CSC8499: MSc ACS Dissertation
Machine Learning for Full Line Code Completion

FILIP KOVARIK (240399999), School of Computing, Newcastle University

Language models for code completion are increasingly used to boost productivity in software development. However, most existing solutions are proprietary, cloud-based services, raising concerns over privacy, recurring costs, and service availability. This work addresses these limitations by introducing open-source models for Java full-line code completion that can be deployed locally, even on low-end hardware, along with the tools for a reproducible training pipeline. To meet the constraints of on-device deployment, we combine model size reduction techniques with a novel data preprocessing and grammar-aware tokenization strategy that surpasses the current state-of-the-art in token compression efficiency. Our approach achieves an average 26% improvement in tokenizer compression compared to leading alternatives, enabling models to fit substantially more code within the same context window. This improvement facilitates more contextually relevant completions, faster generation speeds, and the potential for smaller vocabulary sizes with reduced memory usage. The preprocessing pipeline also delivers notable efficiency gains: starting from 85 GB of raw Java code, the process reduces the dataset to 66 GB while retaining 97.5% of the original files (removing low-quality code and irrelevant files while preserving meaningful content for training). We evaluate our models on publicly available Java code datasets, demonstrating significant improvements in compression ratio compared to existing tokenizers. Our results highlight a practical path toward accessible, privacy-preserving code completion tools that are fully under user control.

ACM Reference format:

Filip Kovarik (240399999). 2025. CSC8499: MSc ACS Dissertation: Machine Learning for Full Line Code Completion. 1, 1 (August 2025), 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

This project delivers an open-source, locally deployable Java full-line code completion system that rivals the performance of proprietary alternatives while operating efficiently on resource-constrained hardware. The system integrates a highly curated data processing pipeline, reducing an 85 GB raw dataset to 66 GB with minimal loss of relevant content, and introduces a grammar-aware tokenization strategy that achieves an average 26% improvement in token compression efficiency over state-of-the-art methods. Together, these innovations enable faster inference, lower memory usage, and richer code context within the same sequence length budget. The trained transformer-based model, built entirely on permissively licensed data, demonstrates the feasibility of delivering high-quality, privacy-preserving developer tooling without reliance on closed-source or cloud-based infrastructure.

Code completion tools are among the most widely used features in modern integrated development environments (IDEs) [1], [2]. These tools vary considerably in computational complexity, resource requirements, and intended use cases. At one end of the spectrum, lightweight single-token predictors rely on language grammar rules and static analysis of the project's structure [3]. Such approaches are highly responsive and resource-efficient, running entirely within the IDE. More recently, proprietary IDE vendors have integrated compact,

domain-specific language models to provide full-line code completions [4]. These models deliver more sophisticated, context-aware suggestions, while maintaining the low latency and small memory footprint required for real-time use on typical developer hardware. These advancements offer a significant productivity gain, while also avoiding the performance, privacy, cost, and availability concerns associated with cloud-based code generation using large language models (LLMs) [5], [6].

Despite these advances, equivalent functionality remains largely unavailable in the open-source ecosystem. Most existing solutions are closed-source [4], [7], often trained on undisclosed datasets and distributed under restrictive licenses. This creates a gap in access to high-quality developer tooling for users of open-source IDEs.

This project addresses this gap by identifying and executing several of the key tasks involved in building a specialised machine learning language model for this use case, using entirely open-source components and data. By applying the findings from promising new research directions, it innovatively improves design areas, such as tokenization and data processing techniques, allowing the solution to exceed the performance of existing closed-source alternatives. A lightweight language model for Java full-line code completion is then trained to analyse and confirm the performance improvements. The model is trained on open, permissively licensed data and is deployable locally.

The specialised nature of the model requires more highly curated training data than is typical, indicating that data cleaning and structuring form a larger component of the development pipeline. A particular focus is also placed on tokenizer design, a critical component in determining both model efficiency and effectiveness. Since the computational cost of transformer-based models scales quadratically with token sequence length [8], investigating opportunities for further improvements in tokenizer design has a high priority given the resource-constrained deployment environment. Prior research [4] has shown that tokenizer innovations are key to enabling high-quality code suggestions from smaller models. In doing so, it seeks not only to exceed the performance of proprietary alternatives but also to contribute reusable tools, curated datasets, and experimental findings to the open-source community.

1.1. Aim and Objectives

The primary aim of this project is to improve the performance and efficiency of full-line code completion models that can be deployed locally with minimal resource requirements, focusing on Java. In addition, the project seeks to advance the state of open-source tooling by contributing curated datasets, preprocessing pipelines, and novel insights into efficient model design. The short project timescale and limited compute resources available make it infeasible to entirely replicate the functionality of a closed-source solution (such as building an IDE plugin that implements the trained model for code completion). Instead, using a literature review driven approach to understanding the problem space, this project will identify and prioritise tasks that may provide foundational work reusable by the open-source community, or contribute novel research insights into model efficiency.

Key objectives:

(1) Investigate the current state-of-the-art techniques for code language models
(2) Find an appropriate Java code dataset
(3) Develop an effective data preprocessing pipeline
(4) Design and evaluate a novel tokenizer for Java code
(5) Train a compact language model for full-line Java code completion

(6) Implement a post-processing algorithm for filtering and ranking model outputs
(7) Evaluate model performance and accuracy against similar existing solutions
(8) Publish reusable tools, datasets, and research findings to support further innovation within the open-source community

1.2. Outline

This introduces an overview of the structure of the dissertation. In sections 2 and 3 we establish the current research for the individual components of this work and related work. Section 4 describes the approach taken for achieving the defined aim and objectives and introduces the achieved results. Section 5 is dedicated to evaluating and explaining those results. Section 6 identifies the constraints and factors that might affect the generalizability of the findings. Section 7 provides directions for future research and Section 8 concludes the findings.

2. Background Research

2.1. Code Completion

Code completion is a common feature in many modern IDEs which helps speed up development and fix common mistakes. Most tools are focused either on single token suggestions or multi-line code generation. The former typically uses simple language grammar rules to identify a list of possible candidates and small ML models to rank the candidates based on relevance probability [3]. The candidate list is typically presented to the programmer as a context window as they type, allowing them to choose the desired completion with a single keystroke. These algorithms are cheap to run, allowing them to be deployed locally within the IDE on virtually any hardware. On the other hand, multi-line generation algorithms are powered by LLMs which were pre-trained to understand natural language and further fine-tuned on code datasets consisting of multiple programming languages. They process code that is already present in the current file as well as additional information about the project structure. The completion suggestion is then presented to the programmer as “grey text” that appears after the cursor as they type. They can also choose to accept the completion with a single keystroke, saving the programmer from manually writing multiple lines of code. However, these algorithms are expensive to deploy and most developer hardware would not be capable of running them locally. Therefore, they are being offered as cloud-based services [5], [6]. This raises security concerns for individuals and organizations, as by using these tools they are sharing private and potentially sensitive data with third party providers.
There is a gap in the market for sophisticated code generation tools that do not compromise sensitive data. Recent research investigated employing smaller-sized language models for single-line code generation, each model focused on a single programming language [4]. Due to the smaller size and narrow focus of these models, they can be deployed locally within the IDE and still offer fast and accurate suggestions. However, as this is a commercial product, the models are closed-source, which makes extending the research in this area difficult.

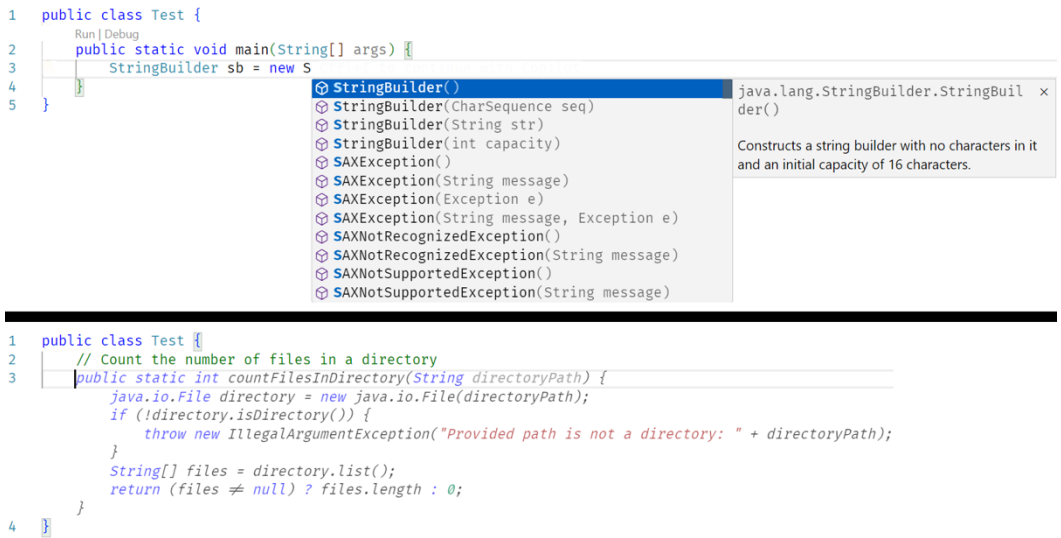


Fig. 1. The first example shows a single-token code suggestion offering an ordered list of candidates. The second example below shows a multi-line code completion. The block of code after the cursor on line 3 is generated based on the context of the comment above. The suggestion can be accepted by pressing the Tab key.

2.2. Language Models

Language models are used to help predict a sequence of words (referred to as tokens) following a defined set of input tokens, known as context. The models used for text generation are either purely statistical or neural. Statistical models estimate the probability of the next token based on a fixed window of previous tokens. These are called n-gram models. They rely on frequency counts of tokens in a training dataset to assign probabilities to unseen sequences [9]. They are simple but don't generalize beyond the fixed n-gram context and lack any notion of long-range structure.

Recurrent neural networks (RNNs) are capable of learning long-term dependencies by carrying a hidden state that captures historical context [10]. However, RNNs process sequences one element at a time and they struggle with very long dependencies, making training slower and difficult to parallelize.

Transformer-based models employ self-attention layers to process all tokens and their interactions in parallel, allowing efficient modelling of long-range dependencies across the context [11], [12]. Input tokens are embedded into vectors augmented with positional encodings (or rotary embeddings in LLaMA-like models), then passed through repeated layers of multi-head self-attention and feed-forward sublayers, each wrapped with residual connections and normalization [13]. In autoregressive mode (decoder-only), the model generates one token at a time, masking attention so each position only sees previous context and approximates the next-token probability distribution via softmax over its vocabulary.

Transformer-based models superseded RNNs and statistical models. They are currently the mainstream in text generation and therefore the model developed for this project uses this architecture.

The focus on a single programming language is justified by attempting to maximise performance and accuracy. Training a transformer-based model on multiple programming languages would require scaling the model's capacity to maintain acceptable accuracy levels, however, at the cost of performance on low-end hardware [14].

2.3. Tokenization

Tokenization is the crucial first step in transformer-based models, splitting raw text (or code) into a sequence of discrete tokens. Choice of tokenizer (algorithm, vocabulary size, pre-processing) has been shown to significantly affect model efficiency, compression rate, and downstream accuracy [8]. It helps language models process large amounts of text during training and inference more efficiently. This step breaks down text into token units, and each token is assigned a numerical representation. These are subsequently passed to the model. The aim is to maximise compression – represent given input sequence with fewest number of tokens possible. Various techniques exist for achieving this, including word-level, character-level, or subword-level tokenizations. Word-level tokenizers split text on whitespace and punctuation into whole words. They are simple to understand and implement, however, they are inefficient as they create large vocabularies and fail to deal with rare or novel words that were seen infrequently during training [15]. Character-level tokenizers treat every character as an individual token. This approach is robust to unknown words, but it produces long sequences and loses word-level semantics [16]. Subword-level tokenizers break down text into subwords, which balances vocabulary size and word coverage. These tokenizers are dominantly used for transformer-based language models [11], [17]. The main algorithms classifying as subword-level tokenizers include Byte-Pair Encoding (BPE), WordPiece, and Unigram. BPE is the algorithm mainly used for autoregressive transformer models, such as GPT or LLaMA [18], [19]. It works by iteratively merging the most frequent character or byte pairs to build tokens from single characters up to full words. Recent research has shown that allowing BPE to merge tokens across tabs and spaces leads to better compression on code datasets, as code primarily consists of common and repeating statements consisting of more than one token [4]. Compression gains matter because the computing cost for transformer-based models grows quadratically with sequence length [8]. Even modest token reductions can cut FLOPs and memory usage by a large percentage.

3. Related Work

Over the past several years, deep learning-based code completion has evolved into a prominent research area, attracting significant attention from both academia and industry. This has led to an expansion of open-source models, along with publicly available training scripts, configuration files, and model weights. Most of these models are built on transformer decoder architectures, with variants inspired by models such as GPT and LLaMA. Notable representatives include InCoder [20], StarCoder [21], CodeLLaMA [22], the CodeGen family [23], [24], and PolyCoder [14], among others.

While these models represent important milestones in code intelligence research, they fall short of meeting the requirements for lightweight, locally deployable code completion systems. One of the key limitations is their size. Many of these models contain hundreds of millions, or billions, of parameters, making them impractical for local environments due to excessive computing, memory, and storage demands.

In addition, only certain architectures, such as GPT-2 and LLaMA, currently benefit from optimized inference backends capable of running efficiently on consumer-grade hardware. Models based on other transformer variants may require substantial custom engineering to achieve real-time responsiveness in a local setting. Furthermore, many of these models have been trained with broad, general-purpose programming datasets, meaning they are not always fine-tuned for the specific, domain-targeted code completion scenarios that developers often require.

Many code LMs reuse tokenizers from natural language base models, which may be suboptimal. Using a code-specific tokenizer was shown to use approximately 25% fewer tokens on code inputs [8]. This compression improved inference speed and allowed a larger effective context for code. Thus, further improving tokenization methods for code models remains a key aspect of increasing inference speed and reducing computing requirements.

4. Methodology

The development of the proposed Java full-line code completion system followed a structured, multi-stage pipeline. Development of each stage was executed as a separate sprint with a predefined timeline to maintain control over the overall progress of the project while focusing on important milestones that needed to be achieved. The process can be summarised as follows:

1) **Data Acquisition and Preprocessing** – A large-scale corpus of publicly available Java source code was collected from permissively licensed repositories to ensure reproducibility and open-source compliance. Extensive filtering, formatting, and heuristic cleaning were applied to remove non-compilable and trivial or malformed samples. This stage was allocated 4 weeks of development time.
2) **Tokenization** – A custom Java grammar-aware lexer segmented source files into semantically meaningful tokens while preserving structural information critical for downstream model performance. Then, BPE was trained on the pre-tokenized corpus to create a vocabulary optimised for Java's lexical characteristics. This stage was also allocated 4 weeks of development time.
3) **Language Model Training** – Two transformer-based models were trained on the processed corpus, leveraging the improved tokenization to enhance inference speed and memory footprint. We reserved 5 weeks for this stage.

This multi-stage methodology ensured that efficiency gains achieved in earlier stages, such as dataset reduction and improved compression, propagated throughout the pipeline, resulting in a lightweight yet competitive model focused on privacy and local deployability.

4.1. Data Acquisition

For the purposes of this project, a Java portion of the StarCoder training dataset was used [25]. It is a branch of The Stack [26], a collection of permissively-licensed source code files, that has also been almost entirely deduplicated and cleaned. This subset consists of approximately 85 GB of source code distributed across roughly 20 million files. Processing such many files grouped together introduced an interesting challenge. Performing any operation on the data was faced with severe performance issues due to operating system's index file fragmentation. To improve the data processing efficiency, the files were split into smaller batches and grouped into uncompressed archives. This resulted in appx. 2,000 archives with 10,000 source code files each, which significantly improved processing performance. Using uncompressed archives ensures that no additional computing is required for decompressing when the source code files need to be accessed.

4.2. Data Preprocessing

This step is a big part of the pipeline. As the language model is relatively small and trained from scratch, there is no pre-training step on natural language. Java source code is the only focus. To

maximise the probability that the model focuses on useful patterns in the source code during training, the dataset needs to have a standardized format, and a comprehensive filter needs to be implemented which removes code that does not provide any meaningful value for the model. The implemented preprocessing steps are described in more detail below.

Filtering out useless code. Every source file was parsed and checked for syntax errors. All files producing errors during compilation were removed to help support the model in generating syntactically correct code. This was achieved with the help of abstract syntax trees (ASTs). The dataset was also found to contain obfuscated code – this is code that has sequences of random characters and numbers instead of meaningful names for classes, variables, identifiers, and other keywords. As this does not classify as valuable data, a filter was developed that detects and removes source files containing these hash-like names. Additionally, the filter was designed to remove source files containing only class declarations with an empty body.

Data cleaning. Code usually contains extra elements that do not affect the core logic of the program but can provide additional context for programmers to improve code readability. These elements are mainly comments, annotations, and prefix tags with repository information (the last was added by the creators of the dataset that we used, not typically found in source code). Training the model to understand natural language from comments or redundant elements could negatively impact code generation accuracy at this model scale, so these elements were removed from the source code.

Note: Some framework-dependent annotations (Spring, Quarkus…) alter the runtime behaviour and removing them can break functionality. For the scope of this project and the model, all annotations were removed regardless of their purpose which results in the model also omitting them in the generated code.

Standardizing the format. Coding style varies among programmers. Some prefer to put an opening brace on the same line with a method statement, others put it separately on a new line below. Styles vary for indentations, spaces, empty lines, and many other things. This creates additional overhead for the model as it would need to learn different patterns for similar source files that only differ in their coding style. As a result, the model can generate code following a different coding style than the context it is being used in. To mitigate this issue, all code sequences are converted to a standardized format during training and inference. Specific coding styles can be re-created using a formatter during post-processing of the generated code. The standardized format involves applying a formatter that converts code to a specific coding style, and subsequently removing all empty lines, indentation, and trailing whitespace as these do not contain any valuable patterns that are critical for the model to learn.

The challenging part of this stage was processing a dataset of this size efficiently. The long execution time of the preprocessing algorithms made it challenging to implement modifications in later stages. To overcome this issue, we implemented multi-threading to the preprocessing algorithms which allowed for faster execution time and more efficient processing.

```
class Solution
{
    public int findDerangement(int n)
    {
        dp = new Long[n + 1];
        return (int) find(n);
    }

    private static final int kMod = (int) 1e9 + 7;

    private Long[] dp;

    private long find(int i)
    {
        if (i == 0)
            return 1;
        if (i == 1)
            return 0;
        if (dp[i] != null)
            return dp[i];
        return dp[i] % kMod;
    }
}
```

```
class Solution {
public int findDerangement(int n) {
dp = new Long[n + 1];
return (int) find(n);
}
private static final int kMod = (int) 1e9 + 7;
private Long[] dp;
private long find(int i) {
if (i == 0)
return 1;
if (i == 1)
return 0;
if (dp[i] != null)
return dp[i];
return dp[i] % kMod;
}
}
```

Fig. 2. Example source code before (left) and after (right) applying the preprocessing steps

4.3. Tokenization

This is the focus of innovation for the project. Improving compression capabilities of the tokenizer would result in a better model performance as the source code can be represented in a lower number of tokens. During inference, the lower number of tokens the model needs to generate to represent a full line of code, the less computation time is spent on its execution. Normally, BPE splits text in the training corpus on whitespace to decompose each word into individual characters and iteratively builds the vocabulary by merging two characters (or existing merges) that appear most frequently in conjunction. Whitespace is treated as a separate token. This means that the number of tokens used to represent any text is *at minimum* equal to the number of individual words or symbols and the number of spaces separating the words. This can be expressed by the following formula: $t = 2n - 1$, where n is the number of words in the given sequence. This only represents scenarios where ideal compression is achieved – where each word in the sequence is represented as a single token in the vocabulary.

Recent research [4] applies a modified BPE tokenizer, which allows merging over tabs and spaces as well. This was found to improve compression on code, as programming languages consist of many common and recurring statements. An example of a common statement in Java would be "*public static void main(String[] args) {*". This merge modification enables statements like these to be represented as a single token in the vocabulary, if they appear frequently enough in the training corpus.

This project significantly improves upon the above findings by introducing a two-stage tokenization process which further exploits the restricted and repetitive syntax of Java.

4.3.1. **Stage 1 – Lexer-based Pre-tokenization**

Java syntax consists largely of a fixed set of keywords, operators, and symbols, each with a well-defined grammatical function. This deterministic structure allows any syntactically correct Java code to be fully parsed into a sequence of grammar tokens. Leveraging this property, a pre-tokenization step is introduced that is designed to improve the efficiency of the BPE tokenizer. Specifically, since the set of grammar tokens defined by the Java language is finite and known in advance, we reserve a subset of the vocabulary space for these tokens prior to training the BPE model. During training, these reserved tokens are treated as indivisible units – BPE does not attempt to decompose and learn them through the merging process.

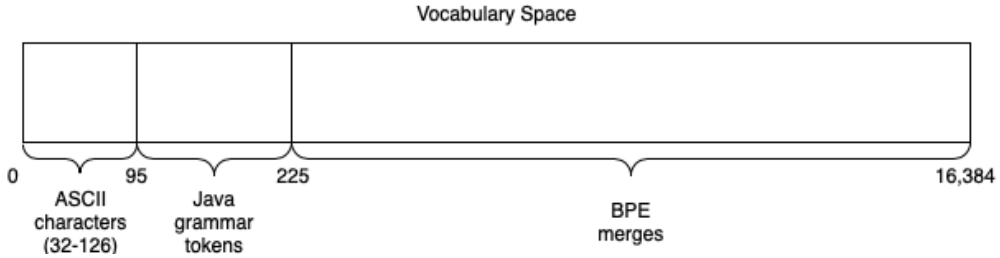


Fig. 3. Division of slots for pre-tokenization and BPE training in the vocabulary space. Slots 0-95 are reserved for ASCII characters than can make up an identifier name or a literal value and the end-of-line (EOL) character. Slots 96-225 are occupied by tokens extracted from a Java lexer. The remaining space is used for tokens merged by the BPE algorithm.

This guarantees that each grammar token is encoded as a single token, eliminating the risk of fragmentation across multiple subwords and improving token compression.

To construct this reserved set, we analysed the entire dataset using JavaParser [27], a lexer and parser library for Java source code. JavaParser classifies tokens into eight primary grammatical groups. Most of these groups (e.g., keywords, operators, and separators) consist of fixed language elements and can be directly added to the vocabulary. The exceptions are identifiers and literals (and comments but they are removed from the dataset), which may contain arbitrary user-defined text. These are left to be handled by the BPE tokenizer, which learns efficient subword representations through its merging process. The full distribution of JavaParser token groups across the dataset is shown in Fig. 4. This grammar-aware token initialization enables the tokenizer to compress code more effectively by leveraging the inherent structure of the Java language.

The most challenging part of this stage was overcoming an issue that was discovered during the core BPE tokenizer training. The BPE algorithm is implemented with Hugging Face's tokenizers library [28], which does not natively support exempting specific tokens from fragmentation while still allowing them to participate in merges with adjacent tokens. It is also not supported by other popular BPE libraries, such as SentencePiece [29] and YouTokenToMe [30]. To achieve the desired functionality, a custom encoding and decoding engine, featuring a series of hacks, was developed to preserve Java grammar tokens as indivisible units during training. The key idea is to represent each grammar token using a single character, thereby preventing the BPE algorithm from splitting these tokens during merge operations. To avoid collisions with existing characters in the dataset, we utilize Unicode's Private Use Area (PUA), a range of code points (U+E000 to U+F8FF) reserved for custom, application-specific characters. Each grammar token (e.g., 'if', 'for', 'class', '{', ';') is assigned a unique ID corresponding with their vocabulary position, which is mapped to a PUA code point by calculating *U+E000 + token_id*.

During the pre-tokenization phase, the dataset is parsed using a Java lexer for ANTLR [31], [32] to identify grammar tokens and classify each word or symbol. These tokens are then replaced with their corresponding PUA characters. Identifiers are handled separately: since their content is user-defined and cannot be assigned a fixed token, an identifier grammar token is inserted before each one, and the identifier's characters are encoded individually using the same PUA mapping. This approach ensures the full source code is transformed into a compact sequence of custom Unicode characters, where each Unicode character represents a complete grammar token or character.

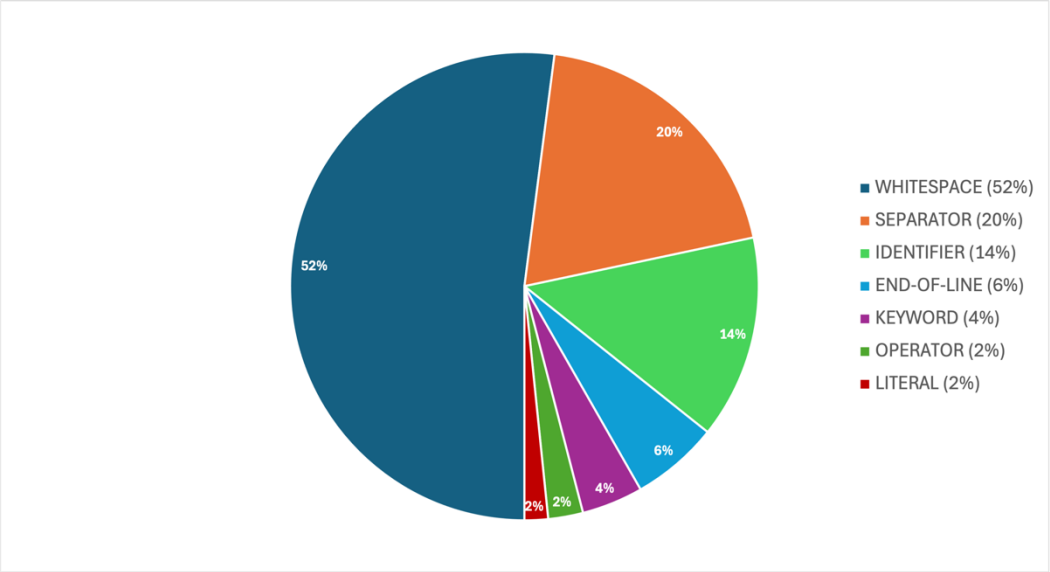


Fig. 4. Distribution of all token types across the dataset. Whitespace represents spaces between tokens (excluding the new-line character), Separators are brackets, a semicolon, a comma, and a dot, Identifiers are programmer-invented names for variables, classes etc., EOL stands for the end-of-line character '\n', Keywords are reserved words that have a predefined meaning (public, while, switch…), Operators (=, >, &), and Literals are digits, booleans, text in a string etc.

The decoding process reverses this transformation. Grammar token characters are mapped back to their original symbols using the vocabulary ID, while identifiers are reconstructed by joining their individual character encodings. Literal values, such as numeric and string literals, were not fully preserved as the mechanisms for it were not developed due to time constraints. Instead, placeholder values were inserted (e.g., 0 for numeric literals, empty quotes for strings) to maintain syntactic correctness without affecting downstream tokenization.

The encoding-decoding workflow allows grammar-aware tokenization while maintaining compatibility with the BPE training pipeline. The encoding process is illustrated in Fig. 5.

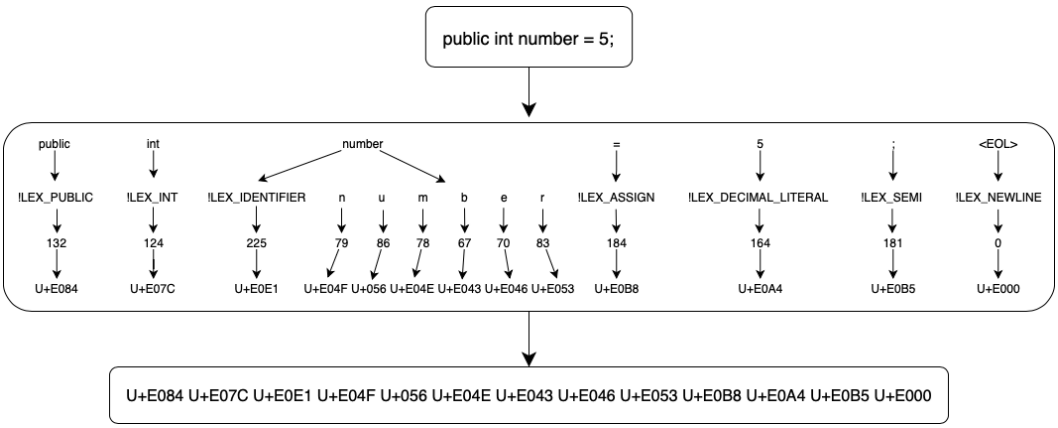


Fig. 5. Process of encoding an example Java statement into a sequence of Unicode characters during pre-tokenization. The IDs assigned to each token are stored in a map. Decoding is just a reverse of the encoding process.

4.3.2. **Stage 2 – BPE Training**

At this stage of the pipeline, the BPE tokenizer is trained on the dataset that has been transformed into sequences of Unicode characters, as described in Section 4.3.1. Following prior work [4], we adopt an initial vocabulary size of 16,384 tokens, which represents a widely accepted trade-off between compression efficiency and model memory requirements. This serves as a baseline for our experiments.

The first 225 entries in the vocabulary are reserved for special tokens, including markers for unknown tokens, padding, and end-of-line indicators, in addition to the ASCII character set and the Java grammar tokens extracted during the pre-tokenization stage. A visualization of the vocabulary layout is provided in Fig. 3. This design ensures that all grammar tokens (except identifiers) remain atomic units during tokenization, preventing BPE from breaking them into subwords.

Using this setup, the BPE model was trained on a 5% subset of the dataset. Empirical testing showed that increasing the training data beyond this point yielded minimal improvements in compression, while significantly increasing training time and hardware requirements. For benchmarking, we also trained a baseline tokenizer using the unmodified state-of-the-art BPE implementation [4]. When tested on the same validation data, our grammar-aware tokenizer consistently outperformed the baseline, achieving a 25% improvement in token compression.

Following this result, we investigated whether a smaller vocabulary size could maintain similar performance while reducing the memory footprint of the downstream language model. To that end, we trained a series of models with vocabulary sizes ranging from 16,384 to 4,096, in steps of 1,024. All vocabulary sizes were chosen as powers of two to align with memory and cache optimization patterns, ensuring efficient use of memory blocks.

Across all vocabulary configurations, we trained tokenizer models using both our grammar-aware approach and the unmodified baseline algorithm. For each vocabulary size, our tokenizer consistently achieved better compression across the board, with an average improvement of 26% and a peak gain of over 29% at a vocabulary size of 4,096. Section 5 provides a more detailed analysis of these results. Notably, we were able to match the baseline tokenizer's compression efficiency using a significantly smaller vocabulary of just 5,120 tokens – representing more than a 68% reduction in vocabulary size.

We also experimented with further optimization by analysing the dataset to identify the most frequently used identifiers (weighted by both frequency and character length), and explicitly adding them as atomic tokens to the vocabulary. However, this strategy did not lead to improved compression. A likely explanation is that the altered token composition disrupted the original character frequency distribution, which in turn affected the BPE merging decisions in a suboptimal way.

4.4. Transformer Language Model Training

Two transformer-based models for full-line Java code completion were trained, differing only in vocabulary size. The first model was trained with a vocabulary size of 16,384 tokens as this aligns with the vocabulary scale of current state-of-the-art completion systems [4]. The intention was to produce a fully open-source alternative capable of competitive full-line code completion performance, while remaining compatible with standard inference workflows. The second model was trained with a reduced vocabulary of 6,144 tokens.

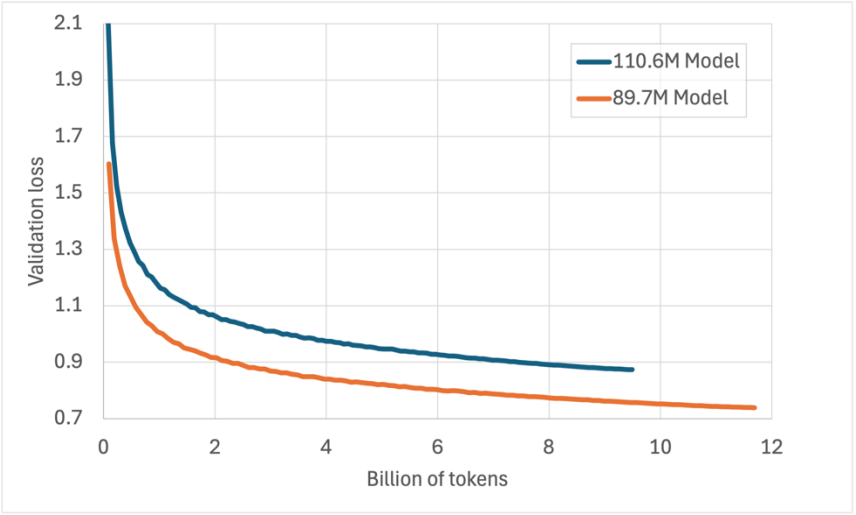


Fig. 66. Validation loss over tokens unseen during training for the 16,384-vocabulary 110.6M model and 6,144-vocabulary 89.7M model. The larger model was trained on less tokens as the tokenizer compresses the training dataset more efficiently.

The purpose was to investigate whether a substantially smaller vocabulary could still produce meaningful completions while benefiting from a smaller model size, lower memory footprint, and potentially faster inference on resource-constrained hardware. Although there was no strict mathematical reasoning for selecting 6,144 over other nearby values, this value was considered sufficiently small to test compression limits without being so restrictive that performance degradation would be inevitable.

Both models employ a transformer architecture inspired by the LLaMA family, with a maximum context length of 1,536 tokens. When paired with the llama.cpp inference engine [33], this architecture was shown to offer the most favourable balance of quality and speed for the target model size when compared to other evaluated architectures [4]. LLaMA builds on the GPT architecture, introducing several enhancements to training stability and performance, such as pre-normalization of inputs for each transformer sub-layer, replacement of the ReLU activation function with SwiGLU, and use of rotary positional embeddings in place of absolute embeddings, improving the handling of long sequence lengths [19].

After tokenization, the training dataset contained approximately 9.5B tokens for the 16,384-vocabulary model and 11.7B tokens for the 6,144-vocabulary model. Each token was used only once during training.

The larger model used a hidden dimension of 1,024, 8 attention heads, and 6 hidden layers, resulting in 110.6M parameters. The smaller model maintained the same architectural configuration, with the reduced vocabulary lowering the parameter count to 89.7M. Both were trained with the AdamW optimizer (learning rate = $1e^{-4}$, weight decay = 0.01, gradient clipping = 1.0), including 1,000 warmup steps. Training was conducted in mixed precision mode on an NVIDIA A10G GPU. The 16,384-vocabulary model required 15 days to train, while the 6,144-vocabulary model required 14 days. The models were periodically evaluated on a withheld portion of the dataset during training (Fig. 6).

Inference was implemented using the custom encoding/decoding algorithm to translate Java code into the Unicode-compatible tokens understood by the model and to map generated tokens back

into valid code. Decoding was performed using the stock beam search implementation from the Hugging Face transformers library [34].
Initial evaluation (Appendix A) showed that the 16,384-vocabulary model generated contextually relevant and meaningful completions. In contrast, the 6,144-vocabulary model struggled with several tasks that the larger model handled successfully, suggesting that the reduced vocabulary size may have significantly impacted accuracy. It remains possible that these limitations were compounded by the use of a standard beam search, and that a tailored inference strategy could yield improved results. Designing such a strategy was beyond the scope of the current work and is identified as a direction for future research.

5. Evaluation

The preprocessing algorithm introduced in Section 4.2 was applied to the full Java source code dataset. It filtered out syntactically incorrect code, removed all comments and annotations, standardized code formatting, and removed trailing and leading whitespace, as well as empty lines, reducing redundant tokens. This reduced the dataset size from approximately 85 GB (20 million files) to 66 GB (19.5 million files). This reduction not only decreased storage and processing requirements but also ensured that the tokenizer was trained on high-quality, semantically valid code. By eliminating noise and enforcing consistent formatting, the algorithm improved the likelihood of learning meaningful and reusable tokens during BPE training.
To measure tokenization efficiency, both the baseline tokenizer and the proposed two-stage tokenizer were trained and evaluated on identical dataset samples. The evaluation was carried out across a range of vocabulary sizes, from 16,384 down to 4,096 tokens. For each vocabulary size, the total number of output tokens produced from the same dataset was recorded. Fig. 7 compares the results between the two tokenizers, while Table 1 presents the statistics in more detail and associates the number of the downstream language model parameters to each vocabulary size.

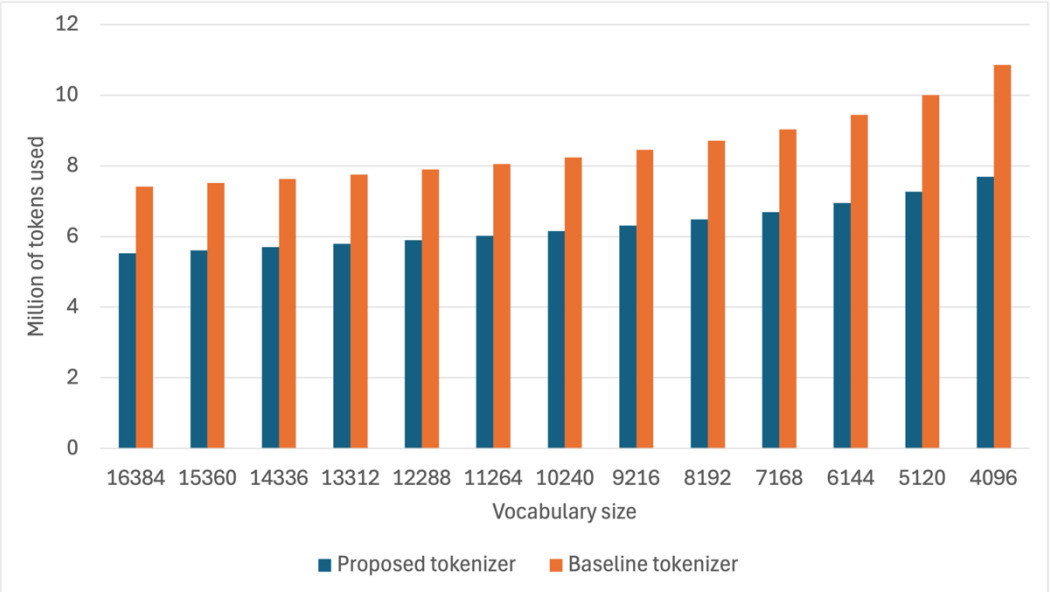


Fig. 77. Comparison of token compression efficiency at different vocabulary sizes (lower value for tokens used is better). Our two-stage tokenizer and the baseline tokenizer were re-trained and tested for each of the listed vocabulary sizes using the same dataset sample.

Table 1. Comparison of token counts, compression, and corresponding model size between the baseline tokenizer and our two-stage tokenizer

Vocab	Baseline tokens	Our tokens	Reduction	Model size
16384	7,409,241	5,528,145	25.39%	110.6M
15360	7,513,196	5,609,179	25.34%	108.5M
14336	7,627,842	5,697,259	25.31%	106.4M
13312	7,755,074	5,793,822	25.29%	104.4M
12288	7,897,171	5,900,620	25.28%	102.3M
11264	8,057,311	6,019,780	25.29%	100.2M
10240	8,240,316	6,153,913	25.32%	98.1M
9216	8,454,322	6,306,592	25.4%	96.0M
8192	8,711,799	6,482,707	25.59%	93.9M
7168	9,030,723	6,692,222	25.89%	91.8M
6144	9,442,470	6,947,808	26.42%	89.7M
5120	10,002,903	7,267,332	27.35%	87.6M
4096	10,857,776	7,691,129	29.16%	85.5M

Across all vocabulary sizes, the proposed tokenizer consistently produced fewer tokens than the baseline. At the largest vocabulary size, the reduction was 25.39% with an average of 6.90 tokens per line of code, and at the smallest size, the reduction remained substantial at 29.16% with an average of 8.48 tokens per line. Since the number of parameters in a transformer-based language model's embedding layer grows linearly with vocabulary size, a smaller vocabulary not only impacts compression but also affects the overall model size.

To assess the downstream effects of tokenization, two language models were trained with different vocabulary sizes (Section 4.4). The inference performance was tested on a CPU, which is the intended deployment platform for the local model. The 16k vocabulary model's generation speed ranged between 23.3–32.4 tokens/sec with RAM usage between 470 MB – 530 MB, depending on the context length of the tested input sequence and the complexity of the generated sequence. The 6k vocabulary model achieved speeds between 16.8–34.5 tokens/sec, with memory consumption between 420 MB – 490 MB. The smaller vocabulary size consistently reduced RAM usage by roughly 40–50 MB compared to the larger model, even with the same architecture. While generation speed was not strictly monotonic with vocabulary size, performance was competitive across all testing scenarios. In some runs, the smaller-vocabulary model achieved higher tokens/sec than the larger-vocabulary equivalent. A recent study [35] revealed that developers expect completion results to be generated within 200 ms to be considered satisfactory. Given the data we collected, the larger model would take between 213-296 ms on average to generate full-line completion. For the smaller model this would be between 246-505 ms on average. However, these values are based on the models running without quantization. Generation speed has been shown to improve substantially for quantized models deployed with an efficient inference engine, such as llama.cpp [4]. Decreasing the time it takes to generate a token provides an opportunity to explore more candidate suggestions during beam search, which can lead to more accurate completions while keeping the overall full-line completion speed under 200 ms.

Appendix A shows a few testing scenarios for code completion. It should be noted that while the 6,144-vocabulary size model produced less meaningful completions under stock Hugging Face beam-search inference, it may not necessarily indicate a fundamental flaw in the model, as the inference algorithm was not optimised for this vocabulary configuration. A tailored decoding strategy might yield better results.

6. Limitations

A direct comparison with the baseline full-line completion model was not feasible, as its exact training data preparation and prompt formatting were unknown, causing it to produce incoherent outputs when prompted with the same test cases.
The models developed in this work were not evaluated for completion accuracy or benchmarked against other open-source code completion models due to the absence of a standardized evaluation framework for Java full-line completion at the time of research. As a result, while training and validation loss provide useful signals, the practical effectiveness of the models in real-world developer workflows remains unverified.
Additionally, the lack of quantization for the proposed models means that their potential memory efficiency and inference speed gains under quantized inference remain unexplored.
All inference tests on our models used the default beam-search decoding from the Hugging Face transformers library. This may not be optimal, particularly for the reduced vocabulary model, where a custom decoding strategy could potentially improve output quality.
Finally, hyperparameters were tuned manually rather than through systematic search, leaving open the possibility of further performance gains with optimized configurations.

7. Future Work

While this project demonstrates clear benefits from grammar-aware tokenization and streamlined model training pipelines, there are several promising directions for future research:

1) Extending grammar-aware tokenization to other programming languages, such as Python and C++, would validate its generality and open new opportunities for multi-language code completion models. There is also potential for incorporating these methods into multi-line code completion and code generation tasks, where token efficiency could yield even greater speed and accuracy improvements.
2) Combining grammar-based methods with efficient BPE algorithms like SentencePiece or YouTokenToMe could further improve model inference speed.
3) Developing tailored inference algorithms for choosing optimal candidates could improve the accuracy and relevance of completions.
4) Further work on quantization, pruning, and low-rank adaptation could make local deployment viable on ultra-low-resource devices, expanding access to on-device AI assistants.
5) Establishing standardized open benchmarks for full-line code completion for Java and other programming languages would allow continuous comparison of models, ensuring sustained relevance and encouraging contributions from the wider community.

8. Conclusion

This work set out to address the limitations of current code completion solutions, which are often proprietary, cloud-based, and inaccessible to developers who require privacy-preserving, offline alternatives. By focusing on Java full-line code completion, we developed a complete open-source workflow encompassing data preprocessing, efficient tokenization, and model training pipelines suitable for deployment on low-end hardware. We achieved this through three major contributions:

1) A scalable and reproducible data preprocessing pipeline that cleans, filters, and formats large-scale Java code datasets. Applied to an 85 GB raw dataset containing 20 million files, this pipeline reduced the size to 66 GB and 19.5 million files, preserving 97.5% of relevant content while eliminating low-quality or redundant data.
2) A novel grammar-aware tokenization strategy that leverages Java language structure to improve token compression efficiency. This approach achieves an average 26% improvement in compression compared to current state-of-the-art tokenizers, enabling significantly more code to be represented within the same context window.
3) A model training pipeline that integrates these optimizations to produce smaller, faster transformer-based models for Java code completion, enabling deployment on devices with limited resources.

The combination of these advances brings multiple downstream benefits. Better context utilization allows models to consider more surrounding code when generating completions, increasing their relevance and correctness. Improved inference speed and reduced memory footprint make the models accessible to a wider range of users and hardware configurations. And open-source availability ensures transparency, reproducibility, and extensibility by the broader research and developer communities. These contributions directly advance the accessibility and sustainability of local language models for code completion, offering a privacy-preserving, cost-effective alternative to commercial cloud-based solutions.

ACKNOWLEDGEMENTS

I would like to thank my industry supervisor Jonathan Halliday for his support and guidance throughout the project. I would also like to extend my gratitude to my academic supervisor Paul Ezhilchelvan and the faculty deliver lead Marc Milgrom for providing me with computing resources I needed to train the language model.

REFERENCES

[1] S. Amann, S. Proksch, S. Nadi, and M. Mezini, ‘A Study of Visual Studio Usage in Practice’, in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2016, pp. 124–134. doi: 10.1109/SANER.2016.39.

[2] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, ‘Fast and Memory-Efficient Neural Code Completion’, Mar. 16, 2021, *arXiv*: arXiv:2004.13651. doi: 10.48550/arXiv.2004.13651.

[3] G. A. Aye and G. E. Kaiser, ‘Sequence Model Design for Code Completion in the Modern IDE’, Apr. 10, 2020, *arXiv*: arXiv:2004.05249. doi: 10.48550/arXiv.2004.05249.

[4] A. Semenkin *et al.*, ‘Full Line Code Completion: Bringing AI to Desktop’, Jan. 08, 2025, *arXiv*: arXiv:2405.08704. doi: 10.48550/arXiv.2405.08704.

[5] ‘Code completions with GitHub Copilot in VS Code’. Accessed: May 28, 2025. [Online]. Available: https://code.visualstudio.com/docs/copilot/ai-powered-suggestions

[6] ‘What is CodeWhisperer? - CodeWhisperer’. Accessed: May 28, 2025. [Online]. Available: https://docs.aws.amazon.com/codewhisperer/latest/userguide/what-is-cwspr.html

[7] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, ‘IntelliCode Compose: Code Generation Using Transformer’, Oct. 29, 2020, *arXiv*: arXiv:2005.08025. doi: 10.48550/arXiv.2005.08025.

[8] G. Dagan, G. Synnaeve, and B. Rozière, ‘Getting the most out of your tokenizer for pre-training and domain adaptation’, Feb. 07, 2024, *arXiv*: arXiv:2402.01035. doi: 10.48550/arXiv.2402.01035.

[9] C. Wei, Y.-C. Wang, B. Wang, and C.-C. J. Kuo, ‘An Overview on Language Models: Recent Developments and Outlook’, *APSIPA Trans. Signal Inf. Process.*, vol. 13, no. 2, 2024, doi: 10.1561/116.00000010.

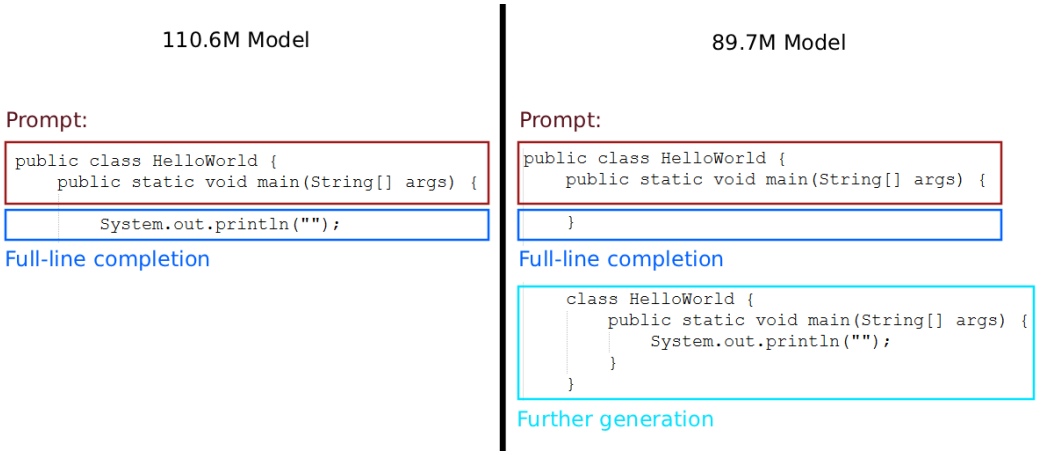
[10] P. Kłosowski, ‘Deep Learning for Natural Language Processing and Language Modelling’, in *2018 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, Sept. 2018, pp. 223–228. doi: 10.23919/SPA.2018.8563389.

[11] A. Vaswani *et al.*, 'Attention Is All You Need', Aug. 02, 2023, *arXiv*: arXiv:1706.03762. doi: 10.48550/arXiv.1706.03762.
[12] Z. Zheng *et al.*, 'Towards an Understanding of Large Language Models in Software Engineering Tasks', Dec. 10, 2024, *arXiv*: arXiv:2308.11396. doi: 10.48550/arXiv.2308.11396.
[13] X. Amatriain, A. Sankar, J. Bing, P. K. Bodigutla, T. J. Hazen, and M. Kazi, 'Transformer models: an introduction and catalog', Mar. 31, 2024, *arXiv*: arXiv:2302.07730. doi: 10.48550/arXiv.2302.07730.
[14] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, 'A Systematic Evaluation of Large Language Models of Code', May 04, 2022, *arXiv*: arXiv:2202.13169. doi: 10.48550/arXiv.2202.13169.
[15] S. J. Mielke *et al.*, 'Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP', Dec. 20, 2021, *arXiv*: arXiv:2112.10508. doi: 10.48550/arXiv.2112.10508.
[16] A. Thawani, S. Ghanekar, X. Zhu, and J. Pujara, 'Learn Your Tokens: Word-Pooled Tokenization for Language Modeling', Oct. 17, 2023, *arXiv*: arXiv:2310.11628. doi: 10.48550/arXiv.2310.11628.
[17] R. Sennrich, B. Haddow, and A. Birch, 'Neural Machine Translation of Rare Words with Subword Units', June 10, 2016, *arXiv*: arXiv:1508.07909. doi: 10.48550/arXiv.1508.07909.
[18] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, 'Language Models are Unsupervised Multitask Learners', *OpenAI Blog*, 2019.
[19] H. Touvron *et al.*, 'LLaMA: Open and Efficient Foundation Language Models', Feb. 27, 2023, *arXiv*: arXiv:2302.13971. doi: 10.48550/arXiv.2302.13971.
[20] D. Fried *et al.*, 'InCoder: A Generative Model for Code Infilling and Synthesis', Apr. 09, 2023, *arXiv*: arXiv:2204.05999. doi: 10.48550/arXiv.2204.05999.
[21] R. Li *et al.*, 'StarCoder: may the source be with you!', Dec. 13, 2023, *arXiv*: arXiv:2305.06161. doi: 10.48550/arXiv.2305.06161.
[22] B. Rozière *et al.*, 'Code Llama: Open Foundation Models for Code', Jan. 31, 2024, *arXiv*: arXiv:2308.12950. doi: 10.48550/arXiv.2308.12950.
[23] E. Nijkamp *et al.*, 'CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis', Feb. 27, 2023, *arXiv*: arXiv:2203.13474. doi: 10.48550/arXiv.2203.13474.
[24] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, 'CodeGen2: Lessons for Training LLMs on Programming and Natural Languages', July 11, 2023, *arXiv*: arXiv:2305.02309. doi: 10.48550/arXiv.2305.02309.
[25] 'bigcode/starcoderdata · Datasets at Hugging Face'. Accessed: July 31, 2025. [Online]. Available: https://huggingface.co/datasets/bigcode/starcoderdata
[26] 'bigcode/the-stack · Datasets at Hugging Face'. Accessed: July 31, 2025. [Online]. Available: https://huggingface.co/datasets/bigcode/the-stack
[27] *javaparser/javaparser*. (May 30, 2025). Java. JavaParser. Accessed: May 31, 2025. [Online]. Available: https://github.com/javaparser/javaparser
[28] 'Tokenizers'. Accessed: Aug. 04, 2025. [Online]. Available: https://huggingface.co/docs/tokenizers/index
[29] T. Kudo and J. Richardson, 'SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing', Aug. 19, 2018, *arXiv*: arXiv:1808.06226. doi: 10.48550/arXiv.1808.06226.
[30] *VKCOM/YouTokenToMe*. (July 24, 2025). C++. VK.com. Accessed: Aug. 05, 2025. [Online]. Available: https://github.com/VKCOM/YouTokenToMe
[31] 'ANTLR'. Accessed: Aug. 04, 2025. [Online]. Available: https://www.antlr.org/
[32] 'GitHub - antlr/grammars-v4: Grammars written for ANTLR v4; expectation that the grammars are free of actions.' Accessed: Aug. 04, 2025. [Online]. Available: https://github.com/antlr/grammars-v4
[33] *ggml-org/llama.cpp*. (Aug. 06, 2025). C++. ggml. Accessed: Aug. 07, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp
[34] 'Transformers'. Accessed: Aug. 07, 2025. [Online]. Available: https://huggingface.co/docs/transformers/en/index
[35] C. Wang *et al.*, 'Practitioners' Expectations on Code Completion', Jan. 10, 2023, *arXiv*: arXiv:2301.03846. doi: 10.48550/arXiv.2301.03846.

APPENDIX

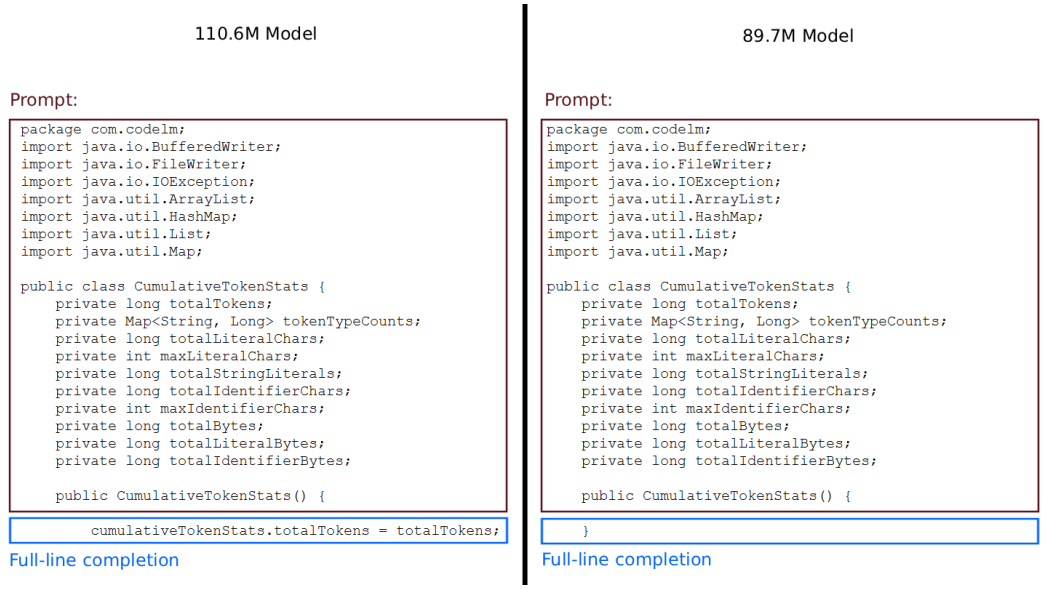
A Model Generation Tests

A comparison between generated results for both models. The prompt is a beginning of a class named HelloWorld. The intention is for the model to generate a print statement with an empty string instead of “Hello World”, as string literal were not part of the training set. The 110.6M model outputs a meaningful completion, the 89.7M outputs just a closing curly brace. However, if we let the model generate more than a single line, it then recreates the class and adds the print statement. This further suggests that with further inference algorithm optimizations, the smaller model could be able to output the correct completion too.



Example generation test

The second task is more complicated, as the context in the prompt is longer. We give a beginning of a class CumulativeTokenStats, where we define a set of variables. The intention here is for the model to generate statements for initializing those variables in the constructor. The larger model correctly initializes the first variable totalTokens. The smaller model only outputs a closing curly brace.



Example generation test 2

B Token Type Distribution

The complete data for the dataset token type analysis. This is supplementing Fig. 4. The total number of individual tokens in the dataset is 27,953,373,175.

Token type	Count	Percentage
Whitespace	14,550,275,542	52
Separator	5,479,967,657	20
Identifier	3,926,216,195	14
End-of-line	1,686,285,013	6
Keyword	1,182,958,366	4
Operator	671,540,185	2
Literal	453,777,293	2