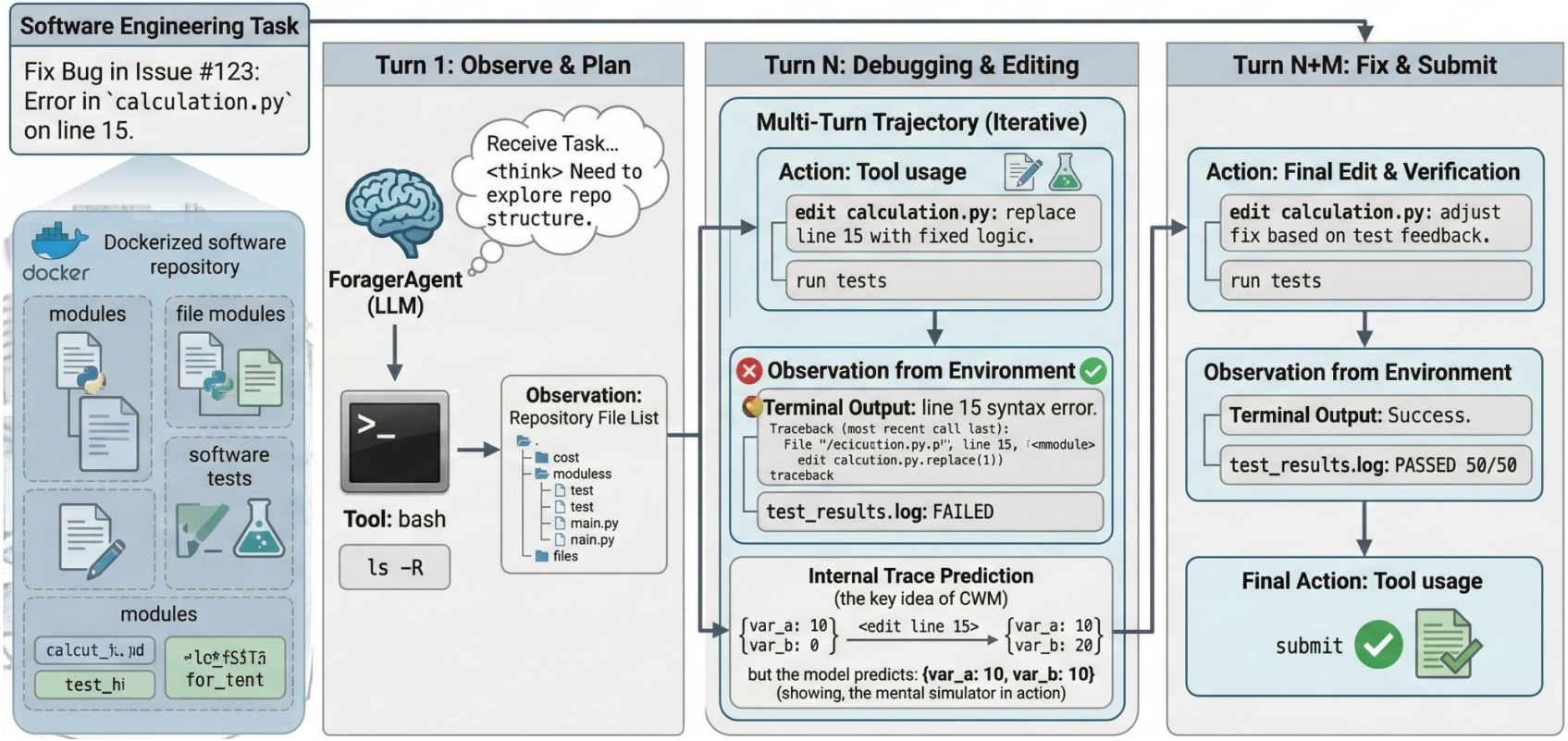
Sources of Execution Data

- Execution traces are collected from several sources, including standalone Python functions and competitive programming solutions.
- The authors also run unit tests on real software repositories to capture realistic program behavior.
- These diverse sources help the model learn execution patterns across different types of coding tasks.

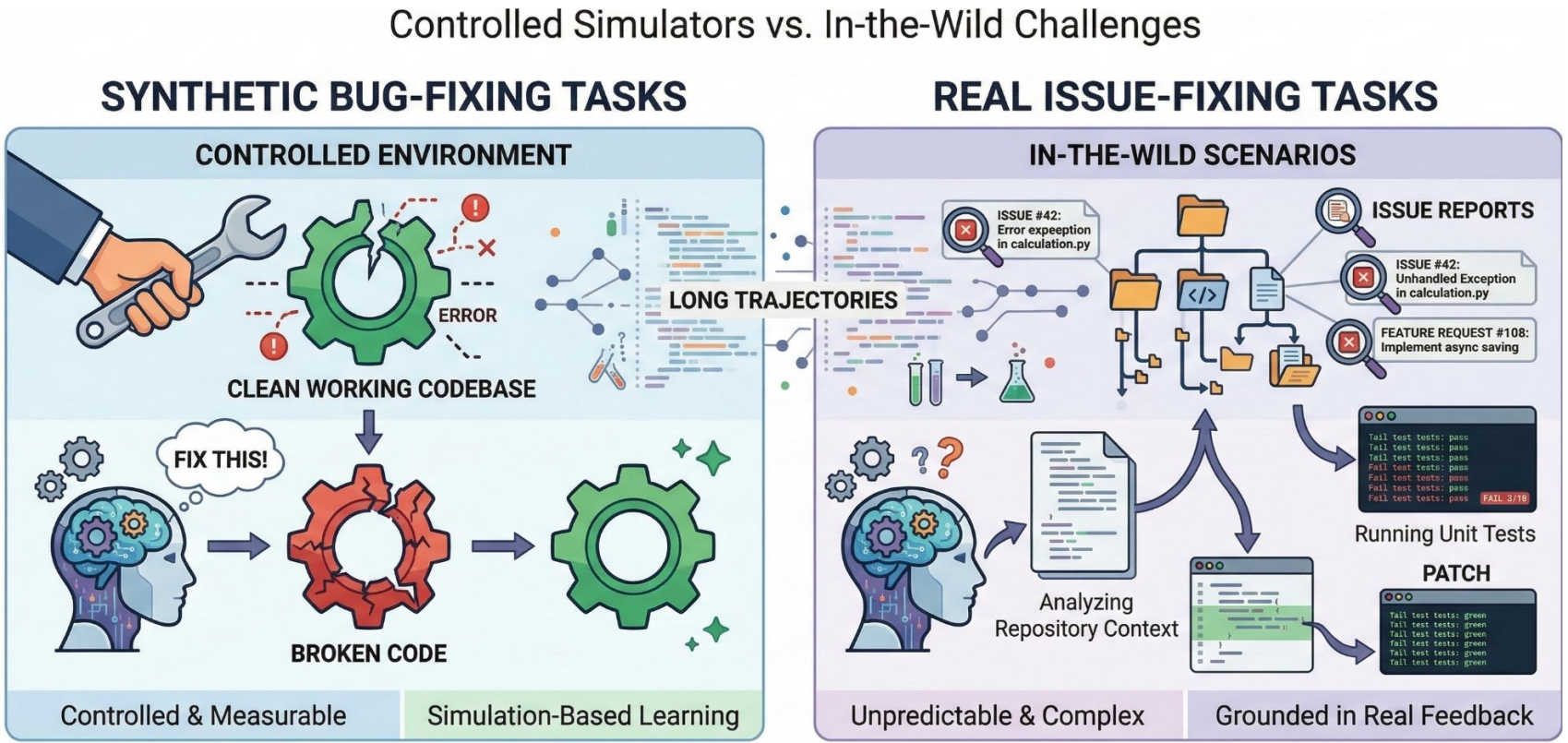
Agent Interaction Data

- In addition to execution traces, the authors collect interaction data using a system called **ForagerAgent**.
- This agent interacts with software repositories by editing files, running shell commands, and navigating codebases.
- These trajectories simulate realistic debugging and development workflows.

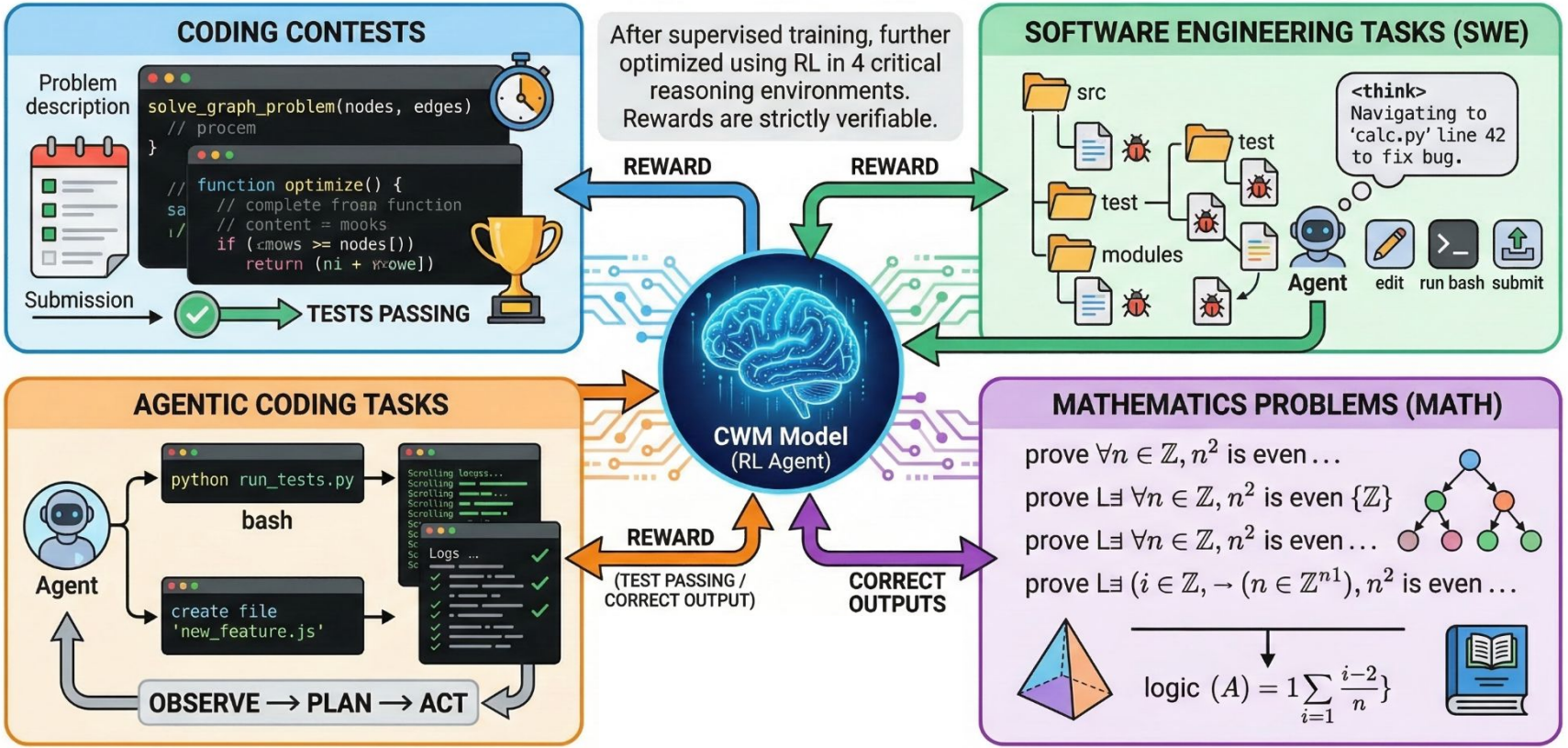
How the Agent Works



Type of Agent Tasks

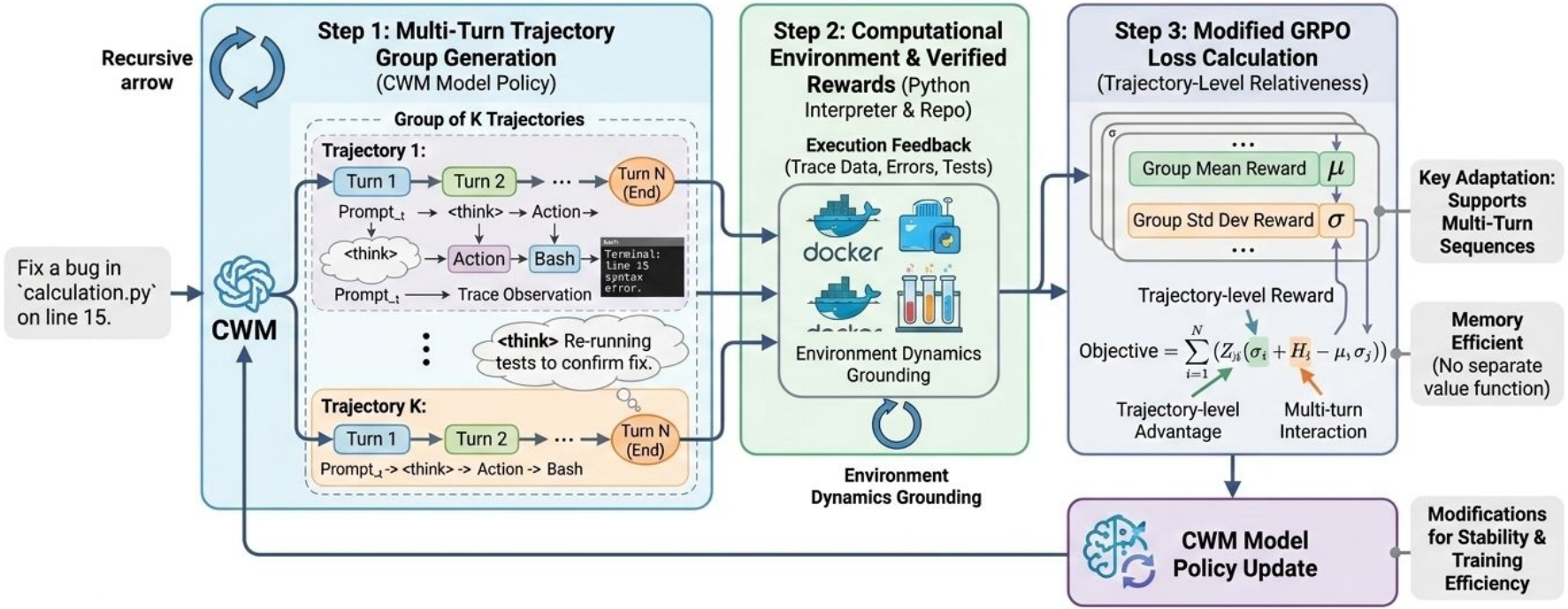


Reinforcement Learning Stage

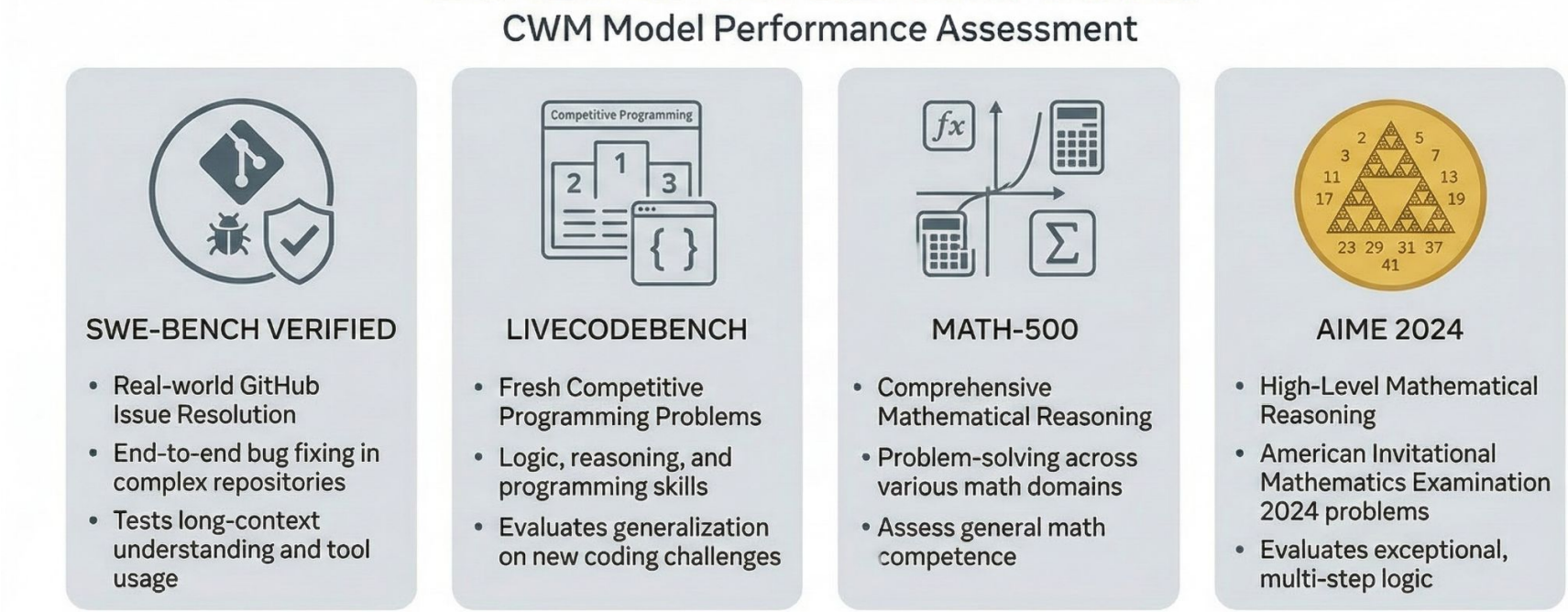


Reinforcement Learning Algorithm

- Modified **Group Relative Policy Optimization (GRPO) training** for CWM paper



Evaluation Benchmarks



These tasks test the model's ability to solve a real programming problems and complex reasoning challenges.

Benchmark Results

General Benchmarks (LCB, Math, AIME)

Model	LCBv5	LCBv6	Math-500	AIME24	AIME25
Magistral-Small-2509-24B	70.0	61.6	--	86.1	77.3
Qwen3-32B	65.7	61.9	97.2	81.4	72.9
gpt-oss-20B (low)	54.2	47.3	--	42.1	37.1
gpt-oss-20B (med)	66.9	62.0	--	80.0	72.1
CWM	**68.6**	**63.5**	**96.6**	**76.0**	**68.2**

Code Synthesis Benchmark (SweBench Verified)

Model	SweBench Verified (%)
Devstral-1.1-2507-24B	53.6
Qwen3-Coder-32B	51.6
gpt-oss-20B (low)*	37.4
gpt-oss-20B (med)*	53.2
gpt-oss-20B (high)*	60.7
CWM	**53.9**
CWM + tts	**65.8**

* **Note:** gpt-oss use a custom subset of 477 problems for SweBench evaluation.

Ablation Evidence for the World-Model Claim

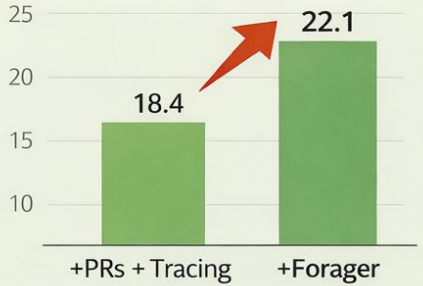
1. Tracing improves Code Reasoning (CruxEval-O)

Execution traces help models learn **program semantics.**

Tracing substantially improves code-output **reasoning.**

2. ForagerAgent improves Agentic SE (SBV pass@1)

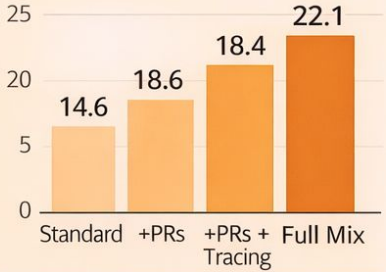
Environment interaction improves **agentic problem solving.**



Adding **ForagerAgent** further improves SWE-bench performance.

3. Combined Mid-training Performs Best (SBV pass@1)

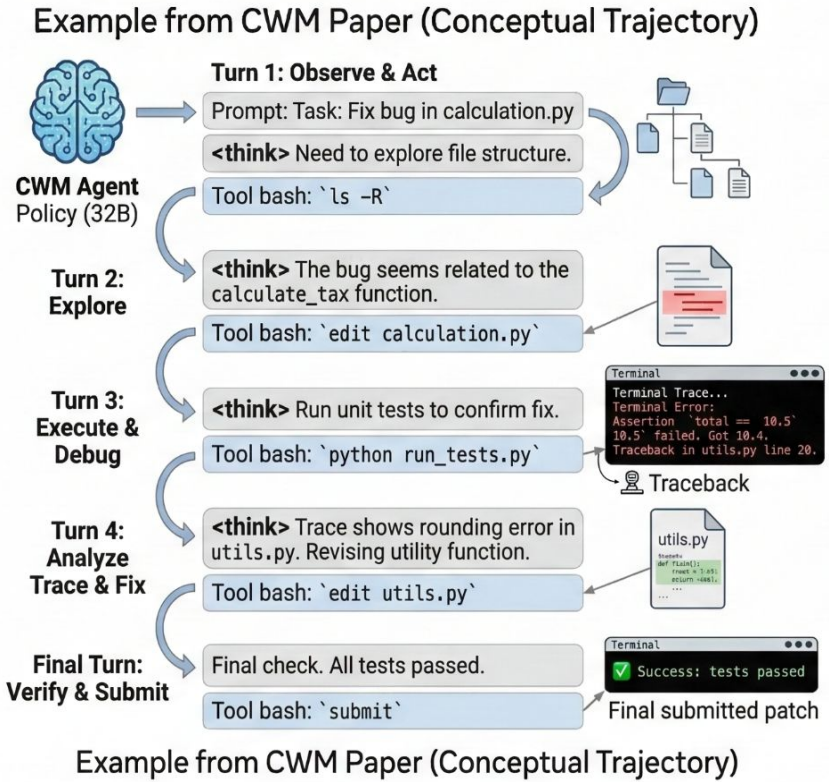
The best downstream performance comes from **combining all** components.



Combining PRs, tracing, and **ForagerAgent** yields the strongest SBV pass@1.

Conclusion: These ablations suggest that modeling computational dynamics improves downstream pformance.

Example of Agentic Coding



- The model can generate code, run tests, analyze outputs, and revise its solution based on feedback.
- This behavior resembles how developers iteratively debug and refine programs.
- The authors suggest that execution-aware models may enable more reliable coding agents.

What CWM Truly Contribute

My framing:

- Not Just Another SOTA Model An open-weights research vehicle for the community.
- Shifting the Paradigm From modeling static syntax to simulating computational dynamics.
- The Ultimate Value Enables rigorous, mechanistic studies on environment-aware AI.

The “World Model” Illusion?