ACoder - Achieving State-of-the-Art Performance on SWE-bench Verified

1. Abstract

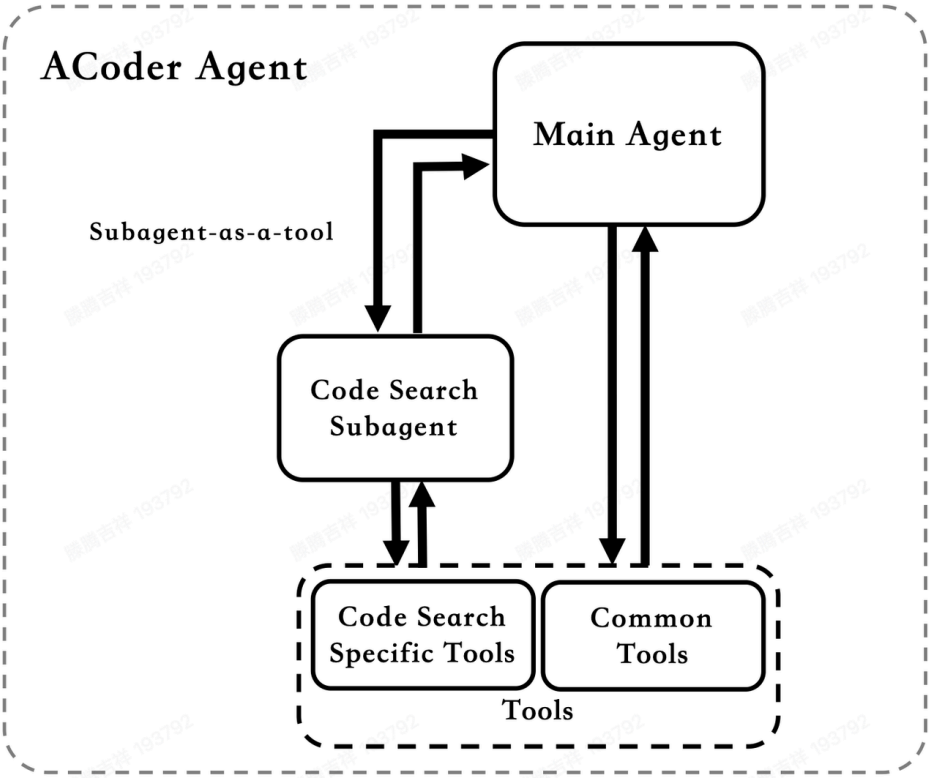
We present ACoder, an agentic system that extends Cline's work and powered by four leading foundation models—Anthropic's Claude 4.0 Sonnet, Claude 4.1 Opus, OpenAI's GPT-5, and Google's Gemini 2.5 Pro 0617. It employs a streamlined subagent architecture, following the philosophy we call *Subagent-as-a-Tool*.

As of now, ACoder has achieved a **76.4%** resolved task rate on SWE-bench Verified, poised to take the top spot on the leaderboard, showcasing the strength of its architecture and methodology.

Inspired by Trae, we adopt a two-stage approach: Generation → Selection, resulting in an improvement of approximately 10% in the problem-solving rate compared to the Single Attempt baseline.

2. Architecture

ACoder Agent is designed for autonomous software engineering, built upon a subagent architecture, equipped with a versatile set of tools to break down complex tasks.



One key aspect of our streamlined multi-agent design is the *"Subagent-as-a-Tool"* philosophy. We encapsulate domain-specific capabilities into subagents, each running in its own independent context window. Treating these subagents as callable tools within the Main Agent's toolset maintains consistency with standard tool usage and enables efficient delegation of specialized functions—such as assigning repository analysis to the Code Search Subagent. This allows the Main Agent to focus on high-level strategy and decision-making.

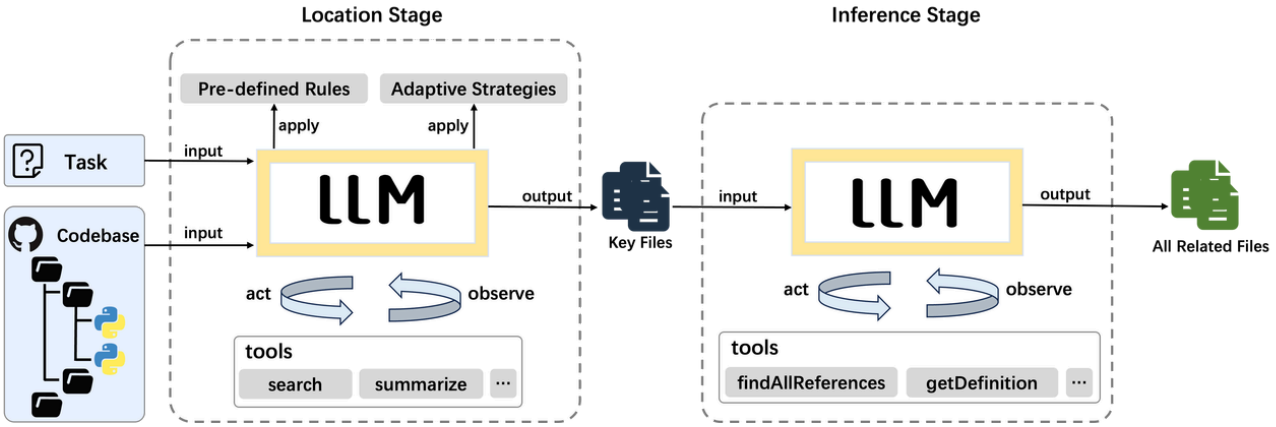
Agent uses a simple, limited set of tools for task completion. General-purpose tools, including file readers, writers, and command executors, are made available to the Main Agent. Furthermore, we have developed a specialized set of tools for the Code Search Subagent. This clear separation ensures that each agent operates with a lean, relevant set of capabilities, preventing the Main Agent's action space from becoming cluttered with overly specialized functions and reinforcing its role as a high-level coordinator.

The Code Search Subagent

In practical software engineering—particularly in problem-solving scenarios—accurately locating all relevant code files and segments is the most critical step. However, when working with large codebases, LLMs frequently struggle to precisely identify the complete scope of a task, primarily due to their limited context window.

We propose **DeepDiscovery**, an innovative subagent specifically designed for code search. Like a human developer, it can perform deep exploration of the repository to

discover all code files relevant to the task.



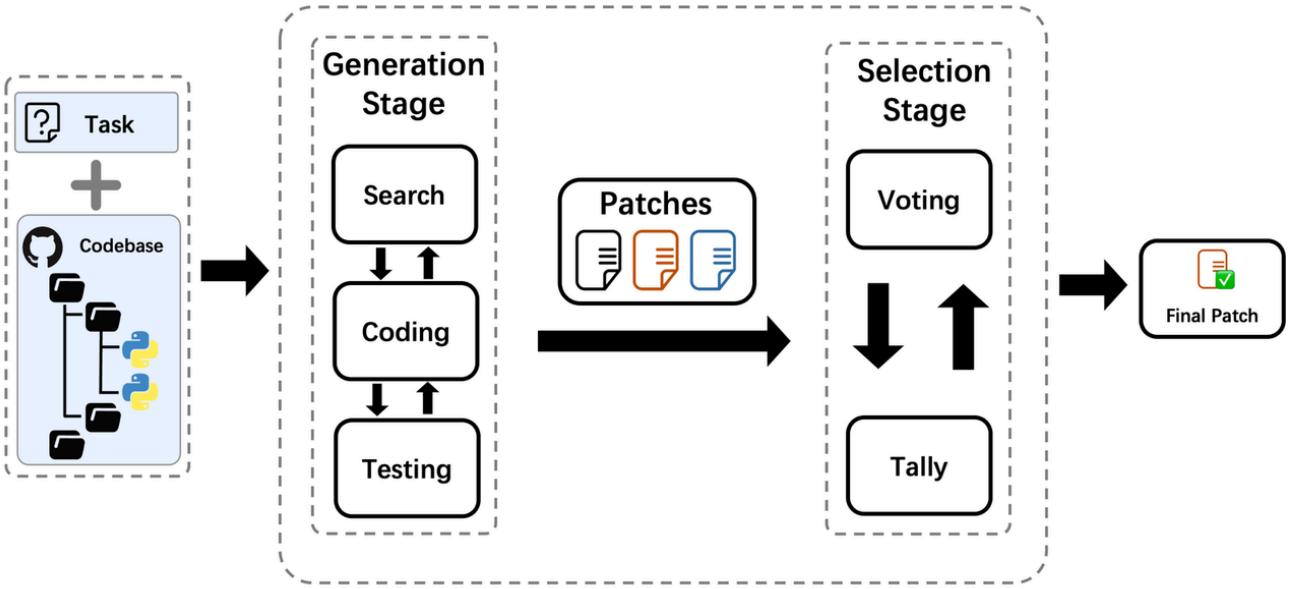
DeepDiscovery adopts a two-stage approach—**Location** followed by **Inference**—reflecting how human developers tackle structured, dependency-rich codebases. Developers typically begin by identifying a few key files related to the problem, then progressively trace dependency relationships to uncover all relevant code and understand its interconnections before making changes.

1. **Location Stage** — The LLM applies Pre-defined Rules—such as filenames (init, index, main) or positions in the dependency graph—alongside Adaptive strategies that adjust code summarization levels based on repository size. Using tools like search and summarize, it pinpoints the key files to investigate.
2. **Inference Stage** — The LLM follows these structural relationships using tools such as findAllReference and getDefinition to iteratively gather related files. Inspired by the "deep research" methodology, the LLM integrates this information with reasoning to determine if sufficient context has been collected, and continues until it determines all relevant code has been identified.

The final output is a set of related files, the reasons they were selected, and their relationships, serving as the foundation for the next coding step.

3. Methodology

The task-handling process of our system comprises two stages: 1) Generation and 2) Selection. We will describe each stage in detail below.



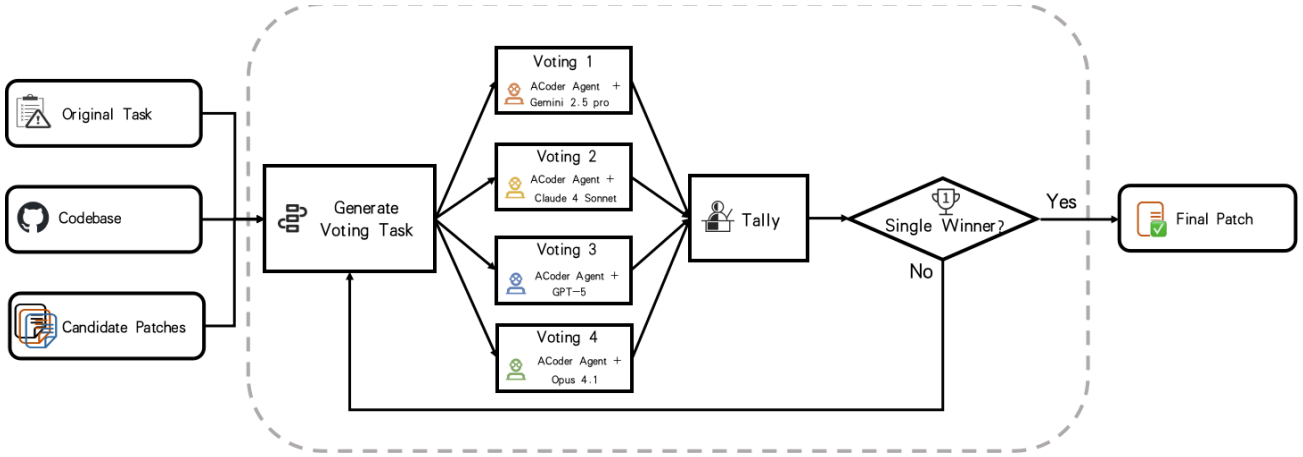
Stage 1: Generation

The **Generation** stage follows a ReAct-style code search → coding → testing iteration, driven by the ACoder Agent.

- **Code Search** — As described above, it leverages the **Code Search Subagent**, which finds all code relevant to the task.
- **Coding** — Several optimizations have been applied. For example, we found that some tasks failed because the Python code generated by the LLM was incompatible with the Python version required by the project. Therefore, the agent checks the project's configuration files to determine the minimum supported Python version before generating any code.
- **Testing** — The agent identifies all existing tests in the repository, analyzes them alongside the task, and ensure that code changes do not break current tests. If needed, it updates or extends the test suite before proceeding to run the tests.

Three models—Claude 4.0 Sonnet, GPT-5, and Gemini 2.5 Pro 0617—are used to generate three candidate patches, one per model. The final patch is then selected from these candidates in the subsequent **Selection** stage.

Stage 2: Selection



After the **Generation** stage, the system proceeds to the **Selection** stage, which determines the optimal patch from the set of candidate patches. This stage is built on the LLM-as-a-Judge philosophy, in which multiple LLMs collaboratively evaluate candidate patches. The ACoder Agent is reused as a judge, and is provided with both the voting task and the complete codebase. This ensures the judge operates with full capabilities and sufficient context, leading to more accurate selection outcomes.

The selection stage is designed for robustness and scalability:

- **Criteria-Based Voting** — The voting task prompt contains the original task, the candidate patches, and a set of evaluation criteria. All LLM judges evaluate using the same set of criteria, ensuring consistency across evaluations. We iteratively refined these criteria with LLM assistance, analyzing the common traits of patches that solve the problem versus those that fail to do so, such as:
  - **Correctness** — Does the patch fully solve the problem described in the task?
  - **Safety** — Does the patch introduce any new bugs, regressions, or side effects?
  - **Simplicity** — Is the solution unnecessarily complex, or is it direct and maintainable?
- **Multi-Judge Panel** — A panel of diverse, leading LLMs serves as judges (Claude 4.0 Sonnet, Claude 4.1 Opus, GPT-5, and Gemini 2.5 Pro 0617). This mitigates the bias of any single model. Each judge receives the same criteria-based prompt and independently casts a binary vote (0 or 1) for each candidate patch.
- **The Tally and Iterative Decision** — Votes are aggregated across the panel. If a single patch receives the highest vote count, it is selected as the final patch. In the event of a tie, the system triggers an additional round of analysis focusing on the tied patches.

Note: In the Selection stage (and, of course, in the Generation stage), the system does not use any hints, SWE-bench evaluation, PASS_TO_PASS / FAIL_TO_PASS tests, or web-browsing capabilities to look up solutions. We strictly adhere to the SWE-bench Verified evaluation criteria.

4. Future Directions

We identify some directions for future improvement:

• **Selection Accuracy** — Introduce an adaptive mechanism to evaluate each LLM judge's reliability across multiple Selection rounds, removing those with the largest deviation from the actual outcome. Pairing this pruning process with the introduction of newly qualified models can maintain diversity and enhance overall accuracy.

• **Refinement** — The current process flow is linear, with the Generation stage not leveraging feedback from the Selection stage. Adding a feedback loop would allow the Agent to refine its outputs based on the selector's evaluation, improving end-to-end performance.

5. Contributors

This work was done by ACoder Team.

Contributors：Tengjixiang Teng, Xinhe Zhang, Jiawei He, Changkong Zhou, Zhidong Qiao, Xi Zhang, Tao Zou