Code Researcher: Deep Research Agent for Large Systems Code and Commit History

Ramneet Singh* **Sathvik Joel*** **Abhav Mehrotra** **Nalin Wadhwa**
Ramakrishna B Bairi **Aditya Kanade** **Nagarajan Natarajan**
Microsoft Research
{ramneet2001,ksjoe30,abhavm1,nalin.wadhwa02}@gmail.com
{ram.bairi,kanadeaditya,nagarajan.natarajan}@microsoft.com

Midterm Project : Paper Reading

Team 10

吳亞宸 林席葦 鍾鎮嶸

Today's Agenda

- Research Question & Motivation
- Related Work
- Method
- Experiment Setup and Results
- Conclusion

The Blind Spot of SOTA AI Agents

- Current State: LLM-based coding agents show impressive results on standard coding benchmarks (e.g., SWE-bench).
- Problem: Their effectiveness on **complex systems code** (like OS kernels, networking stacks) is a major, underexplored challenge.
- Challenge: Making changes to these codebases is a daunting task.

Motivation (1): Why is "Systems Code" So Hard?

1. **Size and Complexity:**

   They are very large (Linux Kernel: 28M lines of code) and feature "global interactions" (concurrency, memory management).

2. **Low-Level & Critical:**

   Complex low-level code that interfaces directly with hardware.

3. **Rich Development History:**

   Possess rich development histories spanning decades.
   Fixing bugs requires referencing this "massive commit history" for context.

Motivation (2): The Gap - Why SOTA Agents Fail

Different Scenarios:

- SOTA (e.g., SWE-agent): Trained on "issue descriptions written by humans," which provide natural language hints and identify "likely relevant" files.
- Problem (Systems Code): The input is a **crash report**. It is noisy, ambiguous, and devoid of natural language hints.

The Methodological Gap:

- Existing agents do not expend much efforts in gathering codebase-wide context.
- Our problem requires an agent to "research many pieces of context" before attempting a fix.

Research Question: From "Coding" to "Deep Research"

- Core Argument: Fixing systems code bugs demands multi-step reasoning and context gathering.
- Goal of this Research:
  - Design the **first deep research agent for complex bug resolution.**
  - Apply it to the problem of generating patches for mitigating crashes reported in systems code.
- Key Capability: The agent must perform "multi-step reasoning about semantics, patterns, and commit history of code".

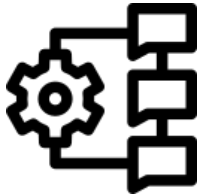
Today's Agenda

- Research Question & Motivation
- Related Work
- Method
- Experiment Setup and Results
- Conclusion

Coding Agents

- Definition: Use a "single ReAct-style loop", such as SWE-agent and OpenHands.
- Limitations:
  - No Deep Context: not designed to reason about complex interactions and gather context. They explore very few files (e.g., SWE-agent: 1.9 files).
  - No History: "Code Researcher is also the first agent to incorporate causal analysis over **historical commits**". This is a key differentiator.

Deep Research Agents



- Definition: A fast emerging subfield in agentic AI designed to tackle "complex, knowledge-intensive tasks".
- Typical Applications: Long-form document generation, complex Q&A over web content.
- Contribution: The first to design & evaluate a **deep research strategy** for **complex bug resolution in large codebases**.

Comparison

Q: Why not just use "Long Context" Models?

- Scale: The code is too big. The Linux kernel far exceeds SOTA model context windows (approx. 100K lines).
- Performance: Long-context LLMs struggle with long sequences and reasoning over much info. They get "lost in the middle".

Q: Why not use other "Kernel Fixers" (e.g., CrashFixer)?

- Assisted vs. Unassisted:
  - CrashFixer: It is *told* which files are buggy.
  - Code Researcher: Identify the buggy files by itself.

Today's Agenda

- Research Question & Motivation
- Related Work
- Method
- Experiment Setup and Results
- Conclusion

Method: overview

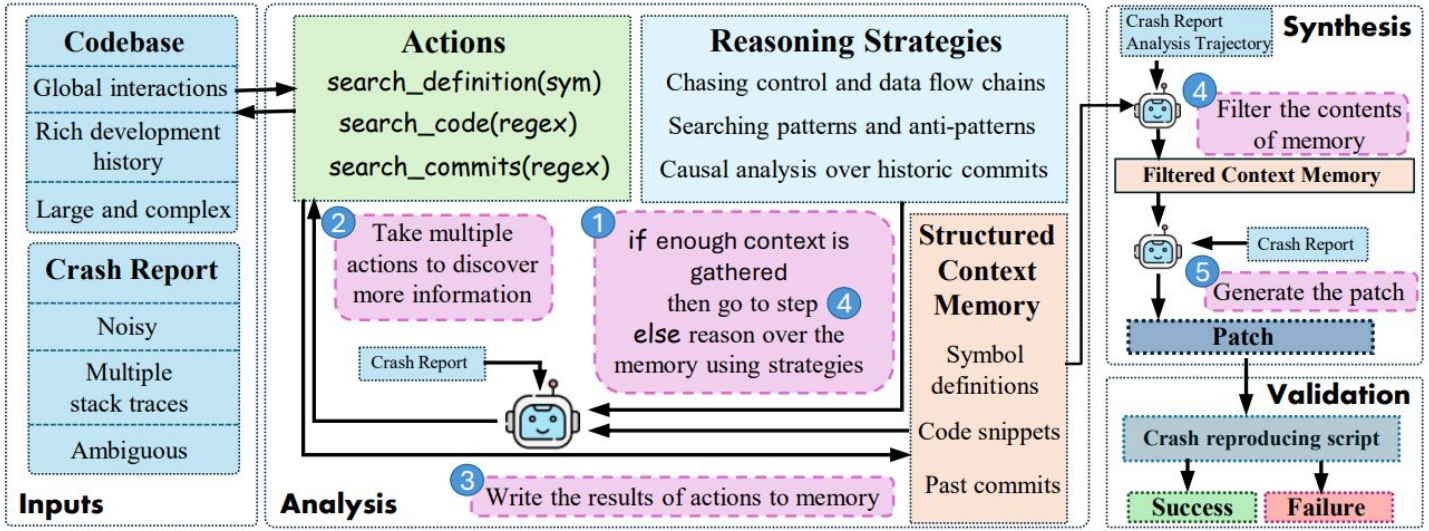
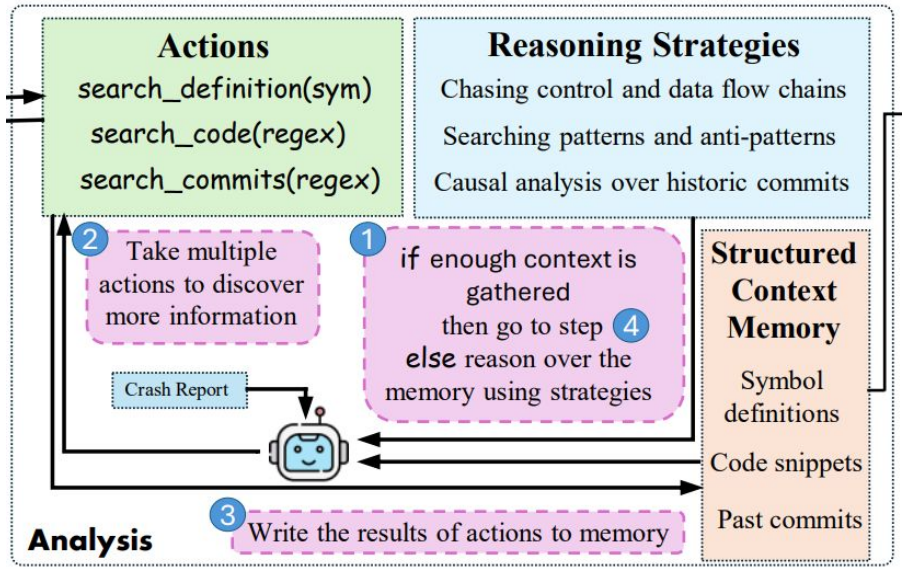


Figure 1: *Code Researcher* conducts deep research over code in three phases: (1) Starting with the codebase and crash report as input, the Analysis phase performs multi-step reasoning about semantics, patterns, and commit history of code. It gathers context in a structured memory. (2) The Synthesis phase filters the contents of the memory to keep relevant context and generates a patch. (3) The Validation phase uses external tools to validate the patch.

Method: analysis phase

- An iterative loop responsible for understanding the why behind the crash.
- Three key components:
  - Actions, Reasoning Strategies, Structured Context Memory.



Method: analysis phase

The agent can invoke several powerful tools to search the environment:

- `search_definition(sym)`: Finds definitions of functions, structs, macros, etc.
- `search_code(regex)`: Searches the entire codebase for specific code patterns.
- `search_commits(regex)`: Key Differentiator: Searches historical commit messages and code diffs for patterns.
- `done`: Action to end the analysis and move to the synthesis phase.

Method: analysis phase

The agent is prompted to use specific reasoning strategies to guide its tool usage.

1. Chasing Control and Data Flow Chains
    a. Follows function calls (control flow) and variable usage (data flow) to understand how execution leads to the crash.
    b. Leads to actions like: `search_definition(function_name)`
2. Searching for Patterns and Anti-Patterns
    a. Finds "normal" code patterns (patterns) to identify "deviant" buggy code (anti-patterns).
    b. If a null check is missing, it searches if other parts of the code do perform that check.
    c. Leads to actions like: `search_code("if (ptr == NULL)")`

Method: analysis phase

3. Causal Analysis Over Historical Commits
    a. Bugs are often related to code evolution and past changes.
    b. Leads to actions like: search_commits("fix memory leak")
4. Iterative Process of Deep Research
    a. starts with the crash report and an empty memory.
    b. it evaluates all context gathered so far and decides which strategy to use next.
    c. It continues until it decides it has sufficient context and calls `done`.

Method: analysis phase

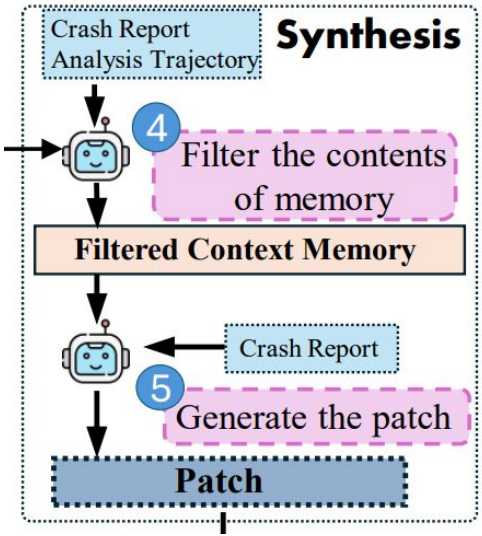
Structured Context Memory:

- A simple, structured list of (action, result) pairs for every reasoning step.
- This memory is reviewed by the agent at every reasoning step.
- It forms the complete "research notes" that are passed to the Synthesis phase.

Method: synthesis phase

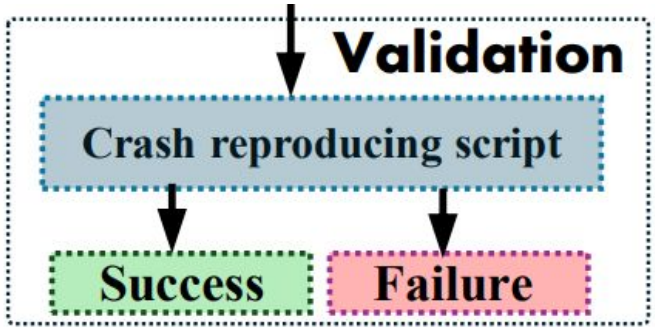
This phase receives the full Context Memory and Reasoning Trace from the Analysis phase.

1. Filter: The agent first discards all (action, result) pairs that are deemed irrelevant to the task of fixing the crash.
2. Hypothesize & Generate: The agent uses the remaining, filtered information to:
    a. Generate a hypothesis about the bug's root cause.
    b. Generate the corresponding code patch (potentially across multiple files).



Method: validation phase

1. The patch is applied to the codebase.
2. The codebase (e.g., Linux kernel) is compiled.
3. The user-space program that originally caused the crash is run.
4. If the crash is not reproduced, the patch is accepted. Otherwise, rejected.



Method: key differences

- Research, Don't Just ReAct
- History is Key
- Systematic & Iterative
- Filter & Validate

Method: examples

```
smsusb:smsusb_probe: board id=7, interface number 0
------------[ cut here ]------------
WARNING:
CPU: 0 PID: 9 at kernel/workqueue.c:3182 __flush_work+0x95c/0xbf0 kernel/workqueue.c:3182
Call Trace:
 <TASK>
 __cancel_work_timer+0x3e6/0x580 kernel/workqueue.c:3276
 smsusb_stop_streaming drivers/media/usb/siano/smsusb.c:182 [inline]
 smsusb_term_device+0xef/0x300 drivers/media/usb/siano/smsusb.c:344
 smsusb_init_device+0xb60/0xd10 drivers/media/usb/siano/smsusb.c:419
 smsusb_probe+0x5b1/0x10c0 drivers/media/usb/siano/smsusb.c:567
 usb_probe_interface+0x307/0x930 drivers/usb/core/driver.c:396
...
worker_thread+0x687/0x1110 kernel/workqueue.c:2552
kthread+0x33a/0x430 kernel/kthread.c:379
ret_from_fork+0x1f/0x30 arch/x86/entry/entry_64.S:308
 </TASK>
```

Figure 3: Crash report of the kernel crash example discussed in Appendix A.

You are analyzing a kernel crash report. Your task is to explore the repository using available tools and gather enough context to completely understand all possible root causes of the crash and how to fix it.

STEP 0 of 9

BUG TITLE : WARNING in smsusb_term_device

CRASH REPORT:
{crash_report}

AVAILABLE ACTIONS (write one action per line with no formatting inside <actions> tags):
<actions>
close_definition(filePath, symbolName, startLine)
Close a definition that is not relevant to the bug (e.g. debug functions like dump_stack)
search_definition(filePath, symbolName)
Open the definition of a specified symbol, which can be a function name, a struct name, a constant, a union or a macro
search_definition(symbolName)
Open the definition of a symbol when you don't know its file path. Use this to search for the definition of symbols whose file you don't know.
search_code("regex pattern")
Search through all tracked files in the repository using git grep with regex pattern. Shows 2 context lines before and after each match, limited to 5 matches total.
done End exploration because you have gathered enough context to understand the bug and propose a fix. You must only take this action if you are satisfied with the justification you provided in the thoughts section. If unsure, continue exploring. You should also write your justification after this action inside <justification></justification> tags. (followed by justification in <justification> tags)
</actions>
You should examine the evidence and write your thoughts about the current state… Focus on gathering all the relevant context and understanding the bug step-by-step, through careful reasoning steps.

← Prompt for LLM in analysis phase

↓ Reasoning and Actions.

The crash report indicates a warning in the `smsusb_term_device` function. The warning is triggered during a call to `__flush_work` in `kernel/workqueue.c`.
…
The issue appears to be related to the handling of work items, possibly involving improper synchronization or misuse of workqueues. To understand the root cause, I need to examine the `smsusb_term_device` function and the `smsusb_stop_streaming` function to see how work items are managed and flushed. Additionally, I should explore how `__flush_work` is used and what could lead to the warning in this context.

<actions>
search_definition("drivers/media/usb/siano/smsusb.c", "smsusb_term_device")
search_definition("drivers/media/usb/siano/smsusb.c", "smsusb_stop_streaming")
search_code("__flush_work")
</actions>

The results of actions are added to memory

The contents from the memory are added to prompt

STEP 1 of 9

PREVIOUS SEARCHES AND RESULTS:
Query: search_code("__flush_work")
Results:

```
include/linux/workqueue.h=447=extern bool queue_rcu_work(struct workqueue_struct *wq, struct rcu_work *rwork);
include/linux/workqueue.h-448-
include/linux/workqueue.h:449:extern void __flush_workqueue(struct workqueue_struct *wq);
include/linux/workqueue.h-450-extern void drain_workqueue(struct workqueue_struct *wq);
include/linux/workqueue.h-451-


===

include/linux/workqueue.h=575=extern void __warn_flushing_systemwide_wq(void)
--
include/linux/workqueue.h-615-  if (0)
include/linux/workqueue.h-616-                  __warn_flushing_systemwide_wq();                \
include/linux/workqueue.h:617:  __flush_workqueue(system_wq);                                   \
include/linux/workqueue.h-618-})
include/linux/workqueue.h-619-
```

(... more results of this query are not shown in this figure due to space constraints)

CURRENTLY OPEN DEFINITIONS:
File: drivers/media/usb/siano/smsusb.c
Symbol: smsusb_term_device Lines 337 to 355

```c
337| static void smsusb_term_device(...)
338|{
... ( we show the complete definition in the actual prompt )
355| }
```

File: drivers/media/usb/siano/smsusb.c
Symbol: smsusb_stop_streaming
Lines 176 to 189

```c
176| static void smsusb_stop_streaming(...)
177|{
... ( we show the complete definition in the actual prompt )
189| }
```

... skipped step 2~8

The `smsusb_onresponse` function is called as a callback for URB completion, and it initializes the work structure with `INIT_WORK(&surb->wq, do_submit_urb)`. ... However, if `smsusb_stop_streaming` is called before any URB completes, the work items might be canceled without being initialized, leading to the warning observed in `__flush_work`.
To fix the bug, I need to ensure that work items are only canceled after they have been initialized. This might involve adding checks to ensure URB completion has occurred or modifying the logic to handle uninitialized work items gracefully during cancellation.

I believe I have gathered enough context to understand the bug and propose a fix. The sequence of initialization and cancellation of work items is clear, and the potential for uninitialized work items leading to warnings is evident.

<actions>
done
</actions>

The results of actions are added to memory

Today's Agenda

- Research Question & Motivation
- Related Work
- Method
- Experiment Setup and Results
- Conclusion

Benchmarks

- kBenchSyz
  - A benchmark of 200 validated Linux kernel crashes.

Evaluation metrics

- Pass@k (P@k):
  - Is **at least one** of the k generated patches successful? 0 = failed, 1 = successful.
- Crash Resolution Rate (CRR):
  - The primary metric.
  - The **percentage of bugs** the agent **successfully** generate a patch for.
- Recall:
  - The number of **modified ground-truth** files / number of **all** modified files.
- All, Any, None:
  - the **percentage** of candidate patches where All, Any or None of the **ground-truth** buggy files are edited.

Table 1: Crash resolution rate (CRR) for different tools on the kBenchSyz benchmark (200 bugs). LLMs used by the agentic tools are in parentheses. *CrashFixer numbers are from [24], out of 279 bugs; results wrt to the 200 bugs (Section 4) will be updated when available.

Setting	Tool	Max calls	P@k	CRR (%)
Assisted	GPT-4o	1	P@5	36.00
	o1	1	P@5	**51.00**
	CrashFixer (Gemini 1.5 Pro-002)*	≥ 4	P@16	49.22*
Stack context	GPT-4o	1	P@5	29.50
	o1	1	P@5	**40.00**
Unassisted	SWE-agent (GPT-4o)	15	P@5	31.50
	***Code Researcher* (GPT-4o)**	15	P@5	48.00
	***Code Researcher* (GPT-4o + o1)**	15	P@5	**58.00**
Unassisted + Scaled	SWE-agent (GPT-4o)	30	P@5	32.00
	***Code Researcher* (GPT-4o)**	30	P@5	47.50
	SWE-agent (GPT-4o)	15	P@10	37.50
	***Code Researcher* (GPT-4o)**	15	P@10	**54.00**

baseline →

- **Assisted:** tool is told which files contain the bugs (ground-truth files that are part of the fix commits)
- **Stack context:** tool is given the files mentioned in the crash report
- **Unassisted:** the tool is given only the codebase and the crash report, with no hints.

How well do the files edited by the tools **match those modified in developer fixes**?

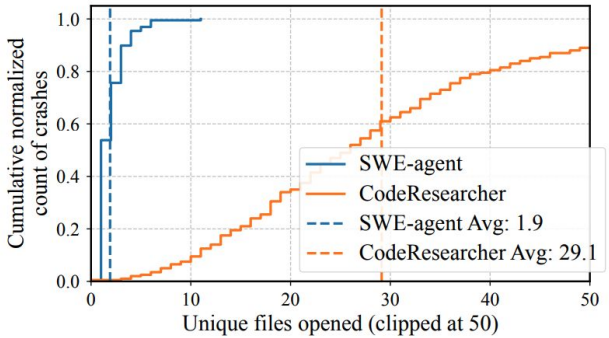
Table 2: Average recall and All/Any/None percentages (metrics defined in Section 4) for the two agentic tools. LLMs used by the tools are in parentheses.

Setting	Tool	Max calls	P@k	Avg. Recall	All/Any/None (%)
Stack context	GPT-4o	1	P@5	**0.45**	**42.5**/7.8/49.7
	o1	1	P@5	0.42	39.7/7.7/52.6
Unassisted	SWE-agent (GPT-4o)	15	P@5	0.37	35.1/5.6/59.3
	***Code Researcher* (GPT-4o)**	15	P@5	0.51	48.2/7.8/44.0
	***Code Researcher* (GPT-4o + o1)**	15	P@5	**0.52**	**50.0**/7.6/42.4
Unassisted + Scaled	SWE-agent (GPT-4o)	30	P@5	0.40	37.9/6.4/55.7
	***Code Researcher* (GPT-4o)**	30	P@5	**0.52**	**49.5**/8.0/42.5
	SWE-agent (GPT-4o)	15	P@10	0.36	34.3/5.4/60.2
	***Code Researcher* (GPT-4o)**	15	P@10	0.51	48.4/7.8/43.8

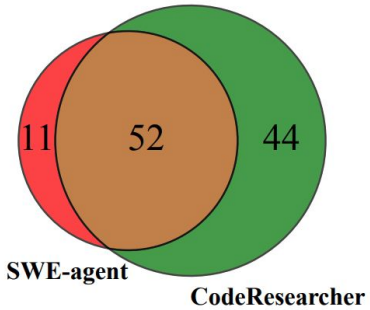
- **Stack context:** tool is given the files mentioned in the crash report
- **Unassisted:** the tool is given only the codebase and the crash report, with no hints.

How effective is context gathering for resolving Linux kernel crashes?

- Code Researcher gathers much more context than SWE-agent
- Code Researcher has better overlap with developer-referenced context
- Code Researcher has a significantly higher CRR than SWE-agent when both the tools correctly identify all the ground-truth modified files:



(a) Files explored for each crash (summed over 5 trajectories).



(b) Overlap of crashes resolved.

Figure 2: SWE-agent vs *Code Researcher* (both at GPT-4o, P@5, 15 max calls)

How important are historical commits for resolving crashes in the Linux kernel?

- Code Researcher is the first agent to explicitly leverage the rich development history of codebases

Table 3: Effectiveness of access to historical commits.

Tool	Max Calls	P@k	CRR(%)	Avg. Recall	All/Any/None(%)
***Code Researcher* (GPT-4o)**	15	P@5	**48.00**	**0.51**	**48.2**/7.8/44.0
w/o `search_commits`[1]	15	P@5	38.00	0.33	32.6/2.4/65.0

Today's Agenda

- Research Question & Motivation
- Related Work
- Method
- Experiment Setup and Results
- Conclusion

Conclusion

- This paper (Code Researcher):
  - Designed a deep research framework in resolving complex issues in large codebase:
    - Analysis phase:
      - Actions:
        - search_definition(sym)
        - search_code(regex)
        - search_commits(regex)
      - Reasoning
      - Store as Structured Context Memory (symbol def, code, past commits)
    - Synthesis phase
    - Validation phase
  - Shows:
    - Better CRR (percentage of bugs solved)
    - More correct fixes
    - Gather more context before starting to patch