Build-Aware Code Context at Million-Line Scale: ByteAsk Context on LLVM, the Linux Kernel and Large C/C++ Repositories

Pratyush Saini, CTO, ByteAsk Research pratyush@byteask.ai
Anirudha Kulkarni, CEO, ByteAsk Research anirudha@byteask.ai

Code and data: github.com/ByteAsk/context-large-codebases-cpp

Abstract

Large language model (LLM) coding agents now fix real issues in real projects, but most of that progress has come in a few widely used languages. C and C++ remain far less explored, although they underpin operating systems, compilers and embedded software, and their largest codebases run to millions of lines. There, the meaning of a name depends on how the project is built: macros, templates, overloads and conditional code make text search return many candidates and leave the agent to sort them out. ByteAsk Context understands a codebase the way the project's own build does and tells an agent where a change belongs. On one 48-CPU machine it was ready to answer on the Linux kernel's build in 9.3 minutes and on LLVM's in 13.7, while Graphify did not finish LLVM in 3 hours. After a source-file edit it re-indexed in 5.8 seconds, against 105–112 seconds for a full index (an edit to a widely included header took 87.3 seconds), and 9 of 10 answers after an edit matched the new code, against 0 of 10 from a stale index. On held-out issues it ranked a changed file first in 57% of tasks, against 30% for sgrep and 20% for Probe.

1 Introduction

A coding agent in a million-line C or C++ codebase spends its first steps finding the code that a task touches, and in these languages that is hard. The meaning of a line depends on the build. The Linux kernel defines `setup_arch` once per architecture, and only the configured one is compiled. Macros generate functions that appear nowhere as text, and configuration options switch whole blocks of code on and off. In LLVM, deep templates and overloads decide which function a call reaches, and many headers are written by the build itself. Which definition a call reaches is fixed by the compile commands of one configured build [10]. A text index sees every copy; the build sees one.

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, or a parallel path for another platform. And everything is large: a search that is cheap on a small project returns thousands of matches on the kernel, and a tool that must read the whole tree before it can answer may not finish at all.

ByteAsk Context [2] is built for this setting. We ask what a build-aware index costs and what it buys on the largest C and C++ codebases, against tools that take other routes: semantic search (claude-context [25], sgrep [4]), keyword search (Probe [14]) and a syntax-tree graph (Graphify [5]). We find:

- **One machine indexes a kernel or LLVM build** in 9.3 and 13.7 minutes, while Graphify did not finish LLVM in 3 hours (Section 4.1).
- **Edits cost seconds, and answers follow them** (Section 4.2): 5.8 s to re-index after a source-file edit instead of 105–112 s for a full index, and 9 of 10 answers right after an edit, against 0 of 10 from a stale index. An edit to a widely included header still costs close to a full index.
- **Retrieval holds up as codebases grow** (Section 4.3): on held-out issues, Context ranks a changed file first about twice as often as sgrep and about three times as often as Probe, and in the two largest projects three times as often as sgrep.

2 The approach

ByteAsk Context rests on three ideas.

Understand the code as the build does. Context reads a project's build configuration (a CMake build directory or a compile database from any build system [10]) and resolves names, overloads, templates, macros and conditional code for that configuration. Its index holds the files the configured build reads, generated headers included, and nothing outside it, so a question about the code gets the answer that applies to this build rather than every textual match.

Answer the task, not the query. From the task text as written, 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 answer stays short whatever the size of the repository.

Keep it current. An agent edits the code it asks about. Context re-indexes only what an edit reaches, and before each answer in a session it brings the edited files up to date, so answers follow the code as it is now.

3 Method

Index study. Each tool built its index once, from a cold start, on one NUMA node of a cloud machine (48 virtual CPUs, 150 GiB, one job at a time, a 3-hour limit), on Linux v2.6.32 (10.9M lines of C/C++, x86_64 default configuration) and LLVM (17.1M lines; `llvm/` with the X86 and AArch64 targets). We report the time until each tool could answer and the C/C++ files that Context indexed.

Re-index and freshness. On AgIsoStack-plus-plus, an ISOBUS library whose build has 86 translation units, we timed two full indexes and seven update cases, once each, with two index workers: no change, new commits, and an edit to one `.cpp` file or to the most-included header, committed or not. For freshness, after one uncommitted edit that adds a function and moves others, we asked 10 questions about the edited code (where a function is, what it calls, who calls it, and where a task belongs) and checked each answer for names, line numbers and calls.

Retrieval study. We used 78 issues from 16 C/C++ projects, none of them used while developing Context. A tool scores a hit when a file that the fix changes is its first answer (top-1) or among its first five (top-5). Context received the task text verbatim and ran three times per task; we report the mean. sgrep and Graphify received a query written by one fixed model (Claude Opus 5.5) from each tool's own documentation; Probe received both a keyword query in its own syntax and an agent-written query. Graphify ran its code-only extraction and sgrep a 64 GiB cache. A task counts for a tool only if the tool's index holds the changed files, and we leave out the 8 tasks whose issue names the location. Head-to-head comparisons use the tasks on which the competitor indexed every file that Context indexed. We say "more often" only when a repository-clustered 95% bootstrap interval and a cluster-robust t interval of the paired difference both lie above zero. Projects are grouped by C/C++ lines (under 1M, 1M to 5M, and 5M and over), a rule fixed before we viewed any result by size; results by size are descriptive.

4 Results

4.1 One machine indexes the configured build

Context built its index of the kernel's x86_64 build and answered a first question 557 s after a fresh start (Table 1): 509 s to index 6,681 translation units with 48 workers, and 48.1 s for the first answer. The index covers 11,495 of the tree's 24,785 C/C++ files (46.4%): the files that this configuration compiles or includes. Of the 13,286 files it leaves out, 9,119 belong to other architectures and 3,161 to drivers that

Table 1: Cold index builds on one machine: time from a fresh start until the tool can answer. Context's time includes its first answer. claude-context stopped at its own limit of 450,000 chunks.

	Time until it can answer (s)	
Tool	Linux kernel	LLVM
ByteAsk Context	557	821
claude-context	10,469	10,599
Graphify	705	unfinished in 3 h

Table 2: Re-index times on AgIsoStack-plus-plus (86 translation units, two workers, one run each). TUs re-indexed: the engine's own count, where it was recorded.

Case	Time (s)	TUs re-indexed
Full index (earlier commit)	105	86
Full index	112	86
No change	1.2	–
One new commit	34.0	–
Five new commits	27.5	12
One `.cpp` edit, committed	5.8	1
One `.cpp` edit, uncommitted	5.0	–
Most-included header edit, committed	87.3	68
Header edit, uncommitted	92.4	–

the configuration does not build. On LLVM, indexing 6,982 files (9.8%) took 785 s, plus 36.1 s for the first answer.

Graphify finished the kernel in 705 s, but on LLVM it had parsed all 85,086 code files and had not yet built its graph when the 3-hour limit stopped it. claude-context ran for almost three hours on each codebase before stopping at its chunk limit with a partial index.

4.2 Re-indexing and fresh answers after edits

A full index of AgIsoStack-plus-plus took 105–112 s (Table 2). With nothing changed, an update took 1.2 s. After an edit to one `.cpp` file, Context re-indexed 1 translation unit in 5.8 s when the edit was committed and in 5.0 s when it was not: about a twentieth of a full index. An edit to the most-included header is the expensive case: 68 of the 86 translation units depend on it, and the update took 87.3 s, close to a full index.

After one uncommitted edit, 9 of the 10 questions asked in one session were answered in line with the edited code: the new function found, current line numbers, and the new call in the call lists. The engine before this feature answered from the committed index and got 0 of 10. A question asked as the first call of a new session still gets the committed index's answer (0 of 10), each marked as possibly stale.

4.3 Retrieval by repository size

Context's configured builds hold only 24.3% of the C/C++ files in these projects, yet they hold the changed files of 69 of the 70 tasks that Context indexed: the code a fix touches is code the build compiles. On 62 tasks, Context ranked a changed file first in 54% and in the top five in 66% (Table 3).

Head to head, on the 30 tasks in 7 projects where sgrep and Probe indexed every file that Context did, Context ranked a changed file first in 57% of tasks, sgrep in 30%, Probe with an agent-written query in 20% and Probe with keywords in 13%. Context did so more often than each, by 27, 37 and 43 percentage points, and the same holds on all tasks that both tools could answer. Context's answers were also shorter: a median of 9.2K characters, against 30.4K for sgrep.

By size, Context's top-1 rate was no lower in the largest bin than in the smallest, while every other tool's rate was lower there. In the two projects of 5M lines and more, Context ranked a changed file first in 60% of its tasks, sgrep in 20% and Probe with keywords in 0%. With two projects and 10 tasks, these are counts, not estimates.

Table 3: Held-out issues: how often a file that the fix changes was each tool's first answer (top-1) or among its first five (top-5), by project size in C/C++ lines, in the columns where Context leads every other tool. Each row counts the tasks that its index covers (Tasks, all sizes), without issues that name the location. Context: the mean of three runs.

		All sizes 16 repos		under 1M 6 repos		1M–5M 9 repos	5M and over 2 repos	
Tool	Tasks	Top-1	Top-5	Top-1	Top-5	Top-1	Top-1	Top-5
ByteAsk Context	62	54%	66%	47%	68%	58%	60%	70%
sgrep	70	30%	61%	28%	55%	35%	20%	50%
Graphify	67	22%	37%	19%	33%	30%	10%	20%
Probe (keywords)	70	14%	31%	21%	45%	13%	0%	10%
Probe (agent query)	69	32%	61%	31%	66%	40%	10%	10%

5 Related work

Kythe indexes each compile action of a build [17], and Glean runs a clang-based indexer in place of the compiler [11]. Agents locate code with file listings, search and reading [20, 21], or with repository maps and code graphs [1, 3, 6, 9, 12, 19]; retrieval studies compare these techniques [7, 15, 16]. Issue benchmarks reach large systems [8, 13, 22, 23], including Linux [24] and LLVM [18].

6 Conclusion

A build-aware index pays its full cost once per configured build: on one machine, Context indexed the kernel's build in 9.3 minutes and LLVM's in 13.7, where Graphify did not finish LLVM. After that, a source-file edit costs seconds and a session's answers follow the edit, though a widely included header still costs close to a full index. On held-out issues Context ranked a changed file first about twice as often as sgrep and about three times as often as Probe, and its rate held in projects of 5M lines and more.

References

[1] Aider maintainers. Building a better repository map with tree sitter. Aider technical blog, October 2023. URL https://aider.chat/2023/10/22/repomap.html. Published 2023-10-22; accessed 2026-11-04.

[2] ByteAsk. ByteAsk Context. https://byteask.ai/context, 2026. Accessed 2026-11-09.

[3] 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.

[4] context.dev. sgrep: Search code and documents in plain English. Command-line tool, version 0.3.0 at commit dec6122, 2026. URL https://github.com/context-dot-dev/sgrep. README fetched 2026-11-07.

[5] Graphify Labs. graphifyy 0.9.103. Python Package Index, 2026. URL https://pypi.org/project/graphifyy/0.9.103/. Package description fetched 2026-11-07.

[6] 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.

[7] Priyanshu Jain. An exploratory study of code retrieval techniques in coding agents. Preprints.org, manuscript 202510.0924, version 1, 2025. URL https://www.preprints.org/manuscript/202510.0924. Not peer-reviewed.

[8] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

[9] 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. URL https://aclanthology.org/2025.naacl-long.7/.

[10] LLVM Project contributors. JSON compilation database format specification. Official Clang documentation, n.d. URL https://clang.llvm.org/docs/JSONCompilationDatabase.html. Accessed 2026-11-04.

[11] Meta. Glean: Open source code indexing. Engineering at Meta blog, December 2024. URL https://engineering.fb.com/2024/12/19/developer-tools/glean-open-source-code-indexing/.

[12] 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.

[13] Yun Peng, Zihan Wu, Zeyang Zhuang, Xin Zhou, Rui Shu, Xu Han, Chun Yong Chong, Yuan Wang, and Jiakun Liu. LoLBench: Evaluating Coding Agents with Long-Horizon Proposals on Large Software Systems, September 2026. URL https://arxiv.org/abs/2609.37143.

[14] Probe Labs. Probe: AI-friendly semantic code search. Command-line tool and MCP server (`probe mcp`), version 0.6.0-rc345, 2026. URL https://github.com/probelabs/probe. README and MCP integration guide fetched 2026-11-07.

[15] Bowen Qin and Yi Xie. Agent Retrieval Bench: Evaluating Repository Context Retrieval for Coding Agents, July 2026. URL https://arxiv.org/abs/2607.24882.

[16] Sourcegraph. CodeScaleBench: A Systematic Evaluation Framework for Assessing the Impact of Enhanced Code Intelligence on AI Coding Agent Performance. Sourcegraph technical report on GitHub; last modified March 5, 2026, March 2026. URL https://github.com/sourcegraph/CodeScaleBench/blob/public/docs/technical_reports/TECHNICAL_REPORT.md. Accessed 2026-11-05.

[17] The Kythe Authors. Kythe compilation database. Kythe documentation, n.d. URL https://kythe.io/docs/kythe-compilation-database.html.

[18] Zhao Tian, Yingquan Zhao, Chenyao Suo, Meng Wang, and Junjie Chen. LLVM-Bench: Benchmarking and Advancing Large Language Models for LLVM Compiler Issue Resolution, July 2026. URL https://arxiv.org/abs/2607.00700.

[19] Martin Vogel, Falk Meyer-Eschenbach, Severin Kohler, Elias Grünewald, and Felix Balzer. Codebase-Memory: Tree-sitter-based knowledge graphs for LLM code exploration via MCP. arXiv preprint arXiv:2603.27277, 2026. URL https://arxiv.org/abs/2603.27277.

[20] 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.

[21] 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.

[22] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2111.

[23] Zejun Zhang, Jian Wang, Qingyun Yang, Yifan Pan, Yi Tang, Yi Li, Zhenchang Xing, Tian Zhang, Xuandong Li, and Guoan Zhang. A benchmark for localizing code and non-code issues in software projects, 2025. URL https://arxiv.org/abs/2509.25242.

[24] Zhenhao Zhou, Zhuochen Huang, Yike He, Chong Wang, Jiajun Wang, Yijian Wu, Xin Peng, and Yiling Lou. Taming System Complexity: Demystifying Software Engineering Agents in Diagnosing Linux Kernel Faults. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18899–18916. Association for Computational Linguistics, July 2026. doi: 10.18653/v1/2026.acl-long.862. URL https://aclanthology.org/2026.acl-long.862/. First arXiv submission May 2025.

[25] Zilliz. claude-context: Semantic code search for coding agents. MCP server; embeddings from an external provider (OpenAI, or Ollama) and a Milvus or Zilliz Cloud vector database, 2026. URL https://github.com/zilliztech/claude-context. README checked 2026-11-05.