TypeScript Repository Indexing for Code Agent Retrieval

Junsong Pu
Sun Yat-sen University
Zhuhai, China
pujs@mail2.sysu.edu.cn

Yichen Li*
Chinese University of Hong Kong
Hong Kong, Hong Kong
ycli21@cse.cuhk.edu.hk

Zhuangbin Chen*
Sun Yat-sen University
Zhuhai, China
chenzhb36@mail.sysu.edu.cn

Abstract

Graph-based code indexing can improve context retrieval for LLM-based code agents by preserving call chains and dependency relationships that keyword search and similarity retrieval often miss. ABCoder is an open-source framework that parses codebases into a function-level code index called UniAST. Its existing parsers combine lightweight AST parsers for syntactic analysis with language servers for semantic resolution, but because such resolution requires a JSON-RPC call for each symbol lookup, these per-symbol calls become a bottleneck on large TypeScript repositories. We present `abcoder-ts-parser`, a TypeScript parser built on the TypeScript Compiler API that works directly with the compiler's AST, semantic information, and module resolution logic. We evaluate the parser on three open-source TypeScript projects with up to 1.2 million lines of code and find that it produces reliable indexes significantly more efficiently than the existing architecture. For a live demonstration, watch: https://youtu.be/ryssr7ouvdE

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

Code Agents, Code Indexing, TypeScript, ABCoder

ACM Reference Format:
Junsong Pu, Yichen Li, and Zhuangbin Chen. 2026. TypeScript Repository Indexing for Code Agent Retrieval. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3832783.3834616

1 Introduction and Background

1.1 Context Retrieval for Code Agent

1.1.1 Code Agent. Recent advances in Large Language Models (LLMs) have given rise to a new class of development tools known as *LLM-based code agents*. Systems such as Claude Code [2] operate as autonomous assistants capable of performing a wide range of software engineering tasks with minimal human intervention. At their core, these agents follow a ReAct-style [17] loop of reasoning and acting: given a developer's request, the agent first gathers relevant context from the project, then reasons over the gathered information to decide on an action, observes the outcome, and loops back if the result is unsatisfactory. This cycle repeats until the task is completed or the agent determines that human input is needed.

1.1.2 Efficient Context Retrieval. The effectiveness of a code agent depends heavily on the quality of its *context retrieval*, which we define as the process of locating and assembling the relevant code fragments that the model needs to reason about a given task. Benchmarks and systems on real repository-level software engineering tasks consistently show that solving a single issue often requires coordinating changes across multiple functions, classes, and files, while identifying only a subset of repository context as truly relevant [6, 9, 10, 15, 16]. Because a language model can only reason about the code that fits within its prompt, poor retrieval leads to incomplete or misleading input, which in turn degrades the model's reasoning even when its underlying capabilities are sufficient [9]. For example, when fixing a bug that spans multiple functions across several files, the model needs to see not only the function where the error manifests but also the functions that call it and the types it depends on; if the retrieval step returns only the direct function, the model often fails to find the root cause and might suggest an incomplete or incorrect patch. In this sense, the quality of context retrieval directly shapes the quality of the model's downstream reasoning, and improving retrieval is a practical path to improving agent performance on complex software engineering tasks.

Existing approaches to context retrieval in code agents fall broadly into three categories. The first is *command-line exploration*, where the agent executes shell commands such as `grep`, `find`, or other utilities to locate relevant files and symbols. Software engineering agents like SWE-agent [16] often rely on this method to navigate codebases. It has the notable advantage of requiring no upfront indexing: it works on any codebase immediately without building or maintaining any auxiliary data structures, making it the simplest strategy to deploy. However, command-line exploration is fundamentally keyword-driven, which causes problems in both directions: it misses relevant code that does not share lexical overlap with the query, and it returns irrelevant results that happen to contain the same keywords, forcing the agent to spend additional rounds filtering out noise. It also places a heavy burden on the agent's planning ability, as the agent needs to decide which commands to run, interpret their outputs, and iteratively refine its search, often consuming many interaction rounds before arriving at useful results. This burden has motivated alternative pipelines such as Agentless [15], which separate localization from repair rather than relying on open-ended interaction throughout the entire loop.

The second is similarity-based retrieval, where systems fetch relevant code snippets using either keyword matching algorithms or semantic vector embeddings. Similarity-based retrieval-augmented systems such as RepoCoder [18] and Repoformer [14] show that this strategy can improve repository-level completion quality over

*Corresponding authors.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834616

purely local baselines. While similarity retrieval is effective at matching natural language queries to relevant code snippets, it is less suited to reasoning about function behavior, which requires capturing structural relationships such as call chains and type dependencies that similarity representations do not encode. Large-scale analyses such as CodeRAG-Bench [13] further show that current similarity retrievers still struggle to fetch useful contexts when lexical overlap is weak, and downstream models need to further analyze the retrieved fragments to reason about the relationships between code elements.

The third category is graph-based code indexing, where a parser performs static analysis on a repository to extract entities such as functions, classes, and modules along with their call and dependency relationships, and organizes them into a queryable graph. This approach preserves the structural information that the previous two methods discard. Recent graph-based retrieval systems such as RepoGraph [9] explicitly model repository structure for downstream software engineering agents. ABCoder [3] is a representative system in this category: it parses a codebase into a function-level graph and builds a Graph RAG that allows an agent to retrieve not just a single function but also its related code entities, including callers, callees, and related type definitions.

1.2 ABCoder Framework

1.2.1 Overview. ABCoder is a toolset that builds a graph-based code index from source code and provides query utilities to improve context retrieval for code agents. Given a repository, it parses the code into a language-agnostic format called UniAST (Universal Abstract Syntax Tree). This format captures code entities like functions, types, and variables alongside their dependencies. LLM agents can then use ABCoder's query tools via the Model Context Protocol (MCP) [1] to navigate the resulting code index for tasks like codebase exploration and cross-file dependency tracing. By supplying a relational graph instead of raw text, the toolset helps the model analyze code as a network of connected entities rather than isolated snippets.

1.2.2 Performance on Large Repositories. Existing ABCoder parsers combine the Language Server Protocol (LSP) [7] with lightweight AST parsers such as Tree-sitter [12]. In this architecture, Tree-sitter scans each source file to identify all code entities and potential dependency sites such as call expressions, type references, and import statements. Since Tree-sitter is a syntax-level parser and does not resolve symbols to their definitions, it cannot determine what a given symbol actually refers to. For each dependency site, the parser therefore sends a request to the language server, which communicates via `JSON-RPC`, to resolve the symbol's definition. The language server returns a file location, which the parser maps back to an entity in Tree-sitter's syntax tree of the target file, thereby establishing an edge in the dependency graph.

This architecture borrows from how most modern code editors work: Tree-sitter provides fast syntax highlighting and structural navigation, while the language server handles semantic queries on demand. It is also largely language-agnostic, as adding support for a new language mainly requires installing the corresponding language server and Tree-sitter grammar, without changing the indexing framework itself. When a developer hovers over a symbol in an editor to see its definition, the cost of resolving one symbol at a time is negligible.

Listing 1: A minimal example for TypeScript indexing.

```ts
// user-repo.ts
export class UserRepo {
  getById(id: string) { return id; }
}

// index.ts
export { UserRepo as Repo } from "./user-repo";

// service.ts
import { Repo } from "./index";
export function loadUser(id: string) {
  const repo = new Repo();
  return repo.getById(id);
}
```

For large repository indexing, however, this cost accumulates quickly. Listing 1 illustrates this with a small TypeScript example. Tree-sitter scans `service.ts` and identifies the function `loadUser` as an entity, along with three dependency sites: the import of `Repo`, the constructor call `new Repo()`, and the method call `repo.getById(id)`. Each of these sites requires at least one language-server request to resolve, and the import of `Repo` requires an additional request, because it first resolves to the *barrel file* `index.ts`, a module that re-exports symbols from other files to provide a single entry point for the package, and a follow-up request is needed to trace the re-export to `UserRepo` in `user-repo.ts`. Each response must also be mapped back to Tree-sitter's syntax tree again. For a large TypeScript project with thousands of functions, the total number of RPC calls can scale significantly. Since the LSP resolves symbols one position at a time and does not offer batch resolution, this makes repository indexing slow for large TypeScript projects. To avoid this, we build our parser directly on the TypeScript Compiler API, which loads the entire project into a single in-process compiler instance and provides the AST, semantic information, and module resolution results without any external RPC calls.

2 System Design

We submitted the first version of `abcoder-ts-parser` to the ABCoder project in September 2025. Since then, we have been maintaining and developing the tool in collaboration with the ABCoder open-source community. As of April 2026, the parser contains about 6 KLOC of TypeScript code, excluding test code. This section describes its usage and output format, followed by how the parser reduces the indexing overhead discussed earlier.

2.1 Tool Overview

2.1.1 Usage. Our TypeScript parser is distributed as a standalone command-line tool called `abcoder-ts-parser`. Given a TypeScript project directory and the paths to its metadata files, it performs a one-pass static analysis and produces a single JSON file conforming to ABCoder's UniAST specification [4]. The tool also supports a monorepo mode for repositories that contain multiple packages under a single root. During parsing, the tool reads these metadata files to determine project configuration and dependencies.

2.1.2 Output Format. The output follows UniAST's hierarchical structure, summarized in Table 1. At the top level, the output contains a set of modules and a repository code index. In a monorepo, each constituent project is represented as a separate module; in a single-project repository, there is exactly one module that corresponds to the repository itself. Each module contains one or more packages, where each package maps to a single source file. Each package contains three kinds of entities: functions, types, and variables. For every entity, the parser records its unique identity (a triple of module path, package path, and symbol name), its complete source text, its file location, and its signature.

Table 1: UniAST output structure.

Level	Contents
Repository	Modules, repository code index
Module	Packages, Dependencies, Files
Package	Functions, Types, Variables
Entity	Identity, source text, file location, signature, per-entity dependency list

The repository code index is what distinguishes UniAST from a flat collection of code snippets. It aggregates the dependency lists of all entities into a global structure. Each node in this structure represents one code entity, and each edge encodes a relationship between two entities. Table 2 describes the four kinds of relations captured by the graph.

Table 2: Relation types in the UniAST dependency graph.

Relation	Semantics
Dependency	Entity A calls, references, or otherwise depends on entity B.
Reference	The inverse of Dependency: entity B is depended upon by entity A.
Implementation	A class or object implements the contract defined by an interface.
Group	Entities declared together as a logical unit, e.g., members of a single `enum` or `const` block.

These relations support different forms of structured retrieval. Dependency and Reference edges form a bidirectional link between related entities, allowing a code agent equipped with a code index retrieval tool to traverse the graph in both directions. For example, given a function that a developer is asking about, the agent can follow its Dependency edges outward to gather the functions it calls and the types it uses, and follow its Reference edges inward to gather the functions that call it. This directly provides the agent with the relevant call and dependency information for that entity, rather than returning isolated fragments that happen to share similar keywords or embeddings. Implementation edges connect interface definitions to their concrete implementations, which is useful in TypeScript codebases where components are often consumed through abstract interfaces.

2.2 Parsing with the TypeScript Compiler API

As discussed in Section 1.2, resolving symbols and dependencies through a separate language server requires many RPC calls between the parser and the server, one for each dependency site in each file. Our parser avoids this by loading the entire project into a single compiler instance, from which the AST, semantic information, and module resolution results are all directly available in memory. The parser walks the compiler's AST and reduces it into a code index following the UniAST specification. It identifies functions, types, and variables, records their signatures, source text, and locations, and extracts the dependency relationships between them, including direct calls, method calls, constructor invocations, and type references. During this process, a symbol resolver traces each referenced symbol along its import chain, following aliases, re-exports, and barrel files until it reaches the original declaration, so that every dependency edge in the output graph points to the actual definition rather than an intermediate re-export.

3 Evaluation

3.1 Testing and Validation

For a code parser, validation focuses on whether entities are identified accurately, dependency edges point to the intended targets, and cross-file symbol resolution follows import chains to the actual definition. Unit testing is a natural fit for checking these properties, as each parsing rule can be tested against specific code patterns with known expected outputs. We therefore use the parser's unit test suite and its code coverage metrics as our primary validation evidence.

Since its initial release in September 2025, `abcoder-ts-parser` has been under continuous development by the ABCoder open-source community. Over the course of eight months, the test suite has grown to 10 test suites comprising 175 individual test cases and about 7 KLOC of test code, of which 174 currently pass. The single failing test is due to a discrepancy in how monorepo root packages are counted and does not affect the core parsing logic.

Table 3: Test coverage of key modules.

Module	Line Cov.	Branch Cov.
FunctionParser	85.36%	52.00%
TypeParser	93.18%	76.10%
VarParser	68.66%	50.92%
ModuleParser	87.05%	58.06%
PackageParser	100%	100%
graph-builder	98.75%	95.65%
symbol-resolver	77.08%	66.40%
Overall	*71.75%*	*51.73%*

Table 3 reports the line and branch coverage for the major components of the parser. The core parsing and graph construction modules generally achieve line coverage above 80%, with several modules reaching above 93%. The overall line coverage across all source files is 71.75%, with the lower figure mainly due to infrastructure code such as the CLI entry point and the parallel processing layer, which are not easily covered by unit tests alone.

3.2 Parsing Performance

To evaluate the parser's efficiency on real-world codebases, we benchmark it on three open-source TypeScript projects of varying scale: Excalidraw [5], an online collaborative whiteboard application; Outline [8], a team knowledge base and document collaboration platform; and Sentry [11], a large-scale error monitoring and performance tracking platform. All experiments run on a machine with an Intel Xeon Platinum 8336C CPU (32 cores at 2.30 GHz) and 62 GB of RAM, with the Node.js process allocated a 16 GB heap.

Table 4: Parsing performance on three open-source TypeScript projects.

Repository	Files	Lines of Code	Time (s)
Excalidraw	601	147K	35.2
Outline	1,875	234K	239.9
Sentry	8,909	1.23M	705.6

As shown in Table 4, the parser handles projects ranging from several hundred to nearly nine thousand source files. For a large project like Sentry, the largest project at over 1.2 million lines of code, the full analysis completes in under 12 minutes. Since the index only needs to be rebuilt when the codebase changes significantly, this one-time cost is acceptable in practice. For smaller projects like Excalidraw, the parsing finishes in about 35 seconds, making the tool suitable for integration into regular development workflows. As a point of comparison, ABCoder's existing Python parser, which uses a lightweight AST parser plus LSP approach, takes 603 seconds to parse Django (160 KLOC) under the same hardware setting. Our parser analyzes a comparable amount of TypeScript code (Excalidraw, 147 KLOC) in 35 seconds, suggesting that avoiding individual symbol-resolution RPC calls yields a substantial reduction in parsing time.

3.3 Downstream Usage

Once the parser produces a UniAST index, it can serve a variety of downstream purposes. The primary use case is graph-based retrieval for code agents. ABCoder provides an MCP server that loads the index and exposes a set of query tools, allowing agents such as Claude Code to navigate the codebase and retrieve a specific function along with its callers, callees, and type dependencies. Because the index is organized at function-level granularity, each retrieval step returns a semantically meaningful unit of code rather than an arbitrary text chunk, reducing irrelevant context in the model's prompt. Another use case is automated codebase documentation. The index provides a precomputed context slice for each function, containing its callers, callees, and type dependencies, so the language model can describe the function's role directly without scanning raw source files to discover these relationships.

4 Data Availability

`abcoder-ts-parser` is publicly available in the ABCoder repository on GitHub at https://github.com/cloudwego/abcoder/tree/main/ts-parser [3]. Also available on npm: https://www.npmjs.com/package/abcoder-ts-parser

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62402536). We thank the CloudWeGo team for their support, especially Wenju Gao, Zhenyang Dai, Xuran Yin, Zekun Wang, and Yi Duan for their sustained collaboration.

References

[1] Anthropic. 2024. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol Published 2024-11-25.
[2] Anthropic. 2026. Claude Code Overview. https://docs.anthropic.com/en/docs/claude-code/overview Official documentation, accessed 2026-04-14.
[3] CloudWeGo Team. 2026. ABCoder: AI-Based Coder (AKA: A Brand-new Coder). https://github.com/cloudwego/abcoder GitHub repository, accessed 2026-04-14.
[4] CloudWeGo Team. 2026. UniAST Specification. https://github.com/cloudwego/abcoder/blob/main/docs/uniast-en.md GitHub documentation, accessed 2026-04-14.
[5] Excalidraw. 2026. excalidraw/excalidraw. https://github.com/excalidraw/excalidraw GitHub repository, "Virtual whiteboard for sketching hand-drawn like diagrams", accessed 2026-04-14.
[6] Yichen Li, Jinyang Liu, Junsong Pu, Zhihan Jiang, Zhuangbin Chen, Xiao He, Tieying Zhang, Jianjun Chen, Yi Li, Rui Shi, and Michael R. Lyu. 2025. Automated Proactive Logging Quality Improvement for Large-Scale Codebases. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 3426–3437. doi:10.1109/ASE63991.2025.00283
[7] Microsoft. 2026. Language Server Protocol. https://microsoft.github.io/language-server-protocol/ Official documentation, version 3.17, accessed 2026-04-14.
[8] Outline. 2026. outline/outline. https://github.com/outline/outline GitHub repository, "The fastest knowledge base for growing teams", accessed 2026-04-14.
[9] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=dw9VUsSHGB
[10] Junsong Pu, Yichen Li, Zhuangbin Chen, Jinyang Liu, Zhihan Jiang, Jianjun Chen, Rui Shi, Zibin Zheng, and Tieying Zhang. 2025. ErrorPrism: Reconstructing Error Propagation Paths in Cloud Service Systems. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 3534–3545. doi:10.1109/ASE63991.2025.00292
[11] Sentry. 2026. getsentry/sentry. https://github.com/getsentry/sentry GitHub repository, "Developer-first error tracking and performance monitoring", accessed 2026-04-14.
[12] Tree-sitter Contributors. 2026. Tree-sitter. https://tree-sitter.github.io/tree-sitter/ Official documentation, accessed 2026-04-14.
[13] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-Bench: Can Retrieval Augment Code Generation?. In *Findings of the Association for Computational Linguistics: NAACL 2025*. Association for Computational Linguistics, Albuquerque, New Mexico, 3199–3214. doi:10.18653/v1/2025.findings-naacl.176
[14] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, Vienna, Austria, 53270–53290. https://proceedings.mlr.press/v235/wu24a.html
[15] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. doi:10.1145/3715754
[16] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. https://openreview.net/forum?id=mXpq6ut8J3 NeurIPS 2024.
[17] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.
[18] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 2471–2484. doi:10.18653/v1/2023.emnlp-main.151

Received 2026-05-09; accepted 2026-06-19