Understanding via Reconstruction: Reversing the Software Development Process for LLM Pretraining

Anonymous ACL submission

Abstract

While Large Language Models (LLMs) excel at code generation, they struggle with the long-horizon reasoning required for complex software engineering. We argue this stems from pre-training on static repositories, which represent only the terminal state of development and abstract away the underlying intellectual process. To bridge this gap, we propose understanding via reconstruction: a paradigm that reverse-engineers the latent agentic trajectories—planning, reasoning, and debugging steps—behind static code. Using a multi-agent simulation grounded in repository structures (e.g., dependency graphs), we synthesize these trajectories and refine them via a search-based optimization of the Chain-of-Thought (CoT). Our results demonstrate that continuous pre-training on these reconstructed traces significantly enhances Llama-3-8B across long-context understanding, coding, and agentic benchmarks, proving that the generative "process" is a superior supervision signal to the "result" alone.

1 Introduction

The remarkable success of Large Language Models (LLMs) can be viewed as a modern validation of Richard Feynman's famous dictum: "*What I cannot create, I do not understand.*" The dominant paradigm of generative pre-training (Floridi and Chiriatti, 2020; Ouyang et al., 2022) is built on this very principle—that the ability to generate text token-by-token serves as the proxy for understanding language. By learning to predict the next token, models internalize the syntax, semantics, and world knowledge embedded within vast corpora.

However, this "understanding via generation" paradigm faces a fundamental limit when applied to complex, long-horizon artifacts, such as substantial software repositories. A software repository, in its final form, is the terminal state of an intricate intellectual process. It is a highly compressed artifact where the "computational steps" of human reasoning—the requirement analysis, architectural planning, trial-and-error debugging, and iterative refinement—have been abstracted away. When we train models solely on this static code, we are essentially asking them to memorize the destination without showing them the map. Consequently, models often learn to mimic the surface-level structural patterns of the result rather than mastering the generative reasoning required to derive it. This explains why models that excel at generating short snippets often fail to grasp the deep, causal logic required to construct and maintain complex software systems (Pham et al., 2025).

To bridge this gap, we propose that to truly understand a repository, a model should learn to reconstruct the process that created it. Our motivation is to reverse-engineer the latent agentic trajectory hidden behind static code (Wang et al., 2025b). We hypothesize that by restoring the missing details of the generation process—explicitly expanding a static repository into a dynamic sequence of planning, reasoning, and execution steps—we can provide a far richer supervision signal than the raw code alone. This allows the model to learn not just what the code is, but why and how it was written, thereby aligning the training data more effectively with the model's next-token prediction objective.

To implement this data-centric philosophy, we developed a framework to synthesize these trajectories from existing high-quality open-source repositories. We treat the repository as the ground truth answer and simulate the problem-solving steps required to arrive there. Specifically, we employ a multi-agent simulation, where a main agent generates high-level requirements and implementation plans, while sub-agents are delegated to handle individual files. These agents utilize a "Read" tool to gather context and a "Write" tool to generate code. Crucially, to prevent the simulation from drifting, we inject structural ground-truth information—such as file hierarchies and dependency graphs extracted from the repository—to guide the agents, ensuring the synthesized trajectory faithfully reconstructs the target artifact.

While this reconstruction provides the "missing steps", the quality of the reasoning itself remains a variable. The initial CoT generated during simulation may be suboptimal. To address this, we introduce a search-based optimization technique to refine the thinking process. We posit that a high-quality thought (z) should maximize the likelihood of the correct code (x), formalized as maximizing $\log p(x|z)$. Drawing inspiration from tree-search algorithms (Qiu et al., 2024; Wang et al., 2025b), we decompose the trajectory into steps and iteratively sample refinements. We replace the original reasoning with refined thoughts only when they lower the perplexity of the target ground-truth code. This process polishes the synthetic trajectory, yielding a dataset that is not only causally complete but also logically rigorous.

We empirically validate our paradigm by continuously pre-training Llama-3-8B on our synthesized dataset. The results demonstrate that learning from these reconstructed trajectories leads to significant performance gains across diverse benchmarks, including long-context understanding, coding, reasoning, and agentic capabilities.

Our contributions are summarized as follows:

1. We propose a novel paradigm for scaling LLM capabilities based on the principle of **understanding via reconstruction**. We argue that static repositories miss crucial generative details, and we introduce a method to reverse-engineer these latent agentic trajectories to provide richer supervision.

2. We devel2op a multi-agent simulation framework that synthesizes these trajectories by grounding the generation process in the structural realities of source repositories, effectively converting static data into dynamic thinking and acting,.

3. Experimental results show that Llama-3-8B, when pre-trained on our reconstructed data, achieves superior performance across benchmarks for long-context understanding, coding, reasoning, and agentic tasks.

2 Approach

Our goal is to create a high-quality, structured dataset of agentic trajectories from existing code repositories for LLM pretraining. Our method consists of two main stages: (1) Multi-Agent Trajectory Curation, where we simulate a developer workflow to reverse-engineer an agentic trajectory from a complete repository, and (2) LongCoT Optimization, where we refine the reasoning within these trajectories using a search-based algorithm. Figure 1 provides an overview of our entire pipeline.

The primary objective of this framework is not agent training, but the curation of a high-fidelity pre-training corpus. By simulating the development process, we provide the model with a reasoning-dense supervision signal characterized by long-horizon context dependencies that go far beyond surface-level code patterns.

2.1 Multi-Agent Trajectory Curation

We design a multi-agent workflow that mirrors a human software development process. Table 1 shows an illustrative example of such a synthesized trajectory. Instead of building a live agent framework, we prompt a powerful LLM to simulate the entire workflow and generating the corresponding trajectory data. The simulation unfolds as follows:

Main Agent: Project Planning The process begins with a Main Agent. Its responsibilities are high-level planning and coordination. Given the entire code repository as context, the Main Agent is prompted to:

- Generate Project Requirements: Synthesize a high-level description of the project's purpose and functionality, as if it were a task brief.

- Formulate an Implementation Plan: Decompose the project into a logical sequence of file creation steps. This plan outlines which files should be created and in what order, establishing a dependency-aware development path.

For each file in the implementation plan, the Main Agent then invokes a specialized Sub-Agent to handle the implementation.

Sub-Agent: File Implementation A Sub-Agent is responsible for generating the code for a single file. This process is also broken down into thought and action steps:

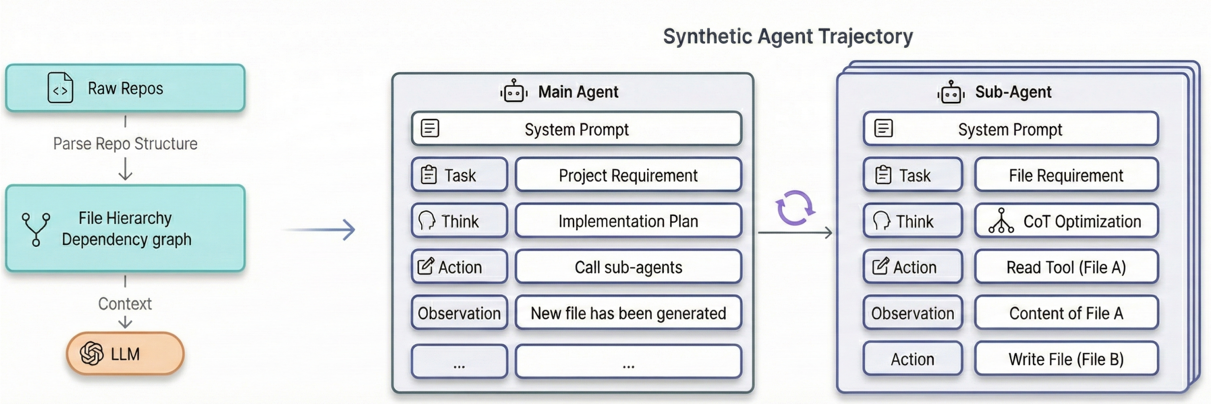


Figure 1: The pipeline of synthetic agent trajectory curation.

- Plan File Implementation: The Sub-Agent first outlines a plan for the specific file's structure and logic.

- Information Gathering (`Read` Tool): Before writing code, the Sub-Agent may need to understand the context of other parts of the repository. It simulates this by calling a `Read` tool to access the content of other, already "implemented" files.

- Code Generation (`Write` Tool): Finally, the Sub-Agent calls a `Write` tool, providing the full code content for the current file.

This entire sequence of thoughts, tool calls (`Read`, `Write`), and tool responses constitutes a single, coherent agentic trajectory (see Steps 8-13 in Table 1 for a detailed instance).

Grounding the Simulation with Extracted Information A purely LLM-simulated trajectory is prone to noise and hallucinations. To enhance the fidelity and accuracy of our synthetic data, we ground the simulation by injecting ground-truth information extracted directly from the source repository. This serves two purposes: guiding the LLM's generation and replacing noisy outputs with factual data.

We extract the following ground-truth information:

- File Structure Tree: A complete directory and file layout of the repository. This is provided to the LLM to simulate the implementation plan of Main Agent.

- Inter-File Dependency Graph: We analyze import statements to build a graph representing how files depend on one another. This is important for the LLM to simulate the tool call and tool response of `Read` Tool.

- Intra-File Structure: For each file, we parse its Abstract Syntax Tree (AST) to extract key structural elements like class and function definitions. This information is provided to the LLM to simulate the Sub-Agent trajectory.

Furthermore, we use this ground-truth data to correct parts of the simulated trajectory. For example:

- The response to a `Read` tool call is replaced with the actual content of the file from the repository.

- The final output of the `Write` tool call is replaced with the ground-truth code of the file.

This grounding process ensures that while the reasoning is generated by the LLM, the actions and outcomes are anchored to reality.

2.2 CoT Optimization via Search

The initial trajectory curation stage leverages an LLM's ability to simulate agentic behavior. However, the generated CoT reasoning (z) may not be optimal for generating the target code (x). An ideal thought process should make the subsequent code generation step as simple as possible. Formally, we aim to find a reasoning path z^* that maximizes the conditional log-probability of the code:

$$z^* = \arg\max_z \log p(x|z)$$

While this objective could be optimized using RL (with $\log p(x|z)$ as the reward), RL training is often complex, expensive, and unstable. We therefore opt for a simpler yet effective inference-time search strategy.

Following Wang et al. (2025b), we decompose the CoT into steps $(z_1, \ldots, z_n)$ and optimize each z_i:

Step	Role	Component	Content
0	User	**Task**	Create a simple calculator project in Python. It should include a file for arithmetic operations and a main script to run a demonstration.
1	Main Agent	**Think**	The project requires a calculator. It needs a file for core logic, `operations.py`, and a main file to run it, `main.py`. The latter depends on the former, so I must create `operations.py` first. The plan is: 1. Create `operations.py`. 2. Create `main.py`.
2	Main Agent	**Action**	`CallSubAgent(task="Create ’operations.py’...”)`
...	...	...	...
5	Main Agent	**Observation**	Sub-agent finished task for `operations.py`. The file has been generated.
6	Main Agent	**Think**	Now proceeding to the next step in my plan.
7	Main Agent	**Action**	`CallSubAgent(task="Create ’main.py’ that uses the add function...”)`
8	Sub-Agent	**Think**	I need to create `main.py`. This file must import the `add` function from `operations.py`. To ensure correctness, I should first read the content of `operations.py`.
9	Sub-Agent	**Action**	`ReadTool(file="operations.py”)`
10	Sub-Agent	**Observation**	`def add(a, b):\n    return a + b`
11	Sub-Agent	**Think**	Okay, I have the content of `operations.py`. Now I can write `main.py`. I will import `add`, call it with 2 and 3, and print the output.
12	Sub-Agent	**Action**	`WriteTool(file="main.py”, content="from operations import add...”)`
13	Sub-Agent	**Observation**	Successfully wrote 89 bytes to `main.py`.
14	Main Agent	**Think**	Sub-agent finished task for `main.py`. All files in the plan have been generated. The project is complete.

Table 1: An illustrative example of a synthesized agent trajectory for creating a tiny project. The process starts with a user instruction, followed by the interaction between the Main Agent and Sub-Agents.

1. Sample: We prompt an LLM to generate a set of k alternative “refinements” for the thought step z_i.
2. Evaluate: For each candidate $z_{cand} = (z_1, \dots, z_i', \dots, z_n)$, measure the Perplexity (PPL) of the ground-truth code x: $\text{PPL}(x|z_{cand})$.
3. Update: If the best refinement $z_i'^*$ results in a lower perplexity than the original step z_i, we permanently update the CoT with this new, improved step.

This iterative refinement ensures the reasoning path is causally structured and directly facilitates correct code generation.

2.3 Continue Pretraining on Synthetic Agent Trajectories

We utilize the synthesized agent trajectories for continual pre-training rather than SFT or post-training. This choice is motivated by the inherent nature of our synthetic data. The trajectories inevitably contain noise and biases stemming from the LLM’s potential hallucinations and our agent workflow. Continuous pre-training, which typically involves larger and more diverse datasets than SFT, is inherently more robust to such imperfections.

Trajectory Flattening To prepare the data, we transform the hierarchical multi-agent interaction into a single sequential document. When the Main Agent calls a Sub-Agent, we recursively inject that Sub-Agent’s entire trajectory (thoughts, tool calls, and observations) directly into the call point. This creates a monolithic, chronological sequence that mirrors the complete development lifecycle of the repository, structurally similar to the example shown in Table 1.

Targeted Loss Masking To ensure the model learns the causal link between reasoning and action rather than memorizing feedback, we mask the tokens corresponding to **Observations** (tool responses). The model is thus trained exclusively to predict `Think` and `Action` tokens, forcing it to internalize the logic of the development process.

3 Experiments

3.1 Experiment Setup

Data Generation: We curate approximately 300k GitHub repositories by filtering for size and quality. Using Team (2025), we generate 4B tokens of synthetic agent trajectories. For CoT optimization, we use a branching factor of 2 and iterate the search-and-replace process for 3 rounds.

Training Configuration: We continually pretrain Llama3-8B-Instruct (Dubey et al., 2024) for 20B tokens with a 64k context window, following Gao et al. (2025). To ensure a fair comparison, all models share a 70% general-domain and 30% repository-related data mixture. Within the 30% repository slot, 18% is fixed (Prolong Repos), while the remaining 12% is allocated to our experimental data variants.

Baselines and Model Variants: We compare the official Prolong baseline against three internal variants, differing only in the 12% experimental data slot:

- Raw-Repos: 12% slot filled with raw source code from our 300k repos.
- Repo2Agent: 12% slot filled with unoptimized synthetic trajectories.
- Repo2Agent-Search: 12% slot filled with search-optimized trajectories.

Our primary comparison focuses on *Raw-Repos*, *Repo2Agent*, and *Repo2Agent-Search* to isolate the impact of converting code into agentic trajectories.

Evaluation Benchmarks We assess the models across four key capabilities. The selection of these benchmarks is directly motivated by the long-context, reasoning-intensive, and code-centric nature of our reconstructed data:

- Long-Context Understanding: As our reconstruction unfolds repositories into massive sequential traces, it introduces long-range causal dependencies. we evaluate this via Ruler (Hsieh et al., 2024) and Helmet (Yen et al., 2025) to test information retrieval across extended horizons.
- Coding: Given our code-domain focus, we use LongCodeBench (Rando et al., 2025) and HumanEval (Chen, 2021) to verify if observing the "process" of code creation enables better synthesis than memorizing static files.
- Reasoning: A central feature of our data is the search-optimized CoT. We evaluate the transferability of this structured logic to general domains using BBH (Suzgun et al., 2022), AGIEval (Zhong et al., 2024), GSM-8k (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), and MMLU-Pro (Wang et al., 2024).
- Innate Agentic Aptitude: Although our model is not an autonomous agent, our data encapsulates patterns of planning and tool selection. We use APTBench (Qin et al., 2025), which is specifically designed to assess the foundational agentic capabilities of pre-trained models (without post-training), to measure the inherent potential instilled by our trajectories.

3.2 Main Results

3.2.1 Long-Context Understanding

We evaluate the long-context capabilities of our models using two comprehensive benchmarks: Ruler and Helmet. Across both benchmarks, our primary observation is that training on structured agent trajectories (Repo2Agent variants) consistently yields superior performance compared to training on flattened code (Raw-Repos). This confirms that reconstructing the process of code generation provides a denser, more instructive signal for long-context modeling than static code files alone. Furthermore, our optimized model, *Repo2Agent-Search*, frequently surpasses the strong external baseline (*Prolong*), particularly in tasks requiring complex information retrieval.

We present the average scores of long-context understanding in Table 2. For a more granular breakdown of performance across all sub-tasks in Ruler and Helmet, please refer to Appendix B (Tables 5 and 6).

Performance on Ruler As shown in Table 2, the models trained on our synthetic data (Repo2Agent & Repo2Agent-Search) consistently outperform the internal *Raw-Repo* baseline across all tested context lengths.

At shorter context lengths (16k and 32k), *Repo2Agent* and *Repo2Agent-Search* maintain a

Benchmark	Context	Prolong	Raw-Repo	Repo2Agent	Repo2Agent-Search
Ruler	16,384	83.61	86.90	**87.50**	87.10
	32,768	81.77	83.20	84.00	**84.40**
	65,536	57.10	61.00	58.10	**61.80**
Helmet	16,384	60.17	60.41	61.56	**61.99**
	32,768	61.57	60.98	62.03	**62.65**
	65,536	**58.10**	57.13	57.32	57.84

Table 2: Summary of Long-Context Understanding performance (Average Scores). Detailed sub-task results are provided in Appendix B.

Benchmark	Prolong	Raw-Repos	Repo2Agent	Repo2Agent-Search
AGI-Eval	**36.91**	35.78	36.32	36.85
BBH	66.69	66.27	66.00	**67.03**
GSM-8k	59.67	**61.94**	**61.94**	60.96
MATH	1.64	2.18	3.72	**3.76**
Human-Eval	16.46	34.76	36.59	**37.20**
LongCodeBench-32k	29.38	34.16	34.51	**36.46**
LongCodeBench-64k	30.52	27.37	**31.05**	30.26

Table 3: Results on Reasoning or Coding Benchmarks, including AGI-Eval, BBH, GSM-8k, MATH, Human-Eval and LongCodeBench.

steady lead over raw code pre-training. The advantage of agentic synthetic data becomes most evident at the 64k window size. While the official *Prolong* baseline and the *Raw-Repo* ablation show significant degradation, *Repo2Agent-Search* achieves the highest robustness with an average score of 61.80. This suggests that learning from a structured, step-by-step construction process helps the model maintain information integrity even when the context is heavily populated.

Performance on Helmet The results on the Helmet benchmark further reinforce the superiority of the reconstruction paradigm.

At 16k and 32k context lengths, *Repo2Agent-Search* achieves peak performance, reaching an average of 62.65 at 32k. This represents a significant improvement over the *Raw-Repo* baseline (60.98), indicating that the search-optimized reasoning steps provide a cleaner and more effective supervision signal for long-range retrieval and reasoning than flattened code files. At the maximum length of 64k, while the *Prolong* baseline remains highly competitive, our *Repo2Agent-Search* continues to outperform the primary *Raw-Repo* ablation. This confirms that even when considering the holistic performance across diverse long-context tasks, converting static repositories into dynamic histories is a more potent data strategy than standard code pre-training.

3.2.2 Coding and Reasoning

We further evaluate whether agentic pre-training benefits fundamental coding and general reasoning (Table 3).

Coding Capabilities Our reconstruction paradigm shows a clear advantage in code generation. On HumanEval, *Repo2Agent-Search* scores 37.20, outperforming the *Raw-Repos* baseline (34.76). This confirms that learning the "process" of creation—incorporating planning and refinement—is superior to memorizing static code. This edge extends to long-horizon tasks; *Repo2Agent-Search* leads on LongCodeBench-32k (36.46).

Reasoning Transfer Despite the lack of math-specific tuning, our method induces positive transfer to general reasoning. On MATH, although absolute scores are low across all models—reflecting the inherent limitations of the Llama-3-8B in complex mathematics—*Repo2Agent-Search* still yields the best results. Furthermore, on BBH and AGI-Eval, our models match or slightly exceed the base-

Category	**Sub-task**	**Raw-Repos**	**Repo2Agent**	**Repo2Agent-Search**
DeepResearch	Openend-Citation	11.20	10.94	**11.49**
	Openend-Plan	**16.11**	13.42	10.40
	Openend-Quality	21.99	24.74	**26.20**
	Plan	47.09	**49.54**	47.81
	Summ-Ans	43.12	**45.30**	44.40
	Average	29.21	**30.49**	30.02
Env-Setup	Action	20.39	**22.05**	21.13
	Error	22.45	23.13	**24.49**
	Plan	18.99	17.85	**19.22**
	Average	20.61	21.01	**21.61**
Issue-Fix	Fix-Patch	26.72	**28.02**	25.43
	Locate	**24.03**	23.67	**24.03**
	Plan	37.04	**40.74**	38.68
	Test-Patch	26.60	**27.08**	26.60
	Tool-Call	54.23	**54.69**	54.28
	Average	33.72	**34.84**	33.80
Overall Average		29.02	**30.10**	29.65

Table 4: Results on APTBench (Merged En/Zh Sub-tasks)

lines. These results demonstrate that the structured logic within agentic trajectories provides a higher-quality supervision signal than raw code, enhancing specialized skills without compromising general intelligence.

3.2.3 Agent Capability

We use APTBench to evaluate the foundational agentic potential instilled by our pre-training. By deconstructing complex trajectories into atomic skills (e.g., planning, tool selection, error diagnosis), APTBench measures a model's inherent aptitude without the confounding effects of post-training.

Repo2Agent excels in planning-centric categories like Issue-Fix (34.84%). This suggests that natural, unrefined CoT provides a more generalizable signal for holistic workflows. *Repo2Agent-Search* leads in Env-Setup (21.61%), particularly in the Error diagnosis sub-task (24.49%). This indicates that search-refined reasoning, being logically more rigorous, is more effective for teaching meticulous, low-level implementation and debugging logic.

In summary, pre-training on synthetic trajectories (Repo2Agent) significantly fosters innate agentic capabilities compared to raw code, with the choice of optimization (Search) offering a tunable balance between broad planning and logical precision.

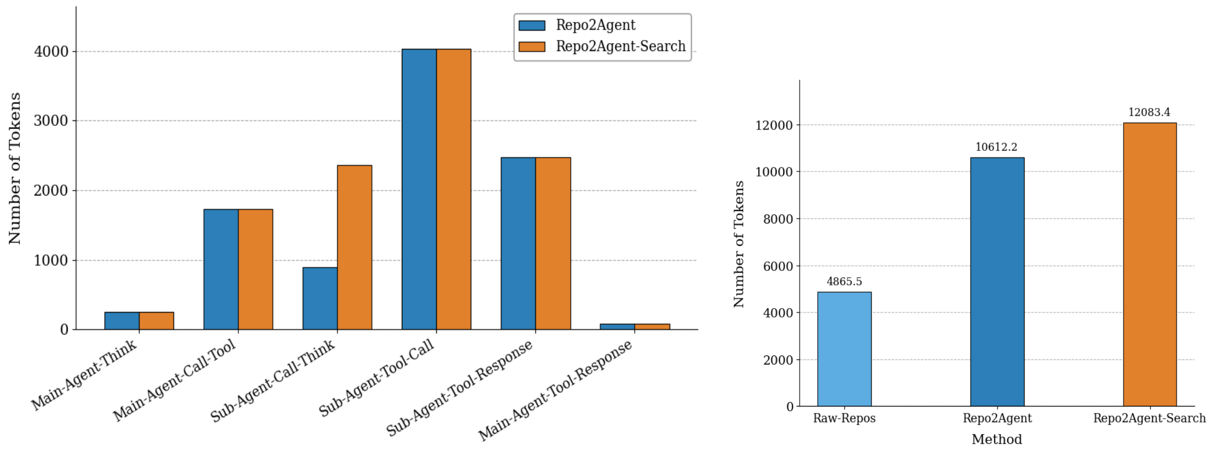
4 Analysis on Synthetic Data

4.1 Token Distribution

We analyze the structural composition and length of our synthetic trajectories to evaluate the impact of agentic reconstruction and search-based optimization (Figure 2).

Composition and Reasoning Expansion As shown in Figure 2a, tokens are primarily concentrated in sub-agent activities (Tool-Calls and Responses), reflecting the detailed implementation process. Crucially, our search optimization significantly deepens the reasoning trace: Sub-Agent-Call-Think tokens more than double from 900 in *Repo2Agent* to 2,300 in *Repo2Agent-Search*. This validates that the search process doesn't merely refine thoughts but substantially elaborates on the logical steps required for implementation.

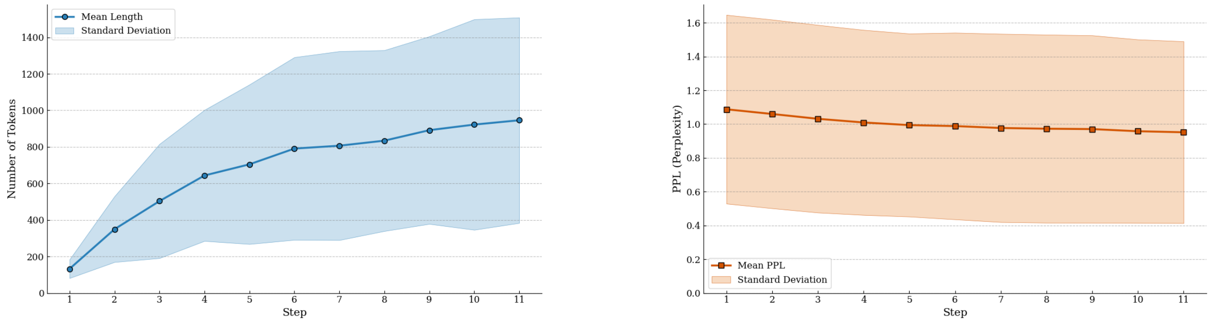
Information Expansion Figure 2b highlights how our paradigm decompresses static code into explicit narratives. Transforming raw code (avg. 4,865.5 tokens) into an agentic trajectory (*Repo2Agent-Search*, avg. 12,083.4 tokens) significantly increases the per-repository token count by making latent planning and execution steps explicit.



(a) Composition of Agent Trajectories

(b) Trajectory Length

Figure 2: (a): Token distribution on thinking, too-call and too-response of main-agent and sub-agent. (b): Average number of tokens for each repo.



(a) CoT Length

(b) PPL of the codes

Figure 3: (a): The CoT increases with more CoT-optimization iterations. (b): PPL of the code to be generated decreases with more iterations.

Importantly, despite the increased sample length, all model variants are trained on a fixed budget of 12% of 20B total tokens. This ensures a fair comparison: the performance gains are driven by the structural quality and informational density of the trajectories, rather than an increase in the total volume of training data.

4.2 Impact of CoT Optimization

We evaluate the relationship between optimization iterations, CoT length, and target code perplexity (PPL) using 100 sample trajectories over 10 iterations (Figure 3).

As shown in Figure 3a, the average CoT length correlates positively with the number of iterations, confirming that our search-based method actively elaborates on the initial reasoning to produce more explicit thought processes. Crucially, this elaboration directly improves reasoning quality: Figure 3b illustrates a steady decrease in code PPL as iterations increase. This inverse relationship supports our hypothesis that more detailed reasoning provides a more informative and predictive context, thereby simplifying the subsequent code generation task.

5 Conclusion

In this work, we addressed the limitations of training on static software artifacts by proposing a novel paradigm of **understanding via reconstruction**. By reverse-engineering latent agentic trajectories through grounded multi-agent simulation and refining the reasoning via search-based optimization, we transformed static repositories into dynamic, causally rich training data. Our experiments with Llama-3-8B demonstrate that learning from these reconstructed processes significantly enhances coding, reasoning, and agentic capabilities.

6 Limitations

While our approach demonstrates significant improvements in long-context coding and reasoning, several limitations remain.

1. More tools or details could be included in the synthetic data, like summarization tool, self-reflection and git commit.
2. The computational cost of search-based CoT optimization is substantial. Scaling this paradigm to millions of repositories would require significant hardware resources.
3. The quality of the synthetic data is inherently constrained by the capabilities of the teacher model used for simulation.

7 Ethical considerations

Large Language Models were utilized throughout the research and preparation of this manuscript. Specifically, AI assistants were employed to support code implementation and debugging of the trajectory curation pipeline. Additionally, LLMs were used for text polishing to improve the clarity and grammatical correctness of the prose. All AI-generated suggestions were critically reviewed and verified by the authors, who maintain full responsibility for the final content and technical accuracy.

References

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, and 1 others. 2025. Scaling agent learning via experience synthesis. *arXiv preprint arXiv:2511.03773*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, and 1 others. 2025. Towards general agentic intelligence via environment scaling. *arXiv preprint arXiv:2509.13311*.

Luciano Floridi and Massimo Chiriatti. 2020. Gpt-3: Its nature, scope, limits, and consequences. *Minds and machines*, 30(4):681–694.

Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. 2025. How to train long-context language models (effectively). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7376–7399.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. 2025. Simulating environments with reasoning models for agent training. *arXiv preprint arXiv:2511.01824*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Vardaan Pahuja, Yadong Lu, Corby Rosset, Boyu Gou, Arindam Mitra, Spencer Whitehead, Yu Su, and Ahmed Hassan. 2025. Explorer: Scaling exploration-driven web trajectory synthesis for multimodal web agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6300–6323.

Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. 2025. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*.

Jiarui Qin, Yunjia Xi, Junjie Huang, Renting Rui, Di Yin, Weiwen Liu, Yong Yu, Weinan Zhang, and Xing Sun. 2025. Aptbench: Benchmarking agentic potential of base llms during pre-training. *arXiv preprint arXiv:2510.24397*.

Jiahao Qiu, Yifu Lu, Yifan Zeng, Jiacheng Guo, Jiayi Geng, Huazheng Wang, Kaixuan Huang, Yue Wu, and Mengdi Wang. 2024. Treebon: Enhancing inference-time alignment with speculative tree-search and best-of-n sampling. *arXiv preprint arXiv:2410.16033*.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. 2025. Longcodebench: Evaluating coding llms at 1m context windows. *arXiv preprint arXiv:2505.07897*.

Saptarshi Sengupta, Harsh Vashistha, Kristal Curtis, Akshay Mallipeddi, Abhinav Mathur, Joseph Ross, and Liang Gou. 2024. Mag-v: A multi-agent framework for synthetic data generation and verification. *arXiv preprint arXiv:2412.04494*.

Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, and 1 others. 2025. Case2code: Scalable synthetic data for code generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 11056–11069.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, , and Jason Wei. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Haozhe Wang, Long Li, Chao Qu, Fengming Zhu, Weidi Xu, Wei Chu, and Fangzhen Lin. 2025a. To code or not to code? adaptive tool integration for math language models via expectation-maximization. *arXiv preprint arXiv:2502.00691*.

Haozhe Wang, Haoran Que, Qixin Xu, Minghao Liu, Wangchunshu Zhou, Jiazhan Feng, Wanjun Zhong, Wei Ye, Tong Yang, Wenhao Huang, and 1 others. 2025b. Reverse-engineered reasoning for open-ended generation. *arXiv preprint arXiv:2509.06160*.

Haozhe Wang, Qixin Xu, Che Liu, Junhong Wu, Fangzhen Lin, and Wenhu Chen. 2025c. Emergent hierarchical reasoning in llms through reinforcement learning. *arXiv preprint arXiv:2509.03646*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 13484–13508.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, and 1 others. 2024. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*.

Yiheng Xu, Dunjie Lu, Zhenan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. 2024. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. *arXiv preprint arXiv:2412.09605*.

Howard Yen, Tianyu Gao, Minmin Hou, Ke Ding, Daniel Fleischer, Peter Izsak, Moshe Wasserblat, and Danqi Chen. 2025. Helmet: How to evaluate long-context models effectively and thoroughly. In *The Thirteenth International Conference on Learning Representations*.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2024. Agieval: A human-centric benchmark for evaluating foundation models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2299–2314.

A Related Work

A.1 Synthetic Agent Trajectories

There are two primary methods for constructing agent trajectories in existing research.

The first method involves generating trajectories through agent exploration in real-world environments (Xu et al., 2024; Pahuja et al., 2025; Team et al., 2025; Fang et al., 2025; Wang et al., 2025a). While this approach ensures the authenticity of the trajectories, it has significant drawbacks, including potentially expensive tool invocation costs and substantial engineering efforts required for environment setup and maintenance.

The second method places the agent in an environment simulated by LLM (Tang et al., 2023; Li et al., 2025; Chen et al., 2025) or prompt LLM to generate an entire synthetic trajectory (Sengupta et al., 2024; Wang et al., 2025b). The main advantage of this approach is its low cost. However, the resulting trajectories may suffer from extensive hallucinations generated by the LLM, compromising data reliability.

The trajectory synthesis method proposed in this paper draws inspiration from the second by using an LLM to generate both the tool calls and their corresponding outcomes. Although the synthesized trajectories may contain some noise, we ensure that their terminal state is a real repository, which serves as the ground truth.

A.2 Synthetic Data for Coding

The use of synthetic data has become a cornerstone in advancing the capabilities of LLMs for code-related tasks. A significant body of work focuses on generating instruction-following datasets. For instance, Magicoder (Wei et al., 2023) synthesizes user instructions for open-source code snippets to create a dataset aimed at enhancing the coding abilities of LLMs. Similarly, Code Alpaca (Chaudhary, 2023) employs the self-instruct methodology (Wang et al., 2023) to generate a dataset of 20,000 code instructions. To improve the quality of these instructions, WizardCoder (Luo et al., 2023) introduces an evolutionary pipeline that progressively increases the complexity and diversity of the initial instructions.

Other research explores different forms of synthetic data. Case2Code (Shao et al., 2025), for example, collects a vast number of input-output test cases by executing existing programs and then generates new programs that satisfy these test cases. More recently, SWE-Synth (Pham et al., 2025) focuses on generating synthetic data for program bug fixing, which has proven effective in improving LLM performance on benchmarks like SWE-Bench (Jimenez et al., 2023). The widespread adoption of such synthetic data in both the pre-training and post-training phases of modern Code LLMs, such as Qwen2 (Hui et al., 2024) and DeepSeek-Coder (Guo et al., 2024), underscores its critical importance (Hui et al., 2024; Wang et al., 2025c).

While our work also contributes to the field of synthetic data for code generation, it diverges from previous efforts in two fundamental aspects. First, instead of augmenting isolated code snippets, we focus on augmenting entire repositories. Second, rather than merely capturing the final code or the associated chain-of-thought, we reconstruct the entire agentic process of developing a repository. This involves synthesizing a sequence of actions, tool interactions, and evolving states, thereby providing a more comprehensive and realistic representation of the software development lifecycle.

Family	Context Length (tokens)	Performance			
		Prolong	Raw-Repo	Repo2Agent	Repo2Agent-Search
NIAH-Multi	16384	99.40	99.50	99.60	**99.70**
	32768	98.40	99.00	**99.20**	**99.20**
	65536	66.20	76.30	68.70	**80.40**
NIAH-Single	16384	100.00	100.00	100.00	100.00
	32768	100.00	99.90	99.90	99.90
	65536	**92.10**	89.90	90.30	91.30
RULER-CWE	16384	80.40	85.00	79.90	**87.30**
	32768	27.60	34.60	33.20	**42.30**
	65536	0.30	**6.50**	0.40	1.60
RULER-FWE	16384	92.70	92.10	**95.10**	92.20
	32768	87.70	88.10	**90.10**	86.00
	65536	73.70	70.30	**79.20**	67.90
RULER-QA	16384	8.90	27.90	**32.90**	28.20
	32768	30.30	33.90	39.00	**39.70**
	65536	20.30	**25.60**	21.50	20.80
RULER-VT	16384	**99.40**	99.00	99.00	98.20
	32768	95.20	**96.00**	95.30	94.10
	65536	**20.50**	14.40	18.60	16.60
Average	16384	83.61	86.90	**87.50**	87.10
	32768	81.77	83.20	84.00	**84.40**
	65536	57.10	61.00	58.10	**61.80**

Table 5: Results on Ruler. We average the results on NIAH-Multi-Key, NIAH-Multi-Value and NIAH-Multi-Query as NIAH-Multi-Multi. The results on RULER-QA-Hotpot and RULER-QA-Squad are averaged as RULER-QA.

B Detailed Results on Long-Context Benchmarks

Ruler Table 5 presents the performance on the Ruler benchmark. The results highlight the robustness of agent-based training data at extreme context lengths.

- Superiority over Raw Code: Our proposed methods consistently outperform the internal *Raw-Repo* baseline. For instance, at the 16k context length, *Repo2Agent* achieves 87.50 compared to 86.90 for *Raw-Repo*. This gap widens in specific tasks; in *RULER-CWE* (32k), *Repo2Agent-Search* scores 42.30, significantly outpacing *Raw-Repo*'s 34.60.
- Robustness at 64k: Performance stability at the maximum window size (64k) is a key differentiator. While the *Prolong* baseline degrades to 57.10 and *Raw-Repo* to 61.00, *Repo2Agent-Search* maintains the highest robustness with an average score of **61.80**.
- Complex Retrieval (NIAH): The benefits of agentic data are most pronounced in the *NIAH-Multi* tasks, which require retrieving multiple pieces of scattered information—a process analogous to an agent locating dependencies across a file system. At 64k tokens, *Repo2Agent-Search* achieves **80.40**, drastically outperforming *Raw-Repo* (76.30) and establishing a massive lead over the *Prolong* baseline (66.20).

Helmet The Helmet benchmark results (Table 6) further validate the efficacy of learning from trajectories, particularly in In-Context Learning (ICL) and Recall tasks.

- Consistent Gains over Raw-Repo: Across all context lengths (16k, 32k, and 64k), *Repo2Agent-Search* consistently achieves a higher average score than the *Raw-Repo* baseline. Notably, at 32k, our search-optimized model reaches **62.65** compared to 60.98 for raw code, demonstrating that the reasoning steps injected during training translate to better general understanding.
- Recall and ICL Capabilities: *Repo2Agent-Search* excels in tasks that mirror the "Recall-Plan-Act" loop of our synthetic agents. In the *Recall* category, it outperforms both *Raw-Repo* and the external *Prolong* baseline at 32k (99.81) and 64k (96.00). Similarly, in *ICL* (In-Context Learning), our method dominates at 16k and 32k, suggesting that observing the step-by-step history of code construction improves the model's ability to learn patterns from context.
- Competitive Performance at Scale: While the *Prolong* baseline shows strength in the *LongQA* subtask at 64k, leading to a slightly higher overall average (58.10), our method remains highly competitive (57.84) and crucially, still outperforms the *Raw-Repo* baseline (57.13). To interpret these results correctly, it is necessary to distinguish the role of the baselines. While *Prolong* serves as an external state-of-the-art reference, *Raw-Repo* is the primary controlled ablation.

In summary, both benchmarks confirm that transforming raw repositories into dynamic agent trajectories is a more effective strategy than standard pre-training on static code, yielding models with sharper retrieval and reasoning capabilities.

C Case Study: Example of synthetic pretraining data

D Case Study: Evolution of CoT

To provide a concrete illustration of how our LongCoT optimization refines the agent's reasoning, we present a case study tracking the evolution of a single Chain-of-Thought (CoT) step through multiple rounds of search. The task is to generate the code for a Python script, `2_Connect_Postgres_DB.py`, which involves database interaction within a Streamlit application.

The Original CoT (Box D) generated by the initial simulation is functional but generic. It outlines a correct but high-level plan, listing seven necessary steps, such as "Checking login and 2FA status" and "Initializing database connection". The reasoning is sparse, providing little detail on the "how" or "why" behind each step.

After the 1st Round of Optimization (Box D) , the CoT becomes more specific and technically precise. Key improvements include:

- Explicit Variable Checks: It explicitly mentions the need to check both `authentication_status` and `status_2FA` in the session state, a detail absent in the original plan.
- Technical Justification: It begins to add rationale, explaining that `@st.cache_resource` is for "one-time initialization" and specifying a "5-minute

		Performance			
Category	Context Length	Prolong	Raw-Repo	Repo2Agent	Repo2Agent-Search
ICL	16384	72.52	68.08	68.88	**73.52**
	32768	75.84	71.84	72.72	**76.32**
	65536	**80.68**	75.92	77.36	78.72
LongQA	16384	28.13	33.44	**36.59**	35.04
	32768	40.40	38.28	**41.55**	40.20
	65536	**46.78**	45.03	44.84	45.48
RAG	16384	64.17	63.88	**64.46**	64.08
	32768	63.33	**63.67**	63.13	63.25
	65536	56.00	57.42	**57.67**	55.58
Recall	16384	99.94	99.94	99.69	99.94
	32768	99.38	98.94	99.19	**99.81**
	65536	95.75	93.31	91.94	**96.00**
Rerank	16384	36.11	36.71	**38.17**	37.35
	32768	28.90	32.19	33.56	**33.69**
	65536	11.31	13.95	**14.77**	13.42
Avg	16384	60.17	60.41	61.56	**61.99**
	32768	61.57	60.98	62.03	**62.65**
	65536	**58.10**	57.13	57.32	57.84

Table 6: Results on Helmet

cache TTL" for `@st.cache_data`. This demonstrates a deeper understanding of the tools being used.

The Final CoT after the 2nd Round (Box D) represents a significant leap in reasoning quality, transforming a simple checklist into a comprehensive implementation blueprint. The enhancements are substantial:

- Detailed Rationale and User Intent: Each step is now accompanied by a rich explanation that links the action to a requirement. For example, it specifies the exact page title to "align with the user's requirement" and explains *why* loading `config.yaml` is "crucial for the authentication flow".
- Elaboration on Edge Cases and Best Practices: The plan now includes explicit error handling logic ("display a warning and prevent further execution") and implementation details ("looping through the fetched rows and formatting the output").
- Holistic Project Awareness: The concluding thought expands beyond the immediate file, mentioning broader concerns like "environment variable loading", "modularity, readability, and follows best practices for Streamlit app development". This indicates a shift from a narrow, file-centric view to a more holistic, project-aware mindset.

This qualitative analysis empirically demonstrates that our search-based optimization does not simply rephrase CoTs. It systematically enriches the reasoning process, making it more detailed, explicit, and context-aware. This enriched reasoning, which more closely mirrors that of an expert developer, provides a much stronger learning signal for the model, which we believe is a key factor behind the performance improvements observed in our experiments.

Original CoT in a Synthetic Agent Trajectory

```
Now I understand that authentication state is managed via st.session_state
by the authenticator. The 'status_2FA' flag is likely set in another
page after 2FA setup, so I will proceed to implement the access control
based on session state as described. I will now generate the full code
for '2_Connect_Postgres_DB.py', including:
1. Page title
2. Loading authentication config from 'config.yaml'
3. Checking login and 2FA status
4. Initializing database connection with @st.cache_resource
5. Query execution with @st.cache_data
6. Displaying results in the required format
7. Error handling for database operations

I will write the code directly since no other function definitions are needed.
```

The CoT after 1st-round Search Optimization

```
Now I understand that the authentication state is managed through `st.session_state`
by the authenticator. The `status_2FA` flag is likely set in a different
page after 2FA setup, so I need to ensure the access control checks both
`authentication_status` and `status_2FA` in the session state. I will
proceed to implement the access control logic as described. Next, I will
generate the full code for `2_Connect_Postgres_DB.py`, making sure to:
1. Set the page title.
2. Load the authentication configuration from `config.yaml`.
3. Check for valid login and 2FA status in the session state.
4. Initialize the PostgreSQL connection using `@st.cache_resource` for
   one-time initialization.
5. Execute a query using `@st.cache_data` with a 5-minute cache TTL.
6. Display the query results in the specified format.
7. Include proper error handling for database connection and query execution.

I will now write the full code, ensuring all requirements are met.
```

The CoT after 2nd-round search optimization

```
Now I understand that the authentication state is managed through `st.session_state`
by the authenticator. The `status_2FA` flag is likely set in a different
page after 2FA setup, so I need to ensure the access control checks both
`authentication_status` and `status_2FA` in the session state. I will
proceed to implement the access control logic as described. Next, I will
generate the full code for `2_Connect_Postgres_DB.py`, making sure to:

1. Set the page title to "Connect Postgres DB" to align with the user's
   requirement.
2. Load the authentication configuration from `config.yaml` as it's used
   to initialize the authenticator, which is crucial for the authentication flow.
3. Check for valid login and 2FA status in the session state. If either is
   missing or not verified, display a warning and prevent further execution.
4. Initialize the PostgreSQL connection using `@st.cache_resource` for
   one-time initialization, ensuring that the connection is not re-established
   on every page rerun.
5. Execute a query using `@st.cache_data` with a 5-minute cache TTL, which
   is specified in the requirement to cache query results.
6. Display the query results in the specified format, looping through the
   fetched rows and formatting the output as `{row[0]} has a :{row[1]}:`.
7. Include proper error handling for database connection and query execution
   to ensure the app handles failures gracefully.

I will now write the full code, ensuring all requirements are met, including
environment variable loading, caching, and user access control. I will also
make sure the code is modular, readable, and follows best practices for
Streamlit app development.
```

E Prompt

Prompt for Generating Main-agent Trajectory

~~~~
A github repo: $repo_code

The tree structure of repo: $file_tree.

Given the repo code and the tree structure of the repo, I want to use it to construct multi-agent
    synthetic data. The main agent needs to generate the implementation plan for the repo based
    on the detailed requirement document of the repo provided by the user, including the tree
    structure of the repo and the implementation order of files. It will also call sub-agents to
    realize the code generation of each file.
‘‘‘
[
    {
        "role": "system_prompt",
        "content": "you are a helpful assistant. ... Show the sub-agent tool usage here.",
    },
    {
        "role": "user",
        "content": "A detailed requirement document for repo, but DO NOT mention implementation
    details of repo",
    },
    {
        "role": "gpt",
        "content": "tree structure of repo, implementation order of repo, call sub-agent to
    generate code for the first file",
        "tool-call": {
            "function_name": "code_generator",
            "arguments": {
                "requirement_for_repo": "requirement for repo",
                "tree_structure": "tree structure of repo",
                "file_name": "first_file.py",
                "file_path": "first_file.py",
                "requirement": "requirement for first_file.py",
            }
        }
    },
    {
        "role": "tool-response",
        "content": "return of function call",
    },
    {
        "role": "gpt",
        "content": "call sub-agent to generate code for the second file",
        "tool-call": {
            ...
        }
    },
    ...
]
The Tool usage which should be put at the system prompt:
    Arguments of sub-agent:
    {requirement for repo, tree structure, file_name, file_path, requirement for file,}
    Return of sub-agent:
    {file_path has been generated successfully}

The memory of main agent should cover the planning of all the files in the repo, and call
    code-generator to generate all these files.
~~~~

Prompt for generating sub-agent trajectory

~~~~
        I have a GitHub repo, and I want to use it to construct multi-agent synthetic data. The
    main agent needs to generate the implementation plan for the repo based on the detailed
    requirement document of the repo provided by the user, including the repo’s tree structure
    and the implementation order of files. It will also call sub-agents to realize the code
~~~~

```
    generation of each file. The sub-agent requires information provided by the main agent,
    including the repo’s requirement document, the repo’s tree structure, the name and path of
    the code file that the sub-agent needs to generate, and the requirement description for this
    code file.

Your task is to generate a JSON list representing the simulated sub-agent’s memory. This memory
    should chronicle the step-by-step thought process of creating a specific file from scratch,
    based on a user’s requirement.

**Crucially, you are simulating the *creation* process, not explaining or refactoring existing
    code.** The agent you are simulating does not have access to the final source code at the
    beginning; it must figure out how to write it.

The format of the memory is as follows:
[
    {
        ”role”: ”system_prompt”,
        ”content”: ”You are ’code_generator’, an expert software engineer. \nYour goal is to
    implement robust, production-ready code from a given requirement.\n\nWorkflows:\n1. ANALYZE
    the file requirement and its place in the repo structure.\n2. IDENTIFY dependencies. If you
    need to use external classes/functions, use the ‘read‘ tool to check their definitions
    first.\n3. PLAN the implementation details (class structure, methods, logic).\n4. WRITE the
    code using the ‘write‘ tool.\n\nTools:\n- read(file_to_read): Returns the
    definition/signature of a file. Usage: When you need to understand how to invoke another
    module.\n- write(file_path, content): Writes the code to the file system.\n-
    final_answer(answer): Reports completion.”,
    },
    {
        ”role”: ”user”,
        ”content”: ”requirement for repo, tree structure, file_name, file_path, requirement for
    file”,
    },
    {
        ”role”: ”gpt”,
        ”content”: ”Here, the agent analyzes the requirement. It decides if external dependencies
    need to be checked based on the specific logic needed. It expresses curiosity or caution
    about specific interfaces it might need to interact with.”,
        ”tool-call”: {
            ”function_name”: ”read”,
            ”arguments”: {
                ”file_to_read”: ”file name”,
            }
        }
    },
    {
        ”role”: ”tool-response”,
        ”content”: ”the content of the file that was read”,
    },
    {
        ”role”: ”gpt”,
        ”content”: ”Here, the agent synthesizes the information from the requirement and any
    dependencies it read. It DOES NOT just list ’Plan: 1, 2, 3.’ Instead, it narrates its
    engineering decisions, mentions specific variable names it *plans* to use, considers edge
    cases for the ‘file_name‘’s logic, and explicitly reasons about how its planned
    implementation will satisfy the requirements.”,
        ”tool-call”: {
            ”function_name”: ”write”,
            ”arguments”: {
                ”file_path”: ”path of file”,
                ”content”: ”The source code that the agent decides to write.”
            }
        }
    }
    ... (rest of the JSON structure)
]

To help you generate this simulated memory, you are provided with the following information. Use
    it as a guide to construct a realistic and accurate thought process.
```

```
*   **Information to construct the user prompt:**
    ‘$arguments_from_main_agent‘
*   **The Golden Source Code for ‘$file_name‘ (The Goal):** This is the target code the simulated
     agent should ultimately produce. **You must not assume the agent has seen this code
     beforehand.** Use it as the "ground truth" to form a plausible thinking path that leads to
     this exact implementation.
    ‘$source_code‘
*   **Source code of related files (Dependencies):** This is the content the agent will see when
     it uses the ‘read‘ tool on other files.
    ‘$related_source_code‘

### CRITICAL INSTRUCTION: THOUGHT PROCESS DIVERSITY
The ‘content‘ fields in the "gpt" turns must contain **highly intelligent, specific, and varied**
     thought processes.
**STRICTLY AVOID** using the same template (e.g., "Okay, I have checked... Plan: 1. 2. 3.") for
     every file.

**Follow these guidelines to generate the thought process:**

1.  **Context-Driven Reasoning**:
    - If the **target** ‘$source_code‘ contains complex algorithms, the simulated thought process
     should focus on algorithmic efficiency and data structures.
    - If the **target** ‘$source_code‘ is a simple DTO or config file, the thought process should
     be brief and focused on correctness.
    - **Mention specific names**: The thought process MUST mention the actual class names,
     variable names, or function names found in the **target** ‘$source_code‘ and
     ‘$related_source_code‘ as part of its reasoning and planning.

2.  **Dependency Logic**:
    - When simulating a ‘read‘ call: Explain *specifically* what the agent is looking for (e.g.,
     "I need to see if the ‘User‘ class has a ‘get_id‘ method or just a public ‘id‘ field before I
     can implement the logic that uses it.").
    - After simulating a ‘read‘: The agent should react to the content found (e.g., "Ah, I see
     ‘User‘’s constructor requires a positional argument, not a keyword argument. I’ll make sure
     to call it correctly in my implementation.").

# Output Format
Return strictly a JSON list representing the memory.
```

Prompt for Optimizing CoT

```
You are an expert software engineer. Your task is to simulate the human reasoning process
     required to solve a programming problem.

**The Goal:**
You need to rewrite a specific part of a reasoning chain (the "Target Block"). The goal is to
     make the reasoning logic more precise, detailed, and aligned with the correct solution,
     WITHOUT breaking the narrative flow.

**Input Data:**
1. **Reference Source Code:** (The correct answer, for your understanding ONLY)
{}

2. **Full Reasoning Context:** (The story so far)
{}

3. **Target Block to Rewrite:** (The weak step needs replacing)
<replace>
{}
</replace>

**CRITICAL INSTRUCTIONS (Read Carefully):**

1.  **The "Time Travel" Rule:**
    You must act as if you are solving this problem *for the first time*. You do NOT know the
     final code yet; you are currently deriving it.
    * **STRICTLY FORBIDDEN:** Do not mention "Reference Code", "Provided Solution", or "Ground
```

```
    Truth".
    * **CORRECT APPROACH:** Instead of saying "The reference code uses a HashMap...", say "I
    think a HashMap would be the best data structure here because..."

2.  **The "Invisible Stitch" Rule:**
    Your output will be copy-pasted directly into the original text to replace the old block. It
    must fit perfectly.
    * **STRICTLY FORBIDDEN:** Do not verify or announce the correction. Never use phrases like
    "In this refinement...", "Correcting the previous step...", "Here is the better
    reasoning...", or "Let’s refine this".
    * **CORRECT APPROACH:** Just write the thought process directly. Start immediately with "I
    need to analyze...", "Next, I will...", etc.

3.  **Tone & Style:**
    * Use **First-Person Singular** ("I check...", "I decide...").
    * Use **Present Tense** (Reasoning happens *now*).
    * Be technical, precise, and deductive.

**Your Workflow:**
1.  **Analyze (‘<think>‘ tags):**
    * Briefly analyze the Reference Code to understand the *correct* logic.
    * Identify why the original ‘Target Block‘ was weak or incorrect.
    * Plan the logic steps needed to bridge the gap.
2.  **Generate (‘<refine>‘ tags):**
    * Write the purely deductive thought process.
    * Ensure it starts and ends in a way that connects with the surrounding text in
    ‘reasoning_chain‘.

Now, generate the replacement block.

<think>
[Your analysis of the gap between the reasoning and the code]
</think>

<refine>
[The seamless, first-person reasoning stream ONLY]
</refine>
```