ByteAsk: Build-Aware Context and a Checking Harness for C++ Coding Agents

Pratyush Saini, CTO, ByteAsk Research pratyush@byteask.ai
Anirudha Kulkarni, CEO, ByteAsk Research anirudha@byteask.ai
Aman Gupta, Carnegie Mellon University amangupt@alumni.cmu.edu

Code and data: github.com/ByteAsk/byteask-system-cpp

Abstract

Large language model (LLM) coding agents have changed how people write software: a developer describes a change in plain language and the agent finds, edits, and tests the code. Most of this progress, however, has concentrated on a few widely used languages, Python above all. C and C++, which still underpin operating systems, browsers, databases, and embedded software, remain far less explored, even though their large, build-dependent codebases are exactly the setting in which agents struggle to find where a change belongs. We propose ByteAsk, a coding agent built for C and C++. At its core is a context engine that understands a codebase the way the project's own build does. Before the model starts, the engine tells it where the change belongs, which related places may need the same change, and which tests cover them; ByteAsk then checks the change against only those tests. We compare ByteAsk with OpenAI's Codex CLI running the same underlying model, on recent issues from popular and little-known C++ projects and on a small set of hard issues. Both agents fix the same number of runs in every set (129 of 144 each, across the two main sets), but ByteAsk uses 63–80% fewer tokens and costs 59–70% less.

1 Introduction

Coding agents now fix real issues in real projects, but in C and C++ much of their effort goes into finding where to work rather than into the fix. The meaning of a line of C++ depends on the build. Whether `book.apply(msg)` calls one overload or another, a template specialisation, or a function generated by a macro depends on include paths, macros, language options and conditional compilation, all of which differ from one build configuration to the next [6]. Text search sees none of this: it returns every function with the right name, every disabled `#if` branch and every unused overload, and leaves the model to sort them out by reading.

A fix is also rarely in one place. A bug in one function often has siblings: the same pattern in an overload, a specialisation for another type, a parallel code path for another platform, or a declaration and its tests that must change together. An agent that finds the first place and stops leaves the rest broken; an agent that searches for all of them reads large parts of the project to do so. Checking a change is just as costly: building a large C++ project and running its whole test suite takes minutes, so agents either skip the check or pay for it in time and tokens.

The result is that a general coding agent spends most of its model calls and tokens searching, reading and rebuilding. Those tokens are the main cost of running an agent, and they grow with the size of the project. ByteAsk is built to move that work out of the model's loop. We describe its approach (Section 2) and compare it with Codex CLI on the same model (Section 4).

2 The approach

ByteAsk is built on three ideas (Figure 1).

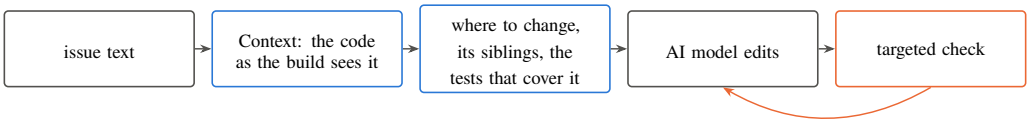


Figure 1: ByteAsk's approach. Before the model starts, Context identifies where the change belongs, which related places may need the same change and which tests cover them; the model edits, and a check limited to those tests confirms the change.

Understand the code as the build does. ByteAsk Context resolves names, overloads, templates, macros and conditional code for the project's actual build configuration, so a question about the code gets one precise answer rather than every textual match. It keeps that understanding current as the code changes.

Answer before the model asks. From the issue text alone, Context finds the places where a change most likely belongs, the sibling functions and related places that usually change with them, and the tests that exercise them. The model starts with that answer instead of searching for it, and can ask Context further questions as it works.

Check only what the change reaches. Instead of rebuilding the project and running every test, ByteAsk checks a change against the code and tests it actually reaches, so the model gets a fast, relevant answer on whether its fix works.

The aim is fewer turns spent searching and reading, fewer full builds, and so fewer tokens and lower cost for the same fixes.

3 How we compare

Both agents run GPT-6.1 Sol at medium reasoning effort with the same task prompt, a fresh clone and a fresh agent home per run, a 45-minute limit and the same resources (5 GB of memory, three CPUs); the two arms of a task start together. The prompt gives the issue text and a few rules: do not change the maintainers' test directories, keep the change focused, build the code and run the relevant tests, do not use the internet, and stop when done; the working tree is the answer. Codex CLI 0.164.0 runs as shipped against OpenAI's API. Context prepared each repository before the runs, and that time is not included in the run time. An issue is fixed when the agent's change, applied to a fresh copy of the repository, passes every fail-to-pass test and keeps every pass-to-pass test passing, offline. Token counts cover input (including cached input) and output; cost uses list prices; both include all of ByteAsk's own model calls. We report ratios of sums over paired runs. Each study's arms, task sets and analysis were written down before its runs. Savings are rounded down; a change is described as fewer or lower only when its 95% interval excludes zero, and the intervals are in the repository.

4 Results

Task sets. *Held-out issues*: 23 issues from popular C++ projects, fixed in pull requests merged after the model's training cutoff; two runs each. *Little-known projects*: 49 issues from little-known C++ projects with real linked issues, none of which we looked at before the runs; two runs each. *Hard issues*: 5 issues from these pools that Claude Code failed in a selection run, each run three times with a later Context build. Keeping only the tasks one agent failed and then comparing would bias the result against that agent, because some failures are run-to-run noise. We therefore discarded the selection run, split the hard pool at random into a development half and a held-out half, and gave every agent fresh runs on the held-out half only.

The same fixes at a fraction of the tokens and cost. Table 1 gives the results. Both agents fixed the same number of runs in every set, with no detected difference: 44 of 46 each on held-out issues, 85 of 98 each on little-known projects (one run differs in each direction), 129 of 144 each pooled, and 9 of 15 each

Table 1: ByteAsk vs Codex CLI, both GPT-6.1 Sol at medium effort, with the same number of runs fixed in every set (each: held-out 44 of 46, little-known 85 of 98, pooled 129 of 144, hard 9 of 15). Values are means per run; savings are computed on sums over the paired runs. Tokens and cost include all of ByteAsk's own model calls. Hard issues: five tasks, three runs each, with a later Context build.

			Tokens per run			Estimated cost per run	
Task set	Tasks	Runs	Codex	ByteAsk	Saving	Codex / ByteAsk	Saving
Held-out issues, popular projects	23	46	727k	263k	63.9%	$0.227 / $0.092	59.9%
Issues in little-known projects	49	98	2,296k	457k	80.1%	$0.456 / $0.137	70.1%
Both sets pooled	72	144	1,795k	395k	78.0%	$0.383 / $0.122	68.1%
Hard issues	5	15	2,085k	400k	80.8%	$0.441 / $0.142	67.9%

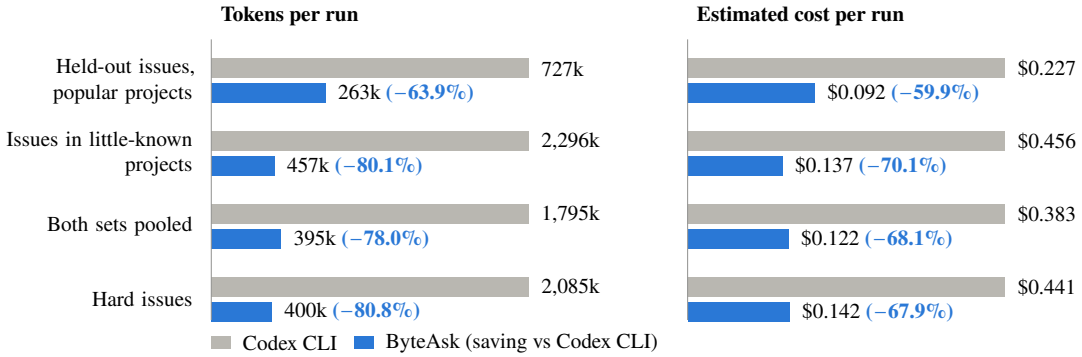


Figure 2: Tokens and estimated cost per run, Codex CLI beside ByteAsk, on the same model and tasks with the same number of runs fixed. Each row is scaled to Codex CLI (full length); the bold figure is ByteAsk's saving, computed on sums over the paired runs.

on hard issues. With the same fixes, ByteAsk used 63.9% fewer tokens on held-out issues, 80.1% fewer on little-known projects and 80.8% fewer on hard issues, at an estimated cost that was 59.9%, 70.1% and 67.9% lower.

The savings are not driven by a few runs: ByteAsk used fewer tokens than Codex CLI in 143 of the 144 paired runs of the two main sets and cost less in 144 of them (Figure 2).

Where the savings come from. ByteAsk made 33.7% fewer model calls than Codex CLI on held-out issues, 62.3% fewer on little-known projects and 67.2% fewer on hard issues (Table 2). This is consistent with the likely places and their code arriving in the first turn, so the model has less to search and read, and with a targeted build standing in for full builds. Each call also carried fewer tokens on average. Over the 144 paired runs of the two main sets, each fixed issue cost $0.427 with Codex CLI and $0.136 with ByteAsk.

5 Related work

Most agents locate code by searching and reading [9, 11], an approach Anthropic's guidance for coding agents prefers over precomputed retrieval, while noting that hybrids often work best [8]; others retrieve code [12] or explore repository graphs [1, 3, 5, 7]. ByteAsk instead works from the code as the configured build sees it and delivers its answer before the model asks. Harness design alone changes results [2, 10, 13], and agent evaluations should report cost alongside accuracy [4].

6 Conclusion

ByteAsk combines a build-aware understanding of C++ code with a first-turn answer and targeted checks. On the same model as Codex CLI, it fixed the same number of runs with 63–80% fewer tokens and 59–70% lower estimated cost.

Table 2: Estimated cost per fixed run and model calls per run, Codex CLI beside ByteAsk, for the same runs as in Table 1.

Task set	Cost per fixed run, Codex / ByteAsk	Model calls per run, Codex / ByteAsk	Fewer model calls
Held-out issues, popular projects	\$0.237 / \$0.096	19.6 / 13.1	33.7%
Issues in little-known projects	\$0.525 / \$0.158	41.1 / 15.5	62.3%
Both sets pooled	\$0.427 / \$0.136	34.2 / 14.7	57.1%
Hard issues	\$0.735 / \$0.236	36.8 / 12.1	67.2%

References

[1] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, 2025. doi: 10.18653/v1/2025.acl-long.426.

[2] Run-Ze Fan, Zihao Zhang, Simin Ma, Yebowen Hu, Shouju Wang, Kaiqiang Song, Fei Liu, Hamed Zamani, and Xiaoyang Wang. An empirical study of harness design for coding agents, 2026. URL https://arxiv.org/abs/2609.20804.

[3] Yuntong Hu, Tongli Su, Liang Zhao, Bowen Zhu, and Hasibul Haque. LARGER: Lexically Anchored Repository Graph Exploration and Retrieval, May 2026. URL https://arxiv.org/abs/2605.16352.

[4] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *Transactions on Machine Learning Research*, 2025.

[5] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL), Volume 1: Long Papers*, pages 142–160, 2025.

[6] LLVM clangd contributors. Compile commands. clangd design documentation, 2026. URL https://clangd.llvm.org/design/compile-commands. Living documentation; snapshot accessed 2026-11-04.

[7] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.14684.

[8] Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield. Effective context engineering for AI agents. Anthropic Engineering, 2025. URL https://www.anthropic.com/research/effective-context-engineering-for-ai-agents.

[9] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741.

[10] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi: 10.1145/3715754. Article FSE037.

[11] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2405.15793.

[12] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023. doi: 10.18653/v1/2023.emnlp-main.151.

[13] Yingwei Zheng, Cong Li, Shaohua Li, Yuqun Zhang, and Zhendong Su. Agentic Harness for Real-World Compilers, March 2026. URL https://arxiv.org/abs/2603.20075. Results checked against version 2, revised July 2026.