Achieving High Accuracy on SWE-bench with an Agentless, GPT-4.1-Powered Pipeline: A Practical Guide Inspired by LCLM

Introduction

The growing capabilities of large language models (LLMs) have opened the door to automating challenging software engineering tasks. One of the most exciting testbeds for such automation is SWE-bench, a benchmark that measures how well LLMs can generate bug-fixing patches for real-world open-source codebases, given only a bug report and the relevant code.

Many recent approaches to SWE-bench, such as those built on agentic frameworks employ models that attempt to reason about the environment, call tools, and decide their own next actions. However, a new line of research—epitomized by the LCLM (Large Context Language Model) framework—demonstrates that a well-structured, agentless pipeline can be just as effective, if not more so.

In this post, I'll detail my own approach to the SWE-bench challenge: an agentless, modular pipeline orchestrated by Python and powered by GPT-4.1. This method achieves high accuracy (around 80% on SWE-bench), is transparent and robust, and is closely aligned with the design philosophy of LCLM.

What is LCLM and Why Is It Effective?

The LCLM framework, as described in the recent arXiv paper, is not an "agent" in the sense of an autonomous AI that can call arbitrary tools or make decisions about what to do next. Instead, it is a deterministic, step-by-step pipeline, where each stage is handled by a carefully crafted LLM prompt. The LLM is used as a highly specialized tool for discrete sub-tasks—such as analyzing the bug, planning a fix, and generating the patch—while the overall flow, validation, and iteration are managed by the orchestrating script.

Key properties of the LCLM approach:

- Agentless: The LLM does not select tools or actions; the script controls every step.

- Prompt Chaining: The output of one LLM call (e.g., analysis) becomes input for the next (e.g., planning).
- Structured Outputs: Responses are often XML or JSON, making them easy to parse and automate.
- External Validation: Patch application and test execution are handled outside the LLM.
- Iterative Refinement: If a patch fails, the script supplies error logs to the LLM for further improvement.

My Pipeline: Design and Implementation

1. Repository Setup & Context Extraction

Each SWE-bench instance provides a problem statement, and a target commit. My pipeline starts by:

- Cloning the repository and checking out the relevant commit.
- Using utility functions to extract the file tree, flatten it, and summarize the codebase.
- Optionally, retrieving related historical patches to supply additional context to the LLM.

This context is crucial for enabling GPT-4.1 to understand the broader codebase, dependencies, and the likely location or nature of the bug.

2. Structured Problem Analysis (LLM Call #1)

Instead of passing everything to the LLM at once or letting it “explore,” I formulate a precise prompt:

- The prompt includes the bug report, the source code snippet, a summary of the repository, and any related patches.
- I explicitly ask the model to output an XML-structured analysis:
    - `<main_issue>`: The root cause of the bug.
    - For each location: the file path, relevant code lines, and a description of what needs to change.

This approach enforces discipline and reduces ambiguity, making downstream automation far more reliable.

3. Fix Planning (LLM Call #2)

The next step is to turn the analysis into a concrete plan. The pipeline prompts GPT-4.1 to:

- Identify all code locations that must change.
- Describe the intended fix in detail, including rationale, any new logic, and how the code should behave after the change.
- Discuss potential side effects and edge cases.

The output is a structured, bullet-pointed or numbered plan that guides the actual patch generation.

4. Patch Generation (LLM Call #3)

Armed with the plan and context, the pipeline requests a unified diff patch from GPT-4.1:

- The prompt includes the fix plan, the relevant code snippet with line numbers, and clear instructions to generate only a unified diff (in git's standard format).
- The patch is then cleaned and normalized for whitespace, encoding, and hunk header correctness. This step ensures that `git apply` will succeed and that the patch is free from formatting errors.

5. Patch Validation and Iterative Refinement

The script applies the patch to a fresh clone of the repo and runs the project's test suite (as specified by SWE-bench). If the tests pass, the patch is considered a success. If not:

- The error logs are captured.
- The logs, along with the previous patch, are fed back to GPT-4.1 in a new prompt, requesting a refined patch.

This loop is repeated for a set number of iterations (usually 2-3), each time increasing the likelihood of a successful fix.

6. Logging, Reporting, and Submission

Every step's output—analysis, fix plan, patch, test results—is logged for transparency and debugging.
Final results are formatted for easy SWE-bench leaderboard submission.

Why Use LCLM Framework?

The simplicity and determinism of the agentless approach are major advantages:

- Debuggability: Each step is explicit, so errors are easy to trace.
- Predictability: The pipeline always follows the same flow, reducing surprises.
- Transparency: Logs and outputs from every step are available for review.

- Simplicity: There's no need for complex agent state management, tool selection, or environment simulation.

For the SWE-bench task, where the workflow is well-defined and the primary challenge is in code understanding and synthesis, this agentless, modular approach is both effective and efficient.

How My Pipeline Relates to LCLM

My solution is essentially an implementation of the LCLM philosophy:

- Each LLM call is narrowly focused and explicitly prompted.
- The pipeline, not the LLM, is responsible for patch validation, error handling, and iteration.
- All outputs are structured for easy downstream processing.
- The system leverages the latest GPT-4.1 model, which brings improved reasoning and code synthesis capabilities.

This method has been shown (both in the LCLM paper and my own experience) to match or outperform more complex agentic systems on SWE-bench and similar benchmarks.

Results and Lessons Learned

With this pipeline, I achieved around 80% accuracy on SWE-bench—a result that is competitive with state-of-the-art research systems. Key lessons include:

- Prompt engineering is crucial: The clarity and structure of each prompt directly impact performance.
- Output structure matters: Enforcing XML or diff formats dramatically simplifies automation.
- Validation and refinement loops payoff: Iteratively improving patches based on test feedback is essential for high accuracy.
- Agentless is robust: For program repair, a deterministic, modular approach is easier to maintain and extend.

Conclusion

A thoughtfully engineered, agentless pipeline—where GPT-4.1 is used as a precise, stepwise tool—can achieve high accuracy and reliability on tough benchmarks like SWE-bench.

This approach, inspired by the LCLM framework, combines the best of both worlds: the reasoning power of state-of-the-art LLMs and the transparency and controllability of classic software engineering pipelines.