MLCPD: A Unified Multi-Language Code Parsing Dataset with Universal AST Schema

Jugal Gajjar[*]
Department of Computer Science
The George Washington University
Washington, D.C. 20052
jugal.gajjar@gwu.edu

Kamalasankari Subramaniakuppusamy
Department of Computer Science
The George Washington University
Washington, D.C. 20052
kamalasankaris@gwu.edu

Abstract

We introduce the MultiLang Code Parser Dataset (MLCPD), a large-scale, language-agnostic dataset unifying syntactic and structural representations of code across ten major programming languages. MLCPD contains over seven million parsed source files normalized under our proposed *universal Abstract Syntax Tree (AST)* schema, enabling consistent cross-language reasoning, structural learning, and multilingual software analysis. Unlike existing corpora that focus purely on token-level code or isolated parsers, MLCPD provides both hierarchical tree representations and rich metadata for every file, ensuring lossless syntactic coverage and structural uniformity. Each entry includes a normalized schema, language-level metadata, and abstracted node semantics stored in Parquet format for scalable retrieval. Empirical analyses reveal strong cross-language structural regularities—demonstrating that syntactic graphs from languages as diverse as Python, Java, and Go can be aligned under a shared schema. We release the dataset publicly on Hugging Face[2] and the accompanying codebase on GitHub[3], which includes complete pipelines for dataset reproduction, grammar compilation, and a visualization tool for exploring the unified AST across languages. Together, these resources establish MLCPD as an open, reproducible foundation for future research in cross-language representation learning and program analysis.

1 Introduction

Understanding and reasoning over program structure across languages remains a central challenge in software intelligence. Despite recent advances in multilingual models such as CodeBERT (6) and GraphCodeBERT (7), there exists a persistent disconnect between language-agnostic modeling goals and the lack of structurally consistent, large-scale datasets. Most existing corpora—such as The Stack (20), StarCoder (21), and CodeSearchNet (25)—focus on token-level or natural language alignments, overlooking fine-grained syntactic nodes and semantic categories that form the foundation for symbolic reasoning, program translation, and vulnerability detection.

The MLCPD dataset was created to address this gap by introducing a unified structural representation of code across ten major programming languages: *C, C++, C#, Go, Java, JavaScript, Python, Ruby, Scala, and TypeScript*. By integrating over seven million code files, MLCPD bridges syntax and semantics under a universal Abstract Syntax Tree (AST) schema. Each file is represented as a hierarchical JSON object preserving all syntactic relations, augmented with node-type statistics and semantic mappings for cross-language comparability. This uniform structure enables researchers to

[*]Corresponding author: 812jugalgajjar@gmail.com

[2]https://huggingface.co/datasets/jugalgajjar/MultiLang-Code-Parser-Dataset

[3]https://github.com/JugalGajjar/MultiLang-Code-Parser-Dataset

conduct language-agnostic analyses, trace syntactic parallels, and train models that reason across paradigms with minimal preprocessing overhead.

Our motivation stems from three key observations: (1) Syntax is a universal interface for reasoning about code, regardless of language surface form. (2) Modern LLM-based code models still lack interpretable and structurally aligned supervision signals. (3) A unified dataset that preserves language-specific nuances while enforcing schema normalization can empower research in cross-language program understanding, translation, and vulnerability analysis.

Beyond serving as a benchmark corpus, MLCPD aims to redefine how multilingual source code is represented and analyzed. Its universal schema provides a bridge between symbolic program representations and neural models, facilitating hybrid reasoning that leverages both structure and semantics. The dataset's alignment across ten languages offers an unprecedented opportunity to study structural transfer, syntactic entropy, and multilingual generalization—laying the groundwork for the next generation of explainable, cross-language software intelligence systems.

2 Related Work

Research in multilingual program analysis has evolved considerably over the past two decades, transitioning from rule-based static analyzers to large-scale structural and representation-learning frameworks. Early efforts, such as cross-language refactoring systems (2) and dependency visualization tools (3), primarily targeted symbol extraction, interface linkage, and modular comprehension within multi-language codebases. These early systems provided valuable insight into heterogeneous software but were restricted to limited language pairs and lacked extensibility across modern ecosystems.

Subsequent frameworks, most notably LiSA (13) and LARA (12), introduced language-independent static analyzers by defining common intermediate representations (IRs) that could capture shared semantic structures. Universal intermediate representations (IRs) such as LLVM IR(32) and Java bytecode(33) provide language-agnostic abstractions at the compilation level. While these approaches advanced portability, they often relied on handcrafted grammars and deterministic rules, resulting in brittle scalability when applied to large, diverse corpora. Later developments such as AXA (14) and PolyCruise (19) extended this idea to cross-language static and dynamic analysis by integrating multiple monolingual analyzers into unified pipelines. However, these frameworks remain tool-specific and execution-oriented rather than dataset-centric, offering limited value for statistical learning or code representation modeling.

In parallel, the introduction of permissively licensed corpora such as The Stack (20) and StarCoder (21) catalyzed the growth of language models for code generation, retrieval, and summarization. Despite their scale—covering terabytes of multilingual source code—these resources emphasize token-level data and metadata rather than syntactic or structural representations (9). Consequently, while they have enabled progress in large language model (LLM) training, they provide little insight into structural patterns or cross-language regularities (24). Smaller datasets like CodeSearchNet (25) and Py150 (22) introduced limited AST supervision but remain narrow in scope, typically covering only one or two languages with incomplete structural annotations.

At the representation-learning level, recent models such as CodeBERT (6), GraphCodeBERT (7), and AST-T5 (28) have demonstrated the utility of incorporating structural signals through token–graph hybrids or attention over AST nodes. Yet, these models depend on heterogeneous parsing resources, each introducing language-specific biases and inconsistent node semantics. Without a standardized schema, embeddings learned from one language cannot be reliably transferred to another. This fragmentation reflects a core limitation across modern code intelligence research: the absence of a unified, lossless, and cross-language structural foundation.

The MultiLang Code Parser Dataset (MLCPD) directly addresses this gap. Unlike prior corpora that focus on textual data or tool-oriented analysis frameworks, MLCPD introduces a scalable, dataset-driven approach to structural unification. It integrates over seven million files across ten languages, each represented under a single, formally defined *universal Abstract Syntax Tree (AST)* schema that preserves syntactic fidelity while enabling cross-language comparability. By decoupling parsing from modeling and by providing standardized, schema-validated structures, MLCPD lays the groundwork for reproducible cross-language learning and fine-grained structural reasoning at scale.

Table 1 summarizes how MLCPD contrasts with prior categories of research, highlighting its combination of universality, losslessness, and scalability.

Table 1: Comparison of prior work categories w.r.t. MLCPD goals (✓: present, ×: absent).

Category	Structural Rep.	Cross-Lang.	Lossless	Uniformity
Code Datasets (The Stack, etc.)	×	×	×	×
Parsing Frameworks (Tree-sitter)	✓	×	✓	×
IRs (LLVM)	✓	✓	×	×
MLCPD	✓	✓	✓	✓

3 Dataset Design and Implementation

The design and construction of the MultiLang Code Parser Dataset (MLCPD) was guided by a single unifying principle: to establish a *universal, lossless, and language-agnostic representation* of program structure that enables consistent analysis across ten programming languages—C, C++, C#, Go, Java, JavaScript, Python, Ruby, Scala, and TypeScript. Achieving this required not only scalable data collection, but also the creation of a new abstraction layer over existing parsing technologies that could reconcile their inherent syntactic diversity.

3.1 Data Collection and Preprocessing

MLCPD builds upon permissively licensed source files from the StarCoder dataset (21), filtered under the MIT, Apache-2.0, and BSD licenses. Each file underwent multi-stage preprocessing: (1) normalization of character encoding to UTF-8, (2) removal of Byte Order Marks, non-printable characters, and redundant whitespace, and (3) statistical filtering to exclude trivial and auto-generated code. Using an interquartile-range (IQR) filter, we eliminated outliers and retained only files containing between 10 and 10,000 lines, ensuring a corpus of non-trivial, parseable, and representative programs.

Following preprocessing, all code samples were revalidated for syntactic completeness and deduplicated via content-based hashing, ensuring that each source file corresponds to a unique structural representation. This normalization process established a high-quality foundation for large-scale parsing and schema inference.

3.2 Design Philosophy and Universal Schema

Our design goal extended far beyond simple code parsing—it sought to encode the *entire syntactic structure* of diverse languages in a shared representational space. To achieve this, we defined four non-negotiable design criteria:

- **Losslessness:** Preserve every syntactic element (tokens, delimiters, punctuation, whitespace) without semantic compression.
- **Uniformity:** Enforce a consistent JSON schema across all languages.
- **Queryability:** Enable efficient and language-independent structural querying.
- **Scalability:** Support millions of files with minimal memory overhead.

Prior to implementation, extensive empirical analysis was conducted over 5,000 sample files (500 per language) to identify structural commonalities. While earlier approaches to unification attempted to impose semantic alignment (e.g., mapping class inheritance across paradigms), we found that such methods violated the losslessness principle. Instead, MLCPD's innovation lies in enforcing *structural homogeneity without suppressing syntactic heterogeneity*. This insight led to the design of a four-layer universal AST schema.

Layer 1: Metadata Block. Captures global file characteristics and parser integrity diagnostics, serving both analytical and validation roles.

Listing 1: Example metadata block for a parsed source file.

```
"metadata": {
  "lines": 247,
  "avg_line_length": 34.2,
  "nodes": 1853,
  "errors": 0,
  "source_hash": "a3f5e8c9..."
}
```

Each field quantifies core properties: line count and average length indicate code density, node count measures syntactic complexity, and errors reflect parser resilience. The `source_hash` field uses SHA-256 for deterministic fingerprinting, chosen over weaker alternatives (MD5, SHA-1) for its collision resistance (17; 18). This facilitates reproducible node-level identification, content-addressable deduplication, and incremental dataset updates.

Layer 2: Flat Node Array. Maps the entirety of the file's syntactic structure into a list of atomic, directly-addressable nodes. This process of linearizing the tree structure significantly boosts the efficiency of code analysis and traversal operations.

Listing 2: Representative AST node structure.

```
"nodes": [
  {
    "id": 0,
    "type": "function_definition",
    "text": "def calculate_sum(a, b):\n    return a + b",
    "parent": null,
    "children": [1, 5, 6],
    "start_point": {"row": 10, "column": 0},
    "end_point": {"row": 11, "column": 20}
  }
]
```

This representation allows constant-time ($O(1)$) traversal between nodes, supports vectorized analysis enabling parallel processing, and ensures serialization stability guaranteeing consistent representation across processing environments.

Layer 3: Node Categorization. Groups syntactic nodes into a universal three-level taxonomy—`declarations`, `statements`, and `expressions`—representing the highest shared abstraction among all ten languages.

Listing 3: Categorical indexing of declarations, statements, and expressions.

```
"node_categories": {
  "declarations": {"functions": [0, 42], "classes": [120, 256]},
  "statements": {"loops": [65, 78], "returns": [35, 140]},
  "expressions": {"calls": [25, 36], "identifiers": [10, 11]}
}
```

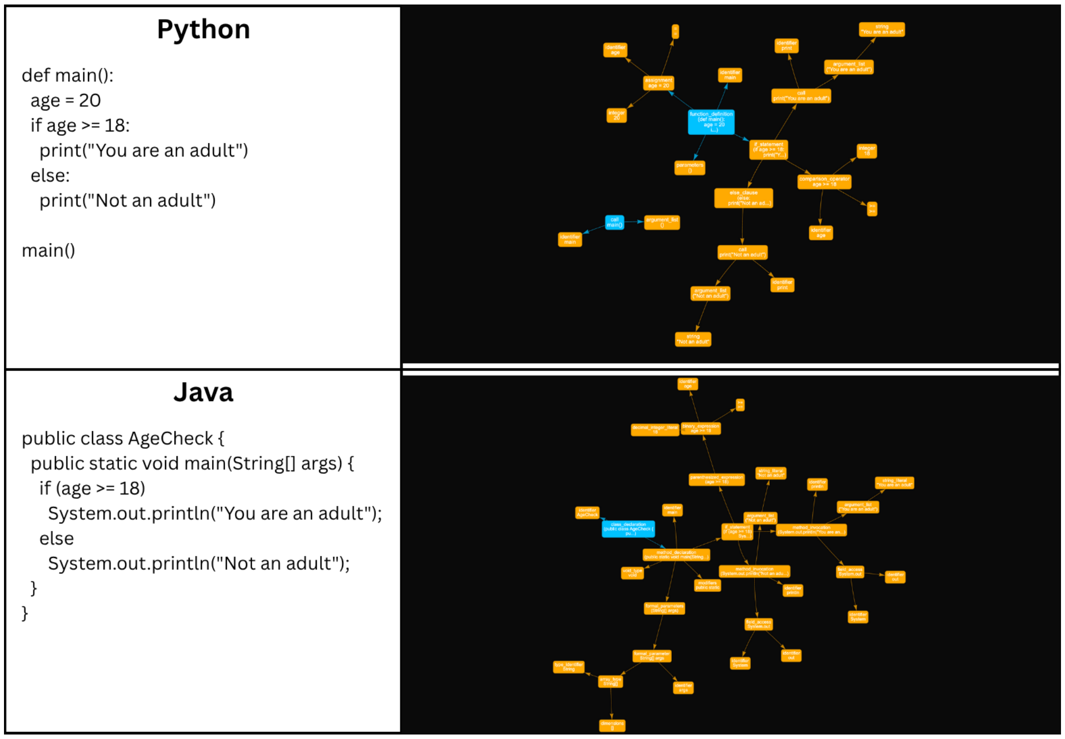
This enables direct category-based retrieval (e.g., all function definitions) without recursive traversal, which is critical for large-scale analytical queries.

Layer 4: Cross-Language Map. Abstracts language-specific constructs into universal schema roles while maintaining the original syntax.

Listing 4: Cross-language normalization of function and class declarations.

```
"cross_language_map": {
  "function_declarations": [
    {"node_id": 0, "universal_type": "function", "name": "calculate_sum"}
  ],
  "class_declarations": [
    {"node_id": 120, "universal_type": "class", "name": "DataProcessor"}
  ]
}
```

This layer bridges syntactic differences, enabling structurally equivalent constructs across languages (e.g., Python's `def` and Java's `public static void`) to be queried uniformly.



```
def main():
  age = 20
  if age >= 18:
    print("You are an adult")
  else:
    print("Not an adult")

main()
```

```
public class AgeCheck {
  public static void main(String[] args) {
    if (age >= 18)
      System.out.println("You are an adult");
    else
      System.out.println("Not an adult");
  }
}
```

Figure 1: Visualization of Python and Java "Age Check" programs parsed under the universal schema, demonstrating consistent alignment across languages.

3.3 Empirical Schema Validation and Parser Evolution

Initial prototypes used rule-based language mappers and regular-expression parsers to define the schema mapping. While these provided limited coverage, both approaches proved brittle and lossy. Rule-based mappers captured only predefined patterns and required quadratic maintenance as languages increased, while regex-based approaches failed on nested constructs and omitted non-captured syntax.

The breakthrough came with the adoption of Tree-sitter (23), an incremental parsing system with hand-crafted, formally verified grammars. Tree-sitter offered three indispensable features: (1) complete syntactic coverage, (2) a uniform parsing interface across languages, and (3) robust error recovery for real-world code fragments. These properties satisfied all four schema design principles, as shown in Table 2.

Table 2: Comparison of parsing techniques with respect to schema design requirements.

Technique	**Lossless**	**Uniform**	**Queryable**	**Scalable**
Rule-Based Mappers	×	×	×	×
Regex-Based Parsing	×	✓	✓	✓
Tree-sitter Grammars	✓	✓	✓	✓

3.4 Parsing and Normalization Pipeline

All ten Tree-sitter grammars were compiled into a shared dynamic library, ensuring uniform parsing behavior. The processing pipeline comprises six deterministic stages: (1) language detection and grammar dispatch, (2) recursive AST extraction, (3) node categorization, (4) cross-language mapping, (5) JSON schema validation, and (6) Parquet serialization for scalable storage.

Algorithm 1 and Algorithm 2 summarize the extraction and mapping stages in Tree-sitter–based parsing.

```
Input: Tree $T$, source string $S$, language $L$
Output: Hierarchical list of AST nodes
Initialize empty list $N$ and counter $id \leftarrow 0$
TraverseNode($node, parent$) if ShouldSkipNode(node) then
    foreach $child$ in $node.children$ do
        TraverseNode($child, parent$)                    // Skip delimiters
    end
    return
end
Create dictionary $d$ with fields:
$d$[id] $\leftarrow id$; $d$[type] $\leftarrow node.type$; $d$[text] $\leftarrow S[node.start\_byte : node.end\_byte]$;
$d$[parent] $\leftarrow parent$; $d$[children] $\leftarrow [\,]$
foreach $child$ in $node.children$ do
    $cid \leftarrow$ TraverseNode($child, id$)
    if $cid \neq$ None then
        append $cid$ to $d$[children]
    end
end
append $d$ to $N$; $id \leftarrow id + 1$
return $d$[id]
TraverseNode($T.root$, None)
return { "nodes" : $N$ }
```

Algorithm 1: Extract AST Structure

```
Input: AST structure $A$, categories $C$, language $L$
Output: Cross-language map $\mathcal{M}$
Initialize $\mathcal{M} \leftarrow$ {"function_declarations" : [ ]}
foreach $f$ in $C$[declarations][functions] do
    $n \leftarrow$ GetNodeById($f$); $name \leftarrow$ ExtractNameFromText($n$[text], $L$)
    Append {"node_id" : $f$, "universal_type" : "function", "name" :
      $name$, "text_snippet" : $n$[text][: 100]} to $\mathcal{M}$["function_declarations"]
end
return $\mathcal{M}$
```

Algorithm 2: Create Cross-Language Map

3.5 Dataset Statistics and Structural Insights

After normalization, MLCPD comprises over seven million records (Table 3), distributed evenly across ten languages. The dataset occupies approximately 114 GB in Parquet format and 600 GB in-memory—an indication of the balance between structural richness and compression efficiency.

Table 3: Dataset overview.

Metric	Value
Total Languages	10
Total Files	40
Total Records	7,021,722
Disk Size	∼114 GB (Parquet)
Memory Size	∼600 GB (in-memory)

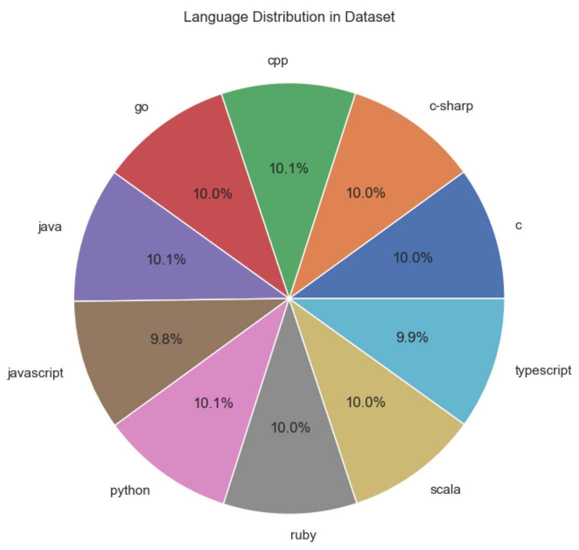


Figure 2: Distribution of records across ten programming languages. The uniform proportions validate balanced representation.

Parsing reliability remains exceptional, with 7,021,718 out of 7,021,722 files parsed successfully (99.99994%), as summarized in Table 4. This near-perfect conversion rate underscores the robustness of the pipeline.

Table 4: Conversion reliability statistics.

Successful conversions	7,021,718
Total attempted	7,021,722
Failures	4
Success rate	99.99994%

Figure 3 compares the average AST node count per file, revealing structural verbosity patterns—C++ and Go produce denser syntactic trees than Python or Ruby due to explicit type and control structures.

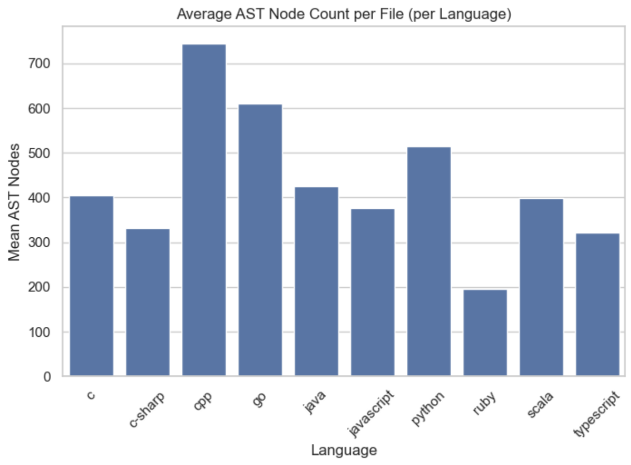


Figure 3: Average AST node count per file across languages. Higher counts reflect verbose syntactic forms and explicit typing.

Node density (nodes per line) in Figure 4 reflects each language's syntactic granularity. Python and Scala show higher densities due to indentation-based structuring and implicit typing, which decompose compact expressions into multiple AST nodes. In contrast, Go and Java achieve lower densities through explicit type declarations and block delimiters, indicating that the schema faithfully captures linguistic verbosity without distortion.

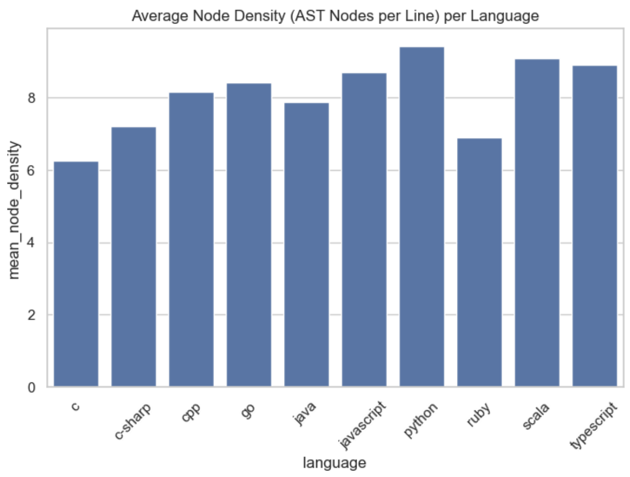


Figure 4: Average node density per language, reflecting syntactic compactness and complexity.

Figure 5 compares disk and memory footprints, revealing a stable compression ratio of roughly $5.5\times$ across all languages. This consistency indicates that the universal schema introduces uniform structural overhead regardless of syntax complexity. The result confirms MLCPD's scalability and demonstrates that schema normalization yields balanced, storage-efficient representations across diverse programming paradigms.

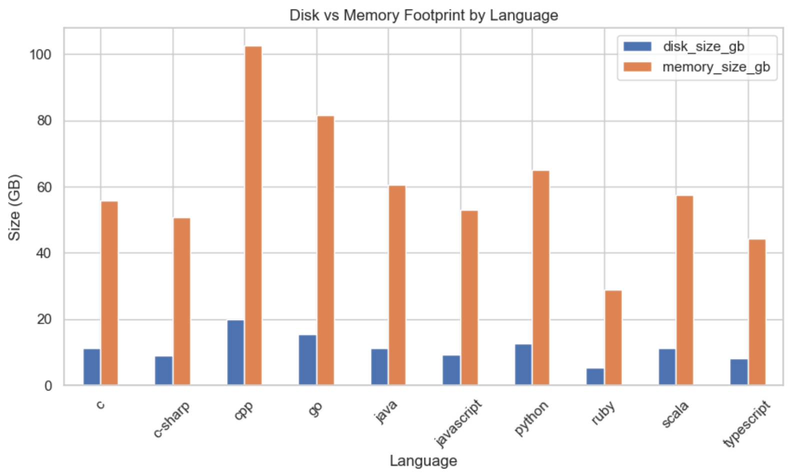


Figure 5: Disk versus memory footprint per language, confirming consistent compression ratios across corpora.

3.6 Cross-Language Structural Correlation

The cosine similarity matrix in Figure 6 quantifies how closely languages align in their syntactic structure under the universal schema. High similarity scores (>0.90) between C and C++ indicate that these statically typed, statement-driven languages share closely matching structural patterns, driven by explicit type declarations and block-based syntax. C# and Scala also exhibit strong alignment with Java (0.87–0.89), reflecting their object-oriented and strongly typed design lineage. In contrast, JavaScript and TypeScript form a distinct high-similarity cluster (0.96), confirming their near-identical grammatical roots within prototype-based programming. Python and Ruby demonstrate moderate mutual similarity (0.88) but diverge structurally from compiled languages due to their expression-oriented, dynamically typed nature. Go, while syntactically simpler, remains closer to C-family languages (0.76–0.82), supporting its design heritage. Overall, these correlations validate that MLCPD's universal schema captures genuine cross-language syntactic relationships rather than superficial token overlaps.

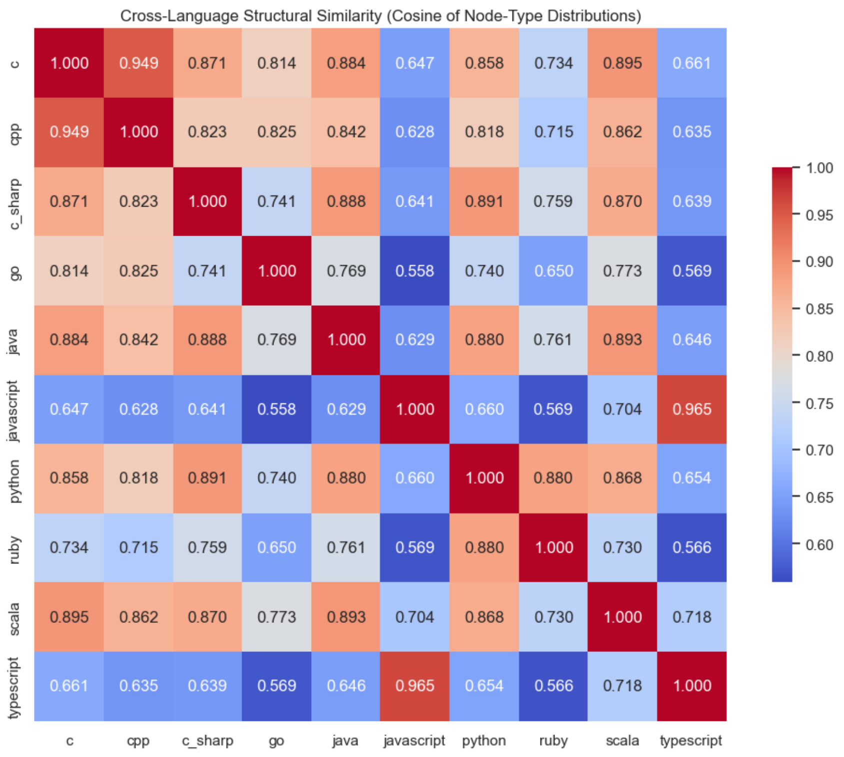


Figure 6: Cross-language structural similarity (cosine of node-type distributions).

Finally, principal component projection (Figure 7) visualizes schema embeddings, where semantically proximate languages (e.g., Java and Scala; JavaScript and TypeScript) occupy neighboring regions in latent space—strong evidence that MLCPD captures deep syntactic link across languages.

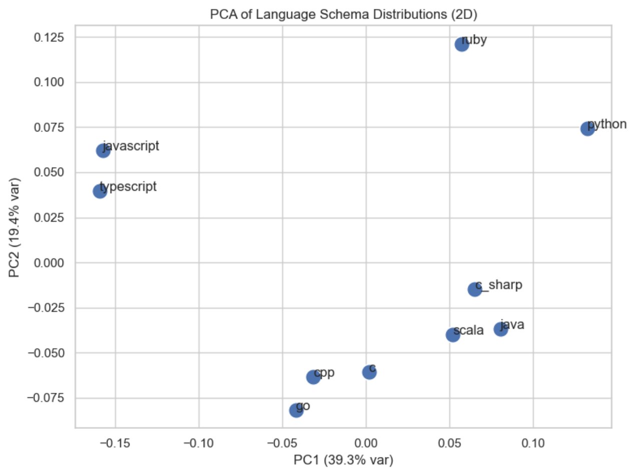


Figure 7: PCA projection of language schema embeddings, revealing natural syntactic clusters.

4 Qualitative Example: Python vs. Java "Age Check" Schema

To demonstrate the universality of MLCPD's schema, we visualize equivalent Python and Java programs that determine if a person is an adult. Despite lexical and syntactic differences, their underlying ASTs align seamlessly under the same universal schema (shown in Figure 1).

Both code samples produce function nodes with identical hierarchical roles: parameter declarations, conditional statements, and return expressions. The cross-language map unifies these constructs through shared `"function"`, `"conditional"`, and `"literal"` node types, validating schema coherence.

5 Discussion and Limitations

MLCPD is designed as a research-grade corpus rather than an exhaustive code archive. While it ensures near-perfect conversion rates and schema uniformity, some limitations remain: *(a)* language coverage is currently limited to ten high-impact languages; *(b)* semantic annotations (e.g., data-flow, control-flow edges) are not yet included; and *(c)* minor discrepancies in indentation and tokenization may persist across certain Tree-sitter grammars.

Nonetheless, the dataset establishes a foundational layer for building cross-language representation models, interpretable code embeddings, and multilingual static analysis tools.

6 Conclusion

We have presented MLCPD, a large-scale, structurally uniform dataset for multilingual code understanding. By constructing a universal AST schema that spans ten programming languages, MLCPD bridges the divide between symbolic program structure and data-driven modeling. Its high conversion accuracy, balanced coverage, and visualized structural correspondences make it a key resource for future research in multilingual program analysis, cross-language translation, and hybrid graph-language modeling. The dataset and scripts are open-sourced on Hugging Face and GitHub for community use.

References

[1] Shatnawi, A., Mili, H., Abdellatif, M., Guéhéneuc, Y. G., Moha, N., Hecht, G., ... & Privat, J. (2019). Static code analysis of multilanguage software systems. arXiv preprint arXiv:1906.00815.

[2] Mayer, P., & Schroeder, A. (2012, September). Cross-language code analysis and refactoring. In 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (pp. 94-103). IEEE.

[3] Linos, P. K., Chen, Z. H., Berrier, S., & O'Rourke, B. (2003, May). A tool for understanding multi-language program dependencies. In 11th IEEE International Workshop on Program Comprehension, 2003. (pp. 64-72). IEEE.

[4] GitHub. GitHub Copilot: Your AI pair programmer. `https://github.com/features/copilot`, 2021. Accessed: October 2024.

[5] Amazon Web Services. Amazon CodeWhisperer: ML-powered code generation. `https://aws.amazon.com/codewhisperer/`, 2022. Accessed: October 2024.

[6] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

[7] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

[8] Li, W., Marino, A., Yang, H., Meng, N., Li, L., & Cai, H. (2024). How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. ACM Transactions on Software Engineering and Methodology, 33(3), 1-46.

[9] He, H., Lin, X., Weng, Z., Zhao, R., Gan, S., Chen, L., ... & Xue, Z. (2024). Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In 33rd USENIX Security Symposium (USENIX Security 24) (pp. 1759-1776).

[10] Siddiqui, S., Metta, R., & Madhukar, K. (2023, October). Towards multi-language static code analysis. In 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 81-82). IEEE.

[11] Mayer, P., Kirsch, M., & Le, M. A. (2017). On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. Journal of Software Engineering Research and Development, 5(1), 1.

[12] Teixeira, G., Bispo, J., & Correia, F. F. (2021, June). Multi-language static code analysis on the lara framework. In Proceedings of the 10th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis (pp. 31-36).

[13] Negrini, L., Ferrara, P., Arceri, V., & Cortesi, A. (2023). LiSA: a generic framework for multilanguage static analysis. In Challenges of Software Verification (pp. 19-42). Singapore: Springer Nature Singapore.

[14] Roth, T., Näumann, J., Helm, D., Keidel, S., & Mezini, M. (2024, October). AXA: Cross-Language Analysis through Integration of Single-Language Analyses. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (pp. 1195-1205).

[15] Casey, B., Santos, J. C., & Perry, G. (2025). A survey of source code representations for machine learning-based cybersecurity tasks. ACM Computing Surveys, 57(8), 1-41.

[16] Microsoft. (n.d.). CA5351: Do not use insecure cryptographic persistence algorithms. Retrieved Oct. 15, 2025, from `https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca5351`

[17] National Institute of Standards and Technology. (2015). Secure Hash Standard (SHS) (FIPS Pub. 180-4). U.S. Department of Commerce.

[18] Velvindron, L., Moriarty, K., & Ghedini, A. (2021). Deprecating MD5 and SHA-1 signature hashes in TLS 1.2 and DTLS 1.2 (Request for Comments No. 9155). RFC Editor.

[19] Li, W., Ming, J., Luo, X., & Cai, H. (2022). PolyCruise: A Cross-Language dynamic information flow analysis. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 2513-2530).

[20] Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., ... & de Vries, H. (2022). The stack: 3 tb of permissively licensed source code. arXiv preprint arXiv:2211.15533.

[21] Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., ... & de Vries, H. (2023). Starcoder: may the source be with you!. arXiv preprint arXiv:2305.06161.

[22] Raychev, V., Bielik, P., & Vechev, M. (2016). Probabilistic model for code with decision trees. ACM SIGPLAN Notices, 51(10), 731-747.

[23] Brunsfeld, M. (2018). Tree-sitter: An incremental parsing system for programming tools. URL: https://doi. org/10.5281/zenodo, 4619183.

[24] Ma, W., Liu, S., Lin, Z., Wang, W., Hu, Q., Liu, Y., ... & Liu, Y. (2023). Lms: Understanding code syntax and semantics for code analysis. arXiv preprint arXiv:2305.12138.

[25] Husain, H., Wu, H. H., Gazit, T., Allamanis, M., & Brockschmidt, M. (1909). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.(2019). arXiv preprint arXiv:1909.09436.

[26] Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., ... & de Vries, H. (2022). The stack: 3 tb of permissively licensed source code. arXiv preprint arXiv:2211.15533.

[27] Hugging Face. (n.d.). Datasets: Index. Retrieved October 15, 2025, from `https://huggingface.co/docs/datasets/index`

[28] Gong, L., Elhoushi, M., & Cheung, A. (2024). Ast-t5: Structure-aware pretraining for code generation and understanding. arXiv preprint arXiv:2401.03003.

[29] Nong, Y., Fang, R., Yi, G., Zhao, K., Luo, X., Chen, F., & Cai, H. (2024, April). Vgx: Large-scale sample generation for boosting learning-based software vulnerability analyses. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-13).

[30] Parr, T. J., & Quong, R. W. (1995). ANTLR: A predicated-LL (k) parser generator. Software: Practice and Experience, 25(7), 789-810.

[31] Ford, B. (2004, January). Parsing expression grammars: a recognition-based syntactic foundation. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 111-122).

[32] Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In International symposium on code generation and optimization, 2004. CGO 2004. (pp. 75-86). IEEE.

[33] Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2013). The Java virtual machine specification. Addison-wesley.

[34] Johnstone, A., & Scott, E. (1998). Generalised recursive descent parsing and follow-determinism. In: Koskimies, K. (eds) Compiler Construction. CC 1998. Lecture Notes in Computer Science, vol 1383. Springer, Berlin, Heidelberg.