Lingxi V1.5 Technical Report

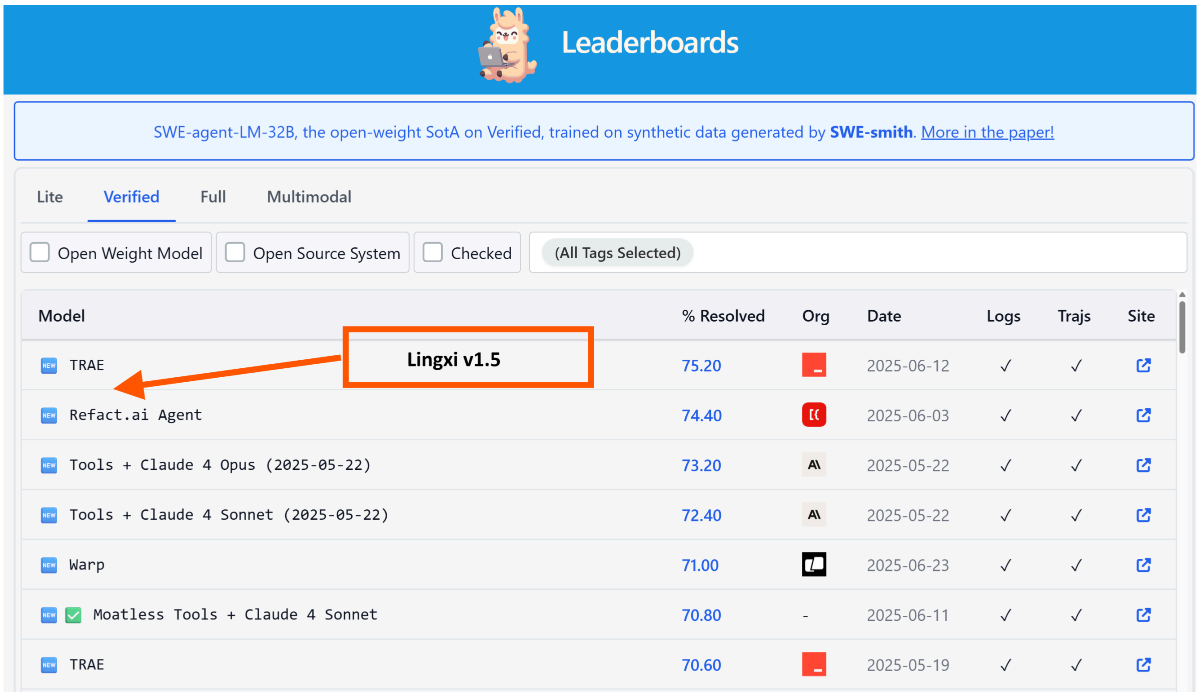
Implementation on SWE-bench – Version 1.5 – 20 July 2025

This is a brief technical report, our papers will come out soon.

Introduction

Lingxi is an open-source, multi-agent framework designed to automate a broad range of software-engineering tasks—from code review and documentation generation to automated program repair.

This report presents our use of Lingxi for automated issue resolving on the SWE-bench-Verified benchmark. By leveraging history development knowledge to guide agents for test time scaling and decomposing the repair workflow into specialized agents, Lingxi achieves a **74.60% issue-resolution rate (373/ 500) on the SWE-bench-Verified benchmark**, ranking **#2** among all models. We detail the system design, tool design, experimental results below.


Leaderboards
SWE-agent-LM-32B, the open-weight SotA on Verified, trained on synthetic data generated by SWE-smith. More in the paper!
Lite
Verified
Full
Multimodal
Open Weight Model
Open Source System
Checked
(All Tags Selected)
Lingxi v1.5

Model	% Resolved	Org	Date	Logs	Trajs	Site
TRAE	75.20		2025-06-12	✓	✓	
Refact.ai Agent	74.40		2025-06-03	✓	✓	
Tools + Claude 4 Opus (2025-05-22)	73.20		2025-05-22	✓	✓	
Tools + Claude 4 Sonnet (2025-05-22)	72.40		2025-05-22	✓	✓	
Warp	71.00		2025-06-23	✓	✓	
✓ Moatless Tools + Claude 4 Sonnet	70.80	-	2025-06-11	✓	✓	
TRAE	70.60		2025-05-19	✓	✓	

2 System Overview

2.1 Motivation for using history development knowledge

While reviewing the SWE-Bench leaderboard, we noticed two common strategies among the top solutions. The first is simply upgrading the base model, for example moving from Sonnet 3.5 to Sonnet 4. The second is test-time sampling, where the system generates many candidate patches and then uses a critic model to select the best one. Both strategies increase cost and latency, and they rely heavily on model capacity.

Our experiments suggest that once a model chooses a clear edit plan, the resulting code edit usually works as long as the root cause is correctly located and the fix follows project conventions. Therefore, instead of sampling many patches, we focus on precise root-cause analysis.

We built a historical development knowledge module that searches the project's history for similar issues and their patches. We reverse-engineer these patches' generation steps to extract the underlying development knowledge. We then inject that knowledge as a prior into several specialized Problem Decoder agents, each of which analyzes the issue from a different perspective according to the development knowledge. An Aggregate module merges these analyses into a single, comprehensive issue report. Using this report, the Solution Mapper and Problem Solver generate one accurate patch without producing multiple candidates.

2.2 Motivation for a Multi-Agent Approach

Single-agent pipelines that keep the entire dialogue in one context inevitably suffer from **context dilution**: by the time the model reaches the code-editing step, earlier discussion tokens dominate the prompt, making it harder for the LLM to focus on the actual diff. Lingxi avoids this by splitting the workflow into compact, purpose-built agents that each receive only the information they need.

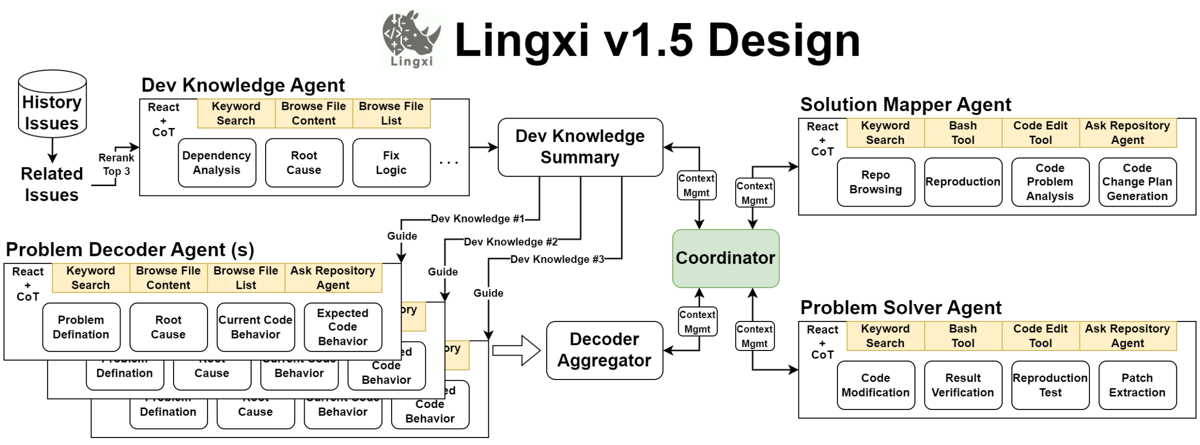
2.3 Design Challenges

A recent Berkeley study reports that naive multi-agent orchestration frequently underperforms single-agent baselines because of

1. **Vague task specifications**,
2. **Unclear responsibility boundaries**, and
3. **Chaotic memory or state management**.

We observed the same pitfalls while prototyping Lingxi and introduced targeted mitigations described below.

2.4 Lingxi Architecture



Agent	Core Question Answered	Responsibility
Dev Knowledge	Which related development knowledge can we leverage?	Find related history issue, reversely retrieve development knowledge, high level summary
Problem Decoder	*What* is wrong and *where* is it?	Locate the root cause, report the failing scenario, and pinpoint files / functions.

Agent	Core Question Answered	Responsibility
Solution Mapper	*How* do we fix it and *why* will that work?	Produce an ordered, line-granular patch plan.
Problem Solver	*Do* the edit.	Apply the patch, run the full test suite, and capture logs.
coordinator	—	Govern agent lifecycles, enforce tool contracts, shrink context, and mediate messages.

Unlike frameworks that imitate human roles (e.g., *developer*, *manager*), Lingxi's roles are derived from **task decomposition** rather than social division of labour. Because every agent is backed by the *same* LLM, we gain little from "specialization"; the win comes from narrowing each agent's task domain so the prompt is shorter, task is easier and the evaluation rubric is crisp.

2.5 Optimizations That Matter

1. **Crystal-clear contracts** – each agent gets a one-page interface spec listing mandatory inputs, expected outputs, and the *only* tools it may call.
2. **Agent messages compression** – after every agents, the coordinator compress conversation history to thinking, observations and actions, discarding verbose function-call results. This help us retain useful insights between agents without information loss.
3. **Explicit awareness of other agents** – every prompt reminds the agent of the team composition and urges it to stay within scope.

2.6 Beyond SWE-bench

Lingxi's agent prompts, tool abstractions, and coordinator skeleton are **task-agnostic**. Lingxi is still under develop, in our vision, porting the framework to other SE tasks like code review, documentation generation requires minimal effort, like swapping task-specific prompts and—if necessary—adding domain-specific tools. We plan to implement Lingxi's on other SE tasks soon.

3 Tooling Design

Lingxi follows a **"minimal tool set, maximal information"** philosophy. We aim to provide accurate, sufficient information to LLM for each tool calling result, so that reduce the burden/steps for LLM. Below are the tools we design for SWE-Bench, all tools are exposed through structured function-calling so the LLM can chain them autonomously.

Tool	Purpose	Key Features
`view_directory`	Explore repository tree	Adaptive depth: prints deeper until a **file-count cap** is hit.
`search_files_by_keywords`	Grep-like semantic search	Multi-keyword queries via *ripgrep*; returns line numbers.

Tool	Purpose	Key Features
`view_file_content`	Inspect file text	Show file content, auto-truncates long files and appends a file structure.
`view_file_structure`	Summarise oversized files	**Invisible** to LLM; automatically injects when needed.
`str_replace_editor`	Apply code edits	Inspired by Anthropic *computer*, Aider, and OpenHand; `undo` removed for simplicity.
`ask_repository_agent`	Repo level Q&A bot	Allow the model to raise questions for the repository, the underling agent are build with an RAG.

4 Results

4.1 End-to-End Performance

Metric	Score
Resolved issues	**373 / 500** (74.60 %)
SWE-bench-Verified OSS leaderboard	**2rd**
SWE-bench-Verified overall leaderboard	**2rd**

5 Discussion

5.1 When Does Multi-Agent Help?

Our results confirm Berkeley's observation that naïve multi-agent systems often trail single-agent baselines. The key to outperforming them is **precise task scoping plus memory hygiene**. Removing large function-call dumps was especially impactful, reducing confusion and token cost.

References

1. **SWE-bench:** "Swe-bench: Benchmarking LLMs on Software Bug Fixing," *arXiv*, 2024.
2. **Google Co-Scientist:** "Towards an AI co-scientist" *arXiv*, 2023.