cAST: Enhancing Code Retrieval-Augmented Generation with Structural Chunking via Abstract Syntax Tree

Yilin Zhang[1*] **Xinran Zhao**[1] **Zora Zhiruo Wang**[1] **Chenyang Yang**[1]
Jiayi Wei[2] **Tongshuang Wu**[1]
[1]Carnegie Mellon University, [2]Augment Code

Abstract

Retrieval-Augmented Generation (RAG) has become essential for large-scale code generation, grounding predictions in external code corpora to improve factuality. However, a critical yet underexplored aspect of RAG pipelines is chunking—the process of dividing documents into retrievable units. Existing line-based chunking heuristics often break semantic structures, splitting functions or merging unrelated code, which can degrade generation quality. We propose chunking via Abstract Syntax Trees (cAST), a structure-aware method that recursively breaks large AST nodes into smaller chunks and merges sibling nodes while respecting size limits. This approach generates self-contained, semantically coherent units across programming languages and tasks, improving performance on diverse code generation tasks, e.g., boosting Recall@5 by 4.3 points on RepoEval retrieval and Pass@1 by 2.67 points on SWE-bench generation. Our work highlights the importance of structure-aware chunking for scaling retrieval-enhanced code intelligence.

1 Introduction

Large-scale code generation has emerged as a cornerstone of modern software engineering, powering tasks that range from automated bug fixing (Meng et al., 2024) to full-fledged repository-level completion (Zhang et al., 2023a). Retrieval-augmented generation (RAG) pushes this frontier further by allowing language models to ground their predictions in a rich external corpus of data (Guu et al., 2020), effectively mitigating hallucinations and improving factual correctness (Izacard et al., 2022).

One crucial preprocessing step in Retrieval-Augmented Generation (RAG) is chunking (Bohnet et al., 2023)—breaking large documents into manageable segments that can be efficiently indexed, retrieved, and used as contextual input during generation. To date, most chunking approaches rely on fixed-size, line-based splitting (Lewis et al., 2020). While simple and generally effective, this method struggles with structured content like code, where the document naturally contains semantic or syntactic blocks. As shown in Figure 1, naive chunking often splits meaningful units (e.g., functions and classes) across different chunks, losing structural integrity and context.

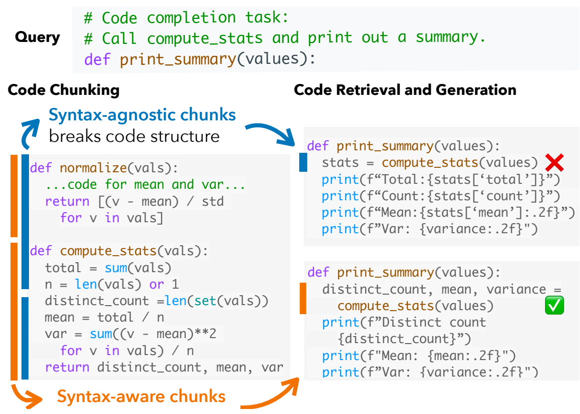

Figure 1: Syntax-agnostic chunking often omits crucial information needed to generate functional code. In this example, fixed-size chunking breaks the structure of the `compute_stats` method, causing the model to lose context regarding its return value. As a result, the model generates incorrect code based on a mistaken assumption of what is returned. In contrast, when given syntax-aware chunks, the model accurately identifies the return values and integrates them correctly within the existing codebase.

Can we chunk documents more intelligently, preserving their original structure? In this work, we explore cAST—Chunking via Abstract Syntax Trees. ASTs represent code as hierarchical trees with typed nodes corresponding to program units. By parsing source code into an AST, we apply a recursive, split-then-merge algorithm to convert tree structures into chunks that are better aligned with syntactic boundaries.

* Corresponding contact email addresses: {jasonzh3,sherryw}@andrew.cmu.edu. Our code is available at `https://github.com/yilinjz/astchunk`

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 8106–8116
November 4-9, 2025

Extensive experiments show that cAST improves performance across a range of code generation tasks. Specifically, it offers three key advantages: (1) *Structure-preserving chunks*: AST traversal yields more self-contained chunks, improving both retrieval and generation. For instance, StarCoder2-7B sees an average of 5.5 points gain on RepoEval (Zhang et al., 2023b). (2) *Cross-language consistency*: The language-agnostic nature of cAST enables better generalization across programming languages, achieving up to 4.3 points gain on CrossCodeEval (Ding et al., 2023). (3) *Metadata retention*: AST-based chunks more faithfully capture metadata at the file, class, and function levels, enhancing context matching in hybrid code+natural language tasks, e.g., up to 2.7 points gain on SWE-bench (Jimenez et al., 2024), which focuses on resolving GitHub issues.

2 cAST

We focus on the first stage of the RAG pipeline: *chunking*. In this step, source code is parsed into semantically meaningful units (such as functions or classes) while preserving the structure of the code. These units are then grouped into coherent chunks, which serve as the retrievable context that can be obtained by a subsequent *retriever* and used to prompt a *language model*.

Design Goal. Our design for cAST pursues four aligned goals: (1) *syntactic integrity*—whenever possible, chunk boundaries should align with complete syntactic units instead of splitting them; (2) *high information density*—each chunk is packed up to, but not beyond, a fixed size budget to maximize content utility; (3) *language invariance*—the algorithm employs no language-specific heuristics so it works unchanged across diverse programming languages and code-related tasks; and (4) *plug-and-play compatibility*—concatenating the chunks must reproduce the original file verbatim, enabling seamless drop-in replacement within existing RAG pipelines.

AST Parsing. To support syntax-aware chunking, we leverage the *Abstract Syntax Tree (AST)* representation of code. An AST is a tree-structured abstraction that captures the syntactic structure of source code in a way that is both hierarchical and semantically rich. Rather than treating code as plain text, AST encodes language constructs—like functions, classes, loops, and conditionals—as distinct nodes in a structured parse tree. This enables us to identify meaningful code boundaries with precision, ensuring that chunking respects the underlying syntax. Since ASTs are widely supported across languages, this approach also enhances the language-invariance and portability of our method. Our work uses the `tree-sitter` library (Tree-sitter, 2025) for the AST tree parsing.

AST-based Recursive Chunking. With the AST tree at hand, we use a recursive, split-then-merge algorithm for converting tree structures into chunks, as shown in Figure 2. To retain as much syntactic information as possible, we first traverse the tree in a top-down manner, to fit those large AST nodes into a single chunk whenever possible. For those nodes that must be split due to exceeding the chunk size limit, to avoid too many overly small chunks, we further perform a greedy merging step, combining adjacent small sibling nodes into one chunk, to maximize the per-chunk information density. The detailed process is also described in Alg. 1.

Chunk size metric. Choosing an appropriate budget for each chunk is nontrivial: two segments of equal line count can carry wildly different amounts of code, and AST-aligned chunks naturally vary in their physical span (e.g., a single import line versus an entire class body). So unlike prior work (Wang et al., 2024), we measure chunk size by the number of non-whitespace characters rather than by lines. This keeps chunks text-dense and comparable across diverse files, languages, and coding styles, ensuring that our budget reflects actual content rather than incidental formatting.

3 Experiments

We evaluate cAST with various top retrieval and generation models in various code task settings. We present results of selected end-to-end RACG pipelines (retriever + LM) in Section 3.2 and full tables in the Appendix (5, 6, 7, 8).

3.1 Experiment Settings

Datasets. We evaluate cAST on various software engineering (SE) tasks using three benchmarks:

- RepoEval (Zhang et al., 2023b): Code completion tasks with long intra-file contexts;
- CrossCodeEval (Ding et al., 2023): Multi-language queries requiring cross-file reasoning;
- SWE-bench (Jimenez et al., 2024): General SE tasks involving code patch generation. We use

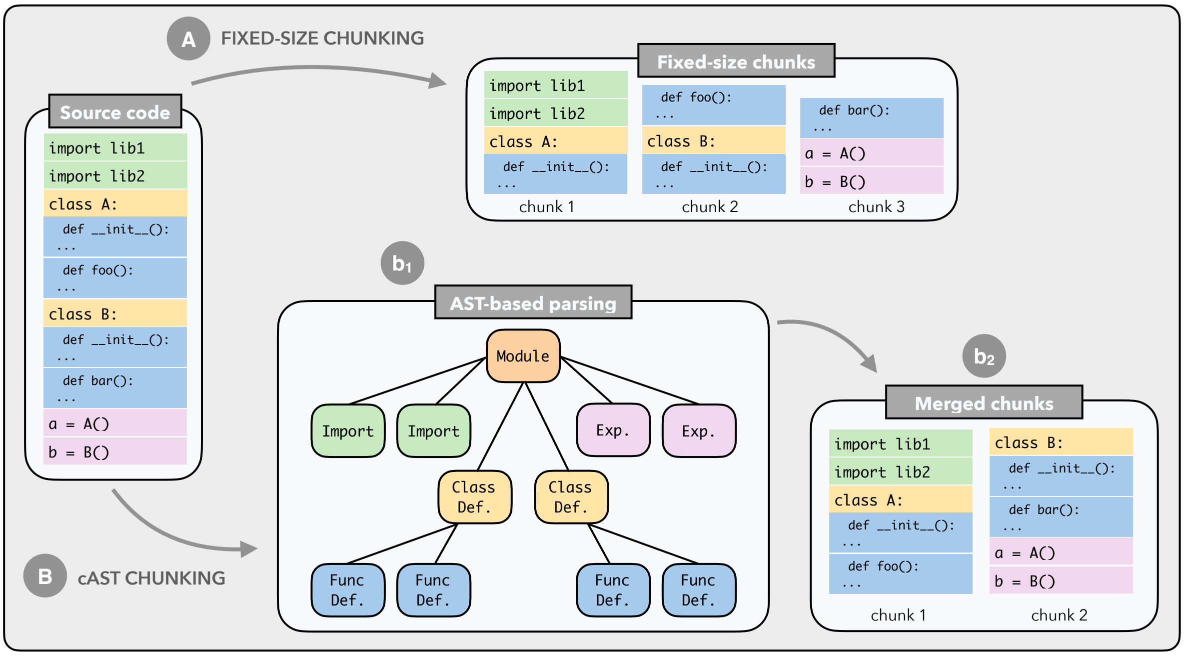

Figure 2: Comparison of fixed-size chunking vs. CAST. For CAST, we first parse the document into a tree of AST nodes. Then, starting from the first level, we greedily merge AST nodes into chunks. If adding a node would exceed the chunk size limit, we recursively break it into smaller nodes. The output of CAST is a list of chunks where each chunk contains a list of AST nodes.

the SWE-bench Lite variant (bench Lite, 2024), a 300-problem subset where each issue is solvable by editing a single file.

Metrics. For retrieval performance, we report three common metrics: nDCG, Precision and Recall, with $k = 5$. Notably, since retrieval scores from different corpus distributions are not directly comparable, we implement a score mapping technique to align AST-based retrieval scores with those of the baseline, with details in Appendix A.2.

As for generation, we use Pass@k (Chen et al., 2021) for execution-based datasets and match-based metrics for the others, following prior work (Wang et al., 2024; Ding et al., 2023). Specifically, we report the canonical Pass@1 score for RepoEval and SWE-bench. Additionally, we record the Pass@8 score for SWE-bench by sampling multiple responses with high temperature following Agentless (Xia et al., 2024a) to examine the robustness of CAST. For CrossCodeEval, we report exact match (EM), edit similarity (ES), and other identifier match metrics in the original work.

Retrieval and Generation Models. We adopt various kinds of retrievers, including general-text dense retrievers: `BGE-base` (Xiao et al., 2023) and `GIST-base` (Solatorio, 2024); and code-specific retriever: `Codesage-small-v2` (Zhang et al., 2024), following CodeRAG-Bench (Wang et al., 2024).

Similarly, for generations, we include two code-specific LMs: `StarCoder2-7B` (Lozhkov et al., 2024), `CodeLlama-7B-Python` (Roziere et al., 2023); and two general-purpose LMs (`claude-3.7-sonnet`, `gemini-2.5-pro-0325`), as both represent the state-of-the-art in coding.

Further details of our experimental setup are introduced in Appendix A.1.

3.2 CAST Results and Analysis

Table 1 presents the end-to-end RACG results with selected retrievers (BGE-base, GIST-base, Codesgae-small-v2) on the three datasets. The results highlight several key observations:

Retrieval. CAST 's structure-aware chunking steadily improves retrieval performance across datasets and retrievers. Specifically, all models show gains of 1.2–3.3 points in Precision and 1.8–4.3 in Recall on code-to-code retrieval (RepoEval), and 0.5–1.4 in Precision and 0.7–1.1 in Recall on the more challenging NL-to-code retrieval (SWE-Bench). These improvements suggest that aligning chunks with abstract syntax boundaries helps diverse retrievers surface semantically coherent code fragments, supplying richer and more accurate evidence for downstream tasks.

	Metric (Model)	CAST chunking			Fixed-size chunking		
		BGE	GIST	CodeSage	BGE	GIST	CodeSage
RepoEval							
R	nDCG	71.1	75.9	85.1	71.3	74.2	83.0
R	Precision	34.9	38.1	44.1	32.8	34.8	42.9
R	Recall	69.8	75.0	83.9	67.4	70.7	82.1
G	Pass@1 (StarCoder2)	51.7	57.9	73.2	47.5	51.2	67.6
G	Pass@1 (CodeLlama)	49.6	56.6	72.1	45.6	51.5	66.5
SWE-Bench							
R	nDCG	44.0	44.4	43.1	42.4	43.1	42.6
R	Precision	39.7	39.1	38.8	38.3	38.6	37.5
R	Recall	18.4	18.5	18.3	17.3	17.8	17.5
G	Pass@1 (Claude)	16.3	15.0	16.7	13.7	14.7	14.0
G	Pass@8 (Gemini)	35.3	33.7	32.7	32.3	33.0	31 0
CrossCodeEval							
R	Identifier Match (EM)	34.7	34.0	39.9	32.0	33.5	36.3
G	EM (StarCoder2)	23.8	23.4	29.1	21.2	23.0	24.8
G	ES (StarCoder2)	72.2	71.9	74.3	71.0	71.7	73.1

Table 1: **R**etrieval and **G**eneration Performances across three benchmarks, using different retrieval models (BGE, GIST, CodeSage) and different LMs (full model names in §3.1).

Generation. CAST benefits both intra-file and cross-file code completion. Notably, gains are most pronounced when the RACG pipeline employs code-specific retrievers, implying that the structurally aligned chunks deliver fuller context to both the specialized retriever and the generation model, which in turn facilitates more accurate context retrieval and coherent code synthesis. On NL-to-code generation, we observe remarkable gains with BGE-base and CodeSage retrievers under one and multiple rounds of sampling.

Correlation between retrieval and generation performance. Among the three retrieval metrics we use, we notice that higher precision tends to convert into better generation performance, aligning with conclusions from prior work (Zhao et al., 2024). This suggests that ensuring the top-k context is highly relevant reduces noise and enables the language model to concentrate on concise, accurate evidence, thereby boosting answer fidelity (Fang et al., 2024; Salemi and Zamani, 2024).

By contrast, recall-oriented metrics and nDCG correlate only weakly with downstream quality—once the necessary evidence appears in the retrieved set, adding lower-ranked chunks yields diminishing returns or can even hurt performance by introducing distractors.

4 Ablations

Necessity of merging. The motivation for introducing merging in our algorithm is to maximize the information density of each chunk. Under a split-only approach, small AST nodes, such as import statements and variable assignments, generate an excessive number of chunks, which unnecessarily enlarges the index and degrades retrieval performance. These fine-grained chunks also contain limited context, making them less effective for downstream tasks, as shown in Table 2. Across all retrievers, we find that both retrieval and generation performance decline under the split-only strategy.

	Metric (Model)	Split-then-merge (CAST)			Split-only		
		BGE	GIST	CodeSage	BGE	GIST	CodeSage
R	nDCG	71.1	75.9	85.1	53.5	59.1	66.1
G	Pass@1 (StarCoder2)	51.7	57.9	73.2	48.3	45.0	65.4
G	Pass@1 (CodeLlama)	49.6	56.6	72.1	47.2	48.5	58.4

Table 2: Ablation study comparing performance metrics for Split-then-merge (CAST) and Split-only methodologies across different models.

Pipeline (R + G)	Context length (tokens)		
	3500	4000	8000
BGE + StarCoder2	46.9	51.7	51.7
GIST + StarCoder2	57.1	57.9	58.2
CodeSage + StarCoder2	70.5	73.2	69.2

Table 3: Ablation study evaluating the impact of different context lengths on the overall performance of several retrieval and generation pipelines.

Selection of context length. In our experiments, we set *max_context_length* = 4000, which roughly corresponds to the top five chunks. A comparison of different context lengths is shown in Table 3. We observe that doubling the context length does not necessarily improve generation, whereas a modest reduction in context length can lead to performance degradation, likely due to chunk truncation.

Selection of maximum chunk size. We set *max_chunk_size* = 2000 in our experiments, as the resulting chunks exhibit similar statistics (e.g., line counts and token counts) to the fixed-size chunking baseline. A sensitivity analysis of *max_chunk_size* is presented in Table 4. We observe that retrieval and generation performance peak when *max_chunk_size* is between 2000 and 2500 characters. Additionally, generation performance also depends on *max_context_length*, as shown in the previous analysis. When context length allows, larger chunks can provide more information, while smaller chunks help mitigate the risk of truncation.

	Metric (Model)	Maximum chunk size				
		1000	1500	2000	2500	3000
R	nDCG	69.0	68.4	71.1	72.3	69.4
G	Pass@1 (StarCoder2)	43.4	45.8	51.7	50.1	51.2

Table 4: Ablation study of maximum chunk size effects on retrieval and generation performance.

5 Related Work

Structure-aware modeling in code tasks. Early work showed clear benefits from feeding explicit syntax to models: TranX (grammar-guided decoding) and path-based encoders code2vec/code2seq leveraged AST productions or paths to outperform token-only baselines in NL-to-code and summarization (Yin and Neubig, 2018; Alon et al., 2019b,a). Transformer-era studies refined this idea. GraphCodeBERT (Guo et al., 2021) and the Code Transformer (Zügner et al., 2021) inject data-flow edges or AST distances, while CODEDISEN (Zhang et al., 2021) disentangles syntax from semantics for cross-language transfer. More recent models layer structure-aware objectives onto large LMs: TypeT5 (Wei et al., 2023) adds static-analysis context for type inference, and AST-T5 (Gong et al., 2024) and StructCoder (Tipirneni et al., 2024) mask or generate subtrees to boost transpilation and Java-Python translation.

Although modern LLMs can often internalize such structure from raw tokens, these results indicate that explicit syntax still provides measurable gains—especially in preprocessing steps like chunking, where respecting function or class boundaries directly controls what the model sees. In light of the importance of structure awareness in the above literature, we propose to leverage the tree structure of code snippets to improve chunking.

Retrieval-augmented code generation. Successful code RAG hinges on pairing high-quality retrievers with generation frameworks that can effectively leverage the fetched context. General-purpose systems—RAG (Lewis et al., 2020), FiD (Izacard and Grave, 2021), and RePlug (Shi et al., 2023)—demonstrate that feeding high-recall evidence to a language model markedly improves factuality. In the software-engineering domain, CodeRAG-Bench (Wang et al., 2024) confirms these gains on repository-level tasks while revealing that lexical-matching retrievers often miss relevant code, motivating code-specific retrieval models. State-of-the-art code retrievers such as CodeBERT (Feng et al., 2020), UniXcoder (Guo et al., 2022), and CodeRetriever (Li et al., 2022) learn joint code–text or code–code embeddings and consistently surpass generic dense models in code search and question answering. Most pipelines still inherit fixed line-based chunking from natural-language RAG. Our work shows that respecting syntactic units with AST-aware chunks further enhances these retrieval-generation loops.

Most relevantly, CodeCRAG (Du et al., 2025) utilizes the graphical view of code flow to improve the overall LLM code generation pipeline. Shen et al. (2024); Xia et al. (2024b); Song et al. (2024) propose to compute code similarity based on the graph structure of code. In our work, we conduct a fine-grained study on one important block of code RAG workflow: chunking.

6 Conclusion and Discussion

In this work, we present cAST as a simple and effective chunking strategy for retrieval-augmented code generation. Through the structural awareness brought by AST, we are allowed to maintain syntactic integrity and high information density during chunking. Extensive experiments on various retrievers, LLM generators, and code generation tasks, validate the gain from cAST over the commonly used fixed-size chunking strategy on both retrieval and RAG tasks.

By maintaining the original RAG pipeline, for the code agent practitioner, cAST could be used as a simple plug-and-play tool to provide informative and formatted chunks for later stage agent use. For code RAG benchmark developers, cAST could serve as additional resources and an effective alternative or complementary retrieval unit.

Limitations

Contextual Awareness. In our experiments, for a fair comparison, we maintain the original retrieval-augmented code generation pipeline to parse code snippets into self-contained chunks, without explicit contextual awareness from higher chunking units in the AST. However, as shown in (Sarthi et al., 2024; Cai et al., 2024), in textual RAG, including multi-level information in the tree structures can improve the retrieval performance, which can also potentially benefit code retrieval with the natural structures that can be extracted with our AST framework.

Multi-view of the code. In this work, we mainly explore chunking with pure code files. However, each code snippet can potentially have multiple views, e.g., the input-output elicitation in the comments, natural language descriptions, pseudo code, and etc. Each of these views can emphasize different facets of the very code snippet. Previous work shows that including multiple views helps model math reasoning (Liang et al., 2023). Similarly, instead of pure AST-based chunking on code snippets, including different chunk candidates from different views can potentially relieve the code completeness reliance of our cAST.

Inner Execution Dynamics. In this work, we focus on introducing the structural awareness to retrieval augmented generation with AST, as a static analysis of the code semantics. However, the execution trace (Ni et al., 2024), type inference (Wei et al., 2023), and compilation (Cummins et al., 2024) can potentially lead to a deep understanding of the variable dynamics. Introducing the awareness of such in-depth query analysis can help augment our cAST with per-query adaptiveness.

Acknowledgments

The authors thank Jamie Callan, Fernando Diaz, Graham Neubig, Daniel Fried, and Pengcheng Yin for their insights into design and evaluation choices. The authors also thank the constructive discussions with colleagues from CMU WInE Lab and Augment Code. Xinran Zhao is supported by the ONR Award N000142312840. This work is supported by the OpenAI Research Credit program, the Amazon AI Research Gift Fund, and the Gemma Academic Program GCP Credit Award.

References

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations (ICLR)*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: Learning distributed representations of code. In *Proceedings of the ACM/IEEE Symposium on Principles of Programming Languages (POPL)*.

SWE bench Lite. 2024. Swe-bench lite. https://www.swebench.com/lite.html.

Bernd Bohnet, Vinh Q. Tran, Pat Verga, Roee Aharoni, Daniel Andor, Livio Baldini Soares, Massimiliano Ciaramita, Jacob Eisenstein, Kuzman Ganchev, Jonathan Herzig, Kai Hui, Tom Kwiatkowski, Ji Ma, Jianmo Ni, Lierni Sestorain Saralegui, Tal Schuster, William W. Cohen, Michael Collins, Dipanjan Das, and 3 others. 2023. Attributed question answering: Evaluation and modeling for attributed large language models. *Preprint*, arXiv:2212.08037.

Fengyu Cai, Xinran Zhao, Tong Chen, Sihao Chen, Hongming Zhang, Iryna Gurevych, and Heinz Koeppl. 2024. MixGR: Enhancing retriever generalization for scientific domain through complementary granularity. *Preprint*, arXiv:2407.10691.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Tong Chen, Hongwei Wang, Sihao Chen, Wenhao Yu, Kaixin Ma, Xinran Zhao, Hongming Zhang, and Dong Yu. 2023. Dense x retrieval: What retrieval granularity should we use? *arXiv preprint arXiv:2312.06648*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *Preprint*, arXiv:2407.02524.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Kounianhua Du, Jizheng Chen, Renting Rui, Huacan Chai, Lingyue Fu, Wei Xia, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2025. Codegrag: Bridging the gap between natural language and programming language via graphical retrieval augmented generation. *Preprint*, arXiv:2405.02355.

Feiteng Fang, Yuelin Bai, Shiwen Ni, Min Yang, Xiaojun Chen, and Ruifeng Xu. 2024. Enhancing noise robustness of retrieval-augmented language models with adaptive adversarial training. *Preprint*, arXiv:2405.20978.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 1536–1547.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: Structure-aware pretraining for code generation and understanding. arXiv preprint arXiv:2401.03003.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, and 1 others. 2023. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. *arXiv preprint arXiv:2310.19923*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 7212–7225.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR.

Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, and 7 others. 2020. Array programming with numpy. *Nature*, 585:357–362.

John D Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95.

Gautier Izacard and Edouard Grave. 2021. Leveraging passage retrieval with generative models for open domain question answering. In *International Conference on Learning Representations (ICLR)*.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane A. Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. Few-shot learning with retrieval augmented language models. *ArXiv*, abs/2208.03299.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 9459–9474.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. CodeRetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2898–2910, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Zhenwen Liang, Dian Yu, Xiaoman Pan, Wenlin Yao, Qingkai Zeng, Xiangliang Zhang, and Dong Yu. 2023. Mint: Boosting generalization in mathematical reasoning via multi-view fine-tuning. *Preprint*, arXiv:2307.07951.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. 2024. An empirical study on llm-based agents for automated bug fixing. *Preprint*, arXiv:2411.10213.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *Preprint*, arXiv:2404.14662.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, and 2 others. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances* *in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Alireza Salemi and Hamed Zamani. 2024. Evaluating retrieval quality in retrieval-augmented generation. *Preprint*, arXiv:2404.13781.

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. 2024. Raptor: Recursive abstractive processing for tree-organized retrieval. In *International Conference on Learning Representations (ICLR)*.

Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kaustubh Vyas, Yuanyi Ji, and Jeff Z Pan. 2024. Improving retrieval-augmented text-to-sql with ast-based ranking and schema pruning. *arXiv preprint arXiv:2407.03227*.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Mike Lewis, Luke Zettlemoyer, and Wentau Yih. 2023. REPLUG: Retrieval-augmented black-box language models. arXiv preprint arXiv:2301.12652.

Aivin V. Solatorio. 2024. Gistembed: Guided in-sample selection of training negatives for text embedding fine-tuning.

Yewei Song, Cedric Lothritz, Xunzhu Tang, Tegawendé Bissyandé, and Jacques Klein. 2024. Revisiting code similarity evaluation with abstract syntax tree edit distance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 38–46, Bangkok, Thailand. Association for Computational Linguistics.

Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. Structcoder: Structure-aware transformer for code generation. *ACM Transactions on Knowledge Discovery from Data*, 18(3):70:1–70:20.

Tree-sitter. 2025. Tree-sitter documentation. `https://tree-sitter.github.io/tree-sitter/`. Accessed: May 11, 2025.

Zora Zhiruo Wang, Akari Asai, Xinyan Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. CodeRAG-Bench: Can retrieval augment code generation? arXiv preprint arXiv:2406.14497.

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq type inference using static analysis. In *International Conference on Learning Representations (ICLR)*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and 1 others. 2019. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv preprint*, abs/1910.03771.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024a. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

Yu Xia, Tian Liang, Weihuan Min, and Li Kuang. 2024b. Improving ast-level code completion with graph retrieval and multi-field attention. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 125–136.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding. *arXiv*.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (System Demonstrations)*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Code representation learning at scale. *arXiv preprint arXiv:2402.01935*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. RepoCoder: Repository-level code completion through iterative retrieval and generation. pages 2471–2484. Association for Computational Linguistics.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023b. 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, Singapore. Association for Computational Linguistics.

Jingfeng Zhang, Haiwen Hong, Yin Zhang, Yao Wan, Ye Liu, and Yulei Sui. 2021. Disentangled code representation learning for multiple programming languages. In *Findings of the Association for Computational Linguistics: ACL–IJCNLP*, pages 4454–4466.

Xinran Zhao, Tong Chen, Sihao Chen, Hongming Zhang, and Tongshuang Wu. 2024. Beyond relevance: Evaluate and improve retrievers on perspective awareness. *Preprint*, arXiv:2405.02714.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*.

A Appendix

A.1 Implementation Details

For Gemini and Claude models, we use the official API service. For other open-sourced models, we use locally served models on nodes with 8 Nvidia A100 (40G) GPU and 8 Nvidia A6000 (40G) GPUs with CUDA 12 installed. Our inference structure is built upon vLLM (Kwon et al., 2023).

For fair comparison of chunks with varying sizes, instead of using top-k chunks directly, We use `max_context_length` to sequentially include retrieved chunks up to a threshold, truncating the final chunk if needed. We set the limit to 4000 for RepoEval and SWE-Bench, and extend it to 10000 for CrossCodeEval to test cross-file retrieval. [1] For generation, we adopt different settings based on evaluation metrics based on prior work (Wang et al., 2024; Li et al., 2023; Xia et al., 2024a): We use $t = 0.2$, $top_p = 0.95$, and 1 sample for Pass@1; $t = 0.8$ and 8 samples for Pass@8.

A.2 Metric Score Mapping Details

In Section 3.1, we denote the distributional incomparability across corpses. We implement a score mapping technique to align AST-based retrieval scores over baselines.

Specifically, similar to (Chen et al., 2023), we assign each line of code a score inherited from its corresponding AST chunk. These line-level scores are then aggregated to recompute the scores of baseline chunks, allowing us to rerank them and estimate AST-based retrieval performance within the baseline framework.

A.3 AST-based Chunking Algorithm Details

In the main paper, we provide textual descriptions of our algorithm. Here, we present the pseudo code of our implementation in Alg. 1.

A.4 Extended Experiment Results

In the main paper, we show concise results from our experiment to demonstrate a clear contribution. We further include detailed results from our settings here. In Table 5, we present the retrieval performance with various metrics and retrievers on RepoEval and SWE-bench. In Table 7, we present the RAG performance on SWE-Bench with various retrievers (large language models) and generators. In Table 6, we present the RAG performance on RepoEval with various retrievers and generators. In Table 8, we show the RAG performance with various retrievers on CCEval across different programming languages.s

These tables show similar conclusions with our findings in the main paper, where cAST consistently performs better than fixed-size line-based chunking with syntactic integrity and high information density.

[1] We use default tokenizers for open-weighted models, and `cl100k_base` for API models.

Algorithm 1 AST-based Chunking Algorithm

```
MAX_SIZE ← maximum chunk size

function CHUNKCODE(code)
    tree ← PARSEAST(code)
    if GETSIZE(code) ≤ MAX_SIZE then
        return [tree]
    else
        return CHUNKNODES(tree.children)
    end if
end function

function CHUNKNODES(nodes)
    chunks ← [ ], chunk ← [ ], size ← 0
    for node in nodes do
        s ← GETSIZE(node)
        if (chunk = [ ] and s > MAX_SIZE) or
          (size + s > MAX_SIZE) then
            if chunk ≠ [ ] then
                chunks.append(chunk)
                chunk, size ← [ ], 0
            end if
            if s > MAX_SIZE then
                subchunks ← CHUNKNODES(node.children)
                chunks.extend(subchunks)
                continue
            end if
        else
            chunk.append(node); size ← size + s
        end if
    end for
    if chunk ≠ [ ] then
        chunks.append(chunk)
    end if
    return chunks
end function
```

A.5 Performance differences across different programming languages

A key limitation of fixed-size, line-based chunking is its poor generalizability across programming languages. Language-specific syntax means a line limit tuned for one language over- or under-segments another, leading to uneven information density and degraded retrieval and generation quality. In contrast, cAST uses structure-aware segmentation based on abstract-syntax units common across languages, mitigating these issues.

Table 8 reports results with the Codesage-small-v2 + Starcoder2-7B pipeline. Though both meth-

Method	CAST						Fixed-size					
	nDCG@5	nDCG@10	P@5	P@10	Recall@5	Recall@10	nDCG@5	nDCG@10	P@5	P@10	Recall@5	Recall@10
RepoEval												
BGE-base	71.1	74.7	34.9	20.4	69.8	77.6	71.3	74.6	32.8	19.1	67.4	74.1
BGE-large	72.2	75.4	34.9	20.2	69.6	76.3	71.1	73.9	31.3	18.1	64.9	70.6
GIST-base	75.9	78.5	38.1	21.2	75.0	80.5	74.2	78.0	34.8	20.6	70.7	78.5
GIST-large	78.9	81.9	38.8	22.0	76.6	82.8	75.1	79.5	34.8	21.1	71.1	80.2
Codesage-small-v2	85.1	88.8	44.1	25.3	83.9	91.0	83.0	86.4	42.9	24.5	82.1	89.1
Jina-v2-code	87.1	90.5	47.9	27.1	87.9	94.7	86.8	90.9	46.3	26.7	84.9	92.9
SWE-bench												
BGE-base	44.0	41.5	39.7	32.5	18.4	26.8	42.4	39.5	38.3	31.2	17.3	24.4
BGE-large	42.2	40.4	37.7	31.6	17.5	26.1	42.8	39.9	38.3	31.2	17.0	24.6
GIST-base	44.4	42.5	39.1	32.9	18.5	27.6	43.1	40.6	38.6	31.8	17.8	25.9
GIST-large	44.0	41.9	39.5	33.1	18.5	27.0	43.5	41.7	39.2	33.2	18.0	26.5
Codesage-small-v2	43.1	41.4	38.8	32.8	18.3	26.4	42.6	40.0	37.5	31.0	17.5	24.7

Table 5: Retrieval performance (nDCG, Precision, Recall@{5,10}) on RepoEval and SWE-bench.

Method	CAST		Fixed-size	
	StarCoder2	CodeLlama	StarCoder2	CodeLlama
BGE-base	51.7	49.6	47.5	45.6
BGE-large	48.8	50.9	45.8	49.9
GIST-base	57.9	56.6	51.2	51.5
GIST-large	61.7	60.3	59.2	55.5
Codesage-small-v2	73.2	72.1	67.6	66.5
Jina-v2-code	80.7	75.9	75.1	75.1

Table 6: RAG performance (Pass@1) on RepoEval with various retrievers.

ods use fixed chunk lengths, performance variation across languages is notably higher for the baseline. Averaged over four languages, CAST improves EM by 2.9 on code and 3.0 on identifier, with the largest gains on TypeScript—the noisiest language. These consistent gains highlight the value of respecting syntax when handling multilingual code.

The performance differences across different languages with different chunking strategies, as well as RAG design choices, can form an interesting future line of work.

A.6 Ethical Statements

We foresee no ethical concerns or potential risks in our work. All of the retrieval models, code generators, and datasets are open-sourced or with public APIs, as shown in Section 3. The LLMs we applied in the experiments are also publicly available. Given our context, the outputs of LLMs (code snippets) are unlikely to contain harmful and dangerous information. All the code is executed in sandboxes, with no threat to the public internet. The natural language part of our experiments is mainly on English. Multiple programming languages are included: Python, Java, C#, and TypeScript.

Our code is open source and available at `https://github.com/yilinjz/astchunk`.

A.7 Licenses of scientific artifacts

We conclude the licenses of the scientific artifacts we used in Table 9. All of our usage for scientific discovery follows the original purpose of the artifacts.

Method	CAST		Fixed-size	
	Claude-3.7-Sonnet	Gemini-2.5-pro	Claude-3.7-Sonnet	Gemini-2.5-pro
BGE-base	16.3	35.3	13.7	32.3
BGE-large	13.3	30.3	14.6	33.7
GIST-base	15.0	33.7	14.7	33.0
GIST-large	15.3	31.0	13.0	33.0
Codesage-small-v2	16.7	32.7	14.0	31.0

Table 7: RAG performance (Claude w/ Pass@1 & Gemini w/ Pass@8) on SWE-bench.

Method	CAST				Fixed-size			
	EM (code)	ES (code)	EM (id)	F1 (id)	EM (code)	ES (code)	EM (id)	F1 (id)
BGE-base + Starcoder2-7B								
Python	23.8	72.2	34.7	63.8	21.2	71.0	32.0	62.1
Java	27.8	70.9	37.5	63.8	27.3	71.6	37.1	64.1
C#	26.9	73.5	32.0	56.4	23.9	71.8	28.3	53.8
TypeScript	13.4	49.6	19.5	43.6	11.4	46.0	17.4	40.2
GIST-base + Starcoder2-7B								
Python	23.4	71.9	34.0	63.7	23.0	71.7	33.5	63.3
Java	28.0	71.2	37.7	64.3	27.0	71.3	36.8	63.7
C#	26.6	73.2	31.2	56.0	24.3	72.5	28.7	54.3
TypeScript	13.0	49.3	19.7	43.9	11.2	46.1	17.2	40.2
Codesage-small-v2 + Starcoder2-7B								
Python	29.1	74.3	39.9	67.6	24.8	73.1	36.3	65.7
Java	30.9	72.2	41.2	66.1	28.1	71.5	38.3	64.6
C#	28.3	74.2	33.4	58.2	25.5	72.4	29.9	54.9
TypeScript	13.7	49.1	19.6	43.5	11.9	46.0	17.7	40.6

Table 8: RAG performance (Code Match & Identifier Match) on CrossCodeEval.

Artifacts/Packages	Citation	Link	License
RepoEval	(Zhang et al., 2023b)	`https://github.com/irgroup/repro_eval`	MIT License
SWE-bench	(Jimenez et al., 2024)	`https://github.com/SWE-bench/SWE-bench`	MIT License
CrossCodeEval	(Ding et al., 2023)	`https://github.com/amazon-science/cceval`	Apache License 2.0
PyTorch	(Paszke et al., 2019)	`https://pytorch.org/`	BSD-3 License
transformers	(Wolf et al., 2019)	`https://huggingface.co/transformers/v2.11.0/index.html`	Apache License 2.0
numpy	(Harris et al., 2020)	`https://numpy.org/`	BSD License
matplotlib	(Hunter, 2007)	`https://matplotlib.org/`	BSD compatible License
vllm	(Kwon et al., 2023)	`https://github.com/vllm-project/vllm`	Apache License 2.0
BGE	(Xiao et al., 2023)	`https://huggingface.co/BAAI/bge-large-en`	MIT license
GIST	(Solatorio, 2024)	`https://huggingface.co/avsolatorio/GIST-Embedding-v0`	MIT license
CodeSage	(Zhang et al., 2024)	`https://huggingface.co/codesage/codesage-small-v2`	Apache License 2.0
Jina-v2-Code	(Günther et al., 2023)	`https://huggingface.co/jinaai/jina-embeddings-v2-base-code`	Apache License 2.0
StarCoder2	(Lozhkov et al., 2024)	`https://huggingface.co/bigcode/starcoder2-7b`	LICENSE
CodeLlama	(Roziere et al., 2023)	`https://huggingface.co/codellama/CodeLlama-7b-hf`	LICENSE

Table 9: Details of datasets, major packages, and existing models we use. The curated datasets and our code/software are under the MIT License.