Bachelor's thesis

PROJECT ADAPTATION IN CODE COMPLETION VIA IN-CONTEXT LEARNING

Maksim Sapronov

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Project adaptation in code completion via in-context learning
Student:	Maksim Sapronov
Supervisor:	Evgenii Glukhov, M.Sc.
Study program:	Informatics
Branch / specialization:	Artificial Intelligence 2021
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2025/2026

Instructions

In recent years, advancements in the fields of code and natural language processing have significantly transformed code completion systems, making them more accurate and capable of assisting with software development tasks. Some recent Large Language Models (LLMs) and Code LLMs can utilize an entire software project. However, most open source pre-trained models can process less than 5% of the project's code files at once. Such a restricted context length of Code LLMs limits their ability to comprehend the structure of a real-world software project fully. As a result, models often struggle to integrate information spread across multiple files, such as dependencies, class hierarchies, and external library usage, leading to outputs that lack project-wide coherence.

It is known that LLMs have an ability for in-context learning, i.e. it can adapt to novel tasks if there are some examples in the context. In particular, for the code completion task, some of the project files are included in the context as examples of completion. It is common practice now to include a repository-level training step in the pre-training of Code LLMs. However, some open questions in this domain are the main focus of this research project.

Research Questions:

For pre-trained Code LLMs with no context window extension:
 1. Does the improvement of the quality of code completion depend on a context

---

composition approach for in-context learning? Estimate the influence.

2. Does fine-tuning help to improve the quality of code completion for a particular context composer?

For pre-trained Code LLMs with context window extension:

1. Are there any context composition techniques that don't impact in-context learning abilities?

2. Does the improvement of the quality of code completion depend on a context composition approach for the repository-level training step? Estimate the influence.

Guidelines for Elaboration:

1. Conduct a survey of the existing papers on the following topics: in-context learning, code completion models, repository-level code completion.

2. Developing a Context Composition Framework for Repository-level Code Completion Task

- Design and implement a framework for extracting a context from a repository, include language-specific processing for code files.

3. Investigating the Impact of Context Composition on Code Completion

- Analyze how variations in context composition (from 2) and context length influence the quality of one-line code completion for a pre-trained Code LLM.

4. Developing a Fine-Tuning Pipeline

- Design and implement a pipeline for the complete fine-tuning of a pre-trained Code LLM.

5. Evaluate Context Composition Methods on Fine-tuned Model

- Base model is https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-base

- The dataset for the fine-tuning is available to company staff at JetBrains through internal resources. It contains data from open-source repositories on GitHub.

6. Extend Model's Context Length with Repository-level Training

- Base model is https://opencoder-llm.github.io

7. Compare Context Extensions for Different Context Composers

- Dataset for the fine-tuning is available to company staff at JetBrains through internal resources. It contains data from open-source repositories on GitHub.

- For evaluation use the approach that is suggested in https://arxiv.org/abs/2406.11612

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (license) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorization is unlimited in time, territory and quantity.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

In Prague on May 16, 2025

Abstract

This thesis addresses the challenge of enhancing code completion models with repository-level context awareness. Modern completion systems struggle with information dispersed across large codebases, limiting their performance. The work presents a context composition framework that extracts relevant repository information and a fine-tuning pipeline for model adaptation, evaluated through systematic experimentation. The research demonstrates that context selection strategy significantly impacts completion quality during inference, while repository-level pre-training preserves in-context learning capabilities. Notably, the study demonstrates that computational requirements for context window extension can be substantially reduced while maintaining competitive performance, advancing code completion by enabling better integration of project-wide information.

Keywords   repository-level code completion, project adaptation, in-context learning, long context, context extension, resource efficiency, Transformer, Code LLM

Home page: `https://github.com/sapromak/adaptive-code-completion`

Abstrakt

Tato bakalářská práce se zabývá problematikou vylepšení modelů pro dokončování kódu s povědomím o kontextu na úrovni repozitáře. Moderní systémy pro dokončování kódu mají potíže s informacemi rozptýlenými napříč rozsáhlými kódovými základnami, což omezuje jejich výkonnost. Práce představuje framework pro kompozici kontextu, který extrahuje relevantní informace z repozitářů, a pipeline pro dolaďování modelů, hodnocené prostřednictvím systematického experimentování. Výzkum ukazuje, že strategie výběru kontextu významně ovlivňuje kvalitu dokončování během inference, zatímco předtrénování na úrovni repozitáře zachovává schopnosti učení v kontextu. Studie zejména prokazuje, že výpočetní požadavky pro rozšíření kontextového okna lze výrazně snížit při zachování konkurenceschopného výkonu, čímž posouvá dokončování kódu vpřed umožněním lepší integrace informací v rámci celého projektu.

Klíčová slova dokončování kódu na úrovni repozitáře, projektové přizpůsobení, učení v kontextu, dlouhý kontext, rozšíření kontextu, účinné využívání zdrojů, Transformer, Code LLM

Domovská stránka: `https://github.com/sapromak/adaptive-code-completion`

Contents

List of Figures

List of Tables

List of abbreviations

ABF	Adjustment of the Base Frequency
API	Application Programming Interface
Adam	Adaptive Moment Estimation
BPE	Byte Pair Encoding
CE	Cross-Entropy
Code LLM	Code Large Language Model
EM	Exact Match
EOS	End of Sequence
ES	Edit Similarity
FIM	Fill-in-the-Middle
FL	File-Level
GPU	Graphics Processing Unit
ICL	In-Context Learning
IoU	Intersection over Union
LCA	Long Code Arena
LLM	Large Language Model
LM	Language Modeling
MLP	Multi-Layer Perceptron
OOV	Out-of-Vocabulary
Or	Original
PD	Path Distance
PPL	Perplexity
RAG	Retrieval-Augmented Generation
RoPE	Rotational Position Embedding

Introduction

Software development plays a pivotal role in shaping the modern digital world. Assisting developers in writing code more efficiently has the potential to accelerate innovation across industries by reducing the cognitive and temporal overhead of programming. Over the decades, numerous tools have emerged to support developers, including high-level programming languages, integrated development environments, version control systems, and, more recently, AI-powered code assistance. Tasks such as code completion, generation, refactoring, and bug localization are increasingly addressed using Large Language Models (LLMs), which have demonstrated strong capabilities in natural language and code understanding.

Code completion, in particular, benefits significantly from the in-context learning capabilities of LLMs—where the model leverages examples or relevant information provided in the input context to improve predictions without parameter updates. However, while LLMs have advanced in modeling local contexts, a critical limitation remains: their restricted ability to integrate and reason over information dispersed across large codebases. This includes understanding dependencies between files, class hierarchies, and interactions with external libraries—information that is often essential for generating coherent and accurate completions.

Despite the emergence of code-specific LLMs and efforts to incorporate repository-level context during training or inference, most pre-trained models still struggle to process more than a small fraction of a project's code files at once. As a result, they fall short in capturing project-wide structure, leading to incomplete or inaccurate predictions. Therefore, there is a need for continued research within the community developing such models to address these limitations.

This thesis contributes to the field by conducting a set of comprehensive experiments and providing tools used to achieve this: a context composition framework for constructing repository context and a fine-tuning pipeline for adapting pre-trained Code Large Language Models (Code LLMs) to these compositions. The research results reveal that repository context significantly

impacts completion quality during inference, and the DeepSeek-Coder-Base 1.3B model exhibits minimal effects from context-specific fine-tuning, indicating robust initial training. Furthermore, context extension maintains the in-context learning capabilities developed previously, while repository context has a marginal influence on the outcomes of context extension. In addition, notable observation is made that repository-level pre-training can achieve competitive results with significantly fewer resources, utilizing 73 million tokens compared to billions. These findings position the thesis at the intersection of code completion and project adaptation, addressing the knowledge gap in this area.

The thesis is organized as follows: Chapter 1 offers an overview of the code completion task, including its definition, significance, historical approaches, and taxonomy. Chapter 2 introduces essential concepts related to Language Modeling (LM) and state-of-the-art architecture for this domain within machine learning. In Chapter 3, the discussion on language modeling is extended to encompass code completion with relevant details. Chapter 4 covers the aspects of in-context learning, which is widely adopted in code-related tasks. The implementation of the tools used to answer the research questions is elaborated in Chapter 5. Finally, Chapter 6 describes the research, including precise question formulations, experimental design, and results interpretation.

Objectives of the Thesis

The objectives of this thesis are divided into two parts, corresponding to the theoretical and practical aspects of the work.

The theoretical part aims to provide a comprehensive and high-level survey of the current state of the field related to project adaptation in the context of full line code completion, focusing on the capabilities and limitations of in-context learning and repository-level code completion methods. This part establishes the necessary background to understand the practical contributions of the thesis.

The practical part addresses several research questions concerning the impact of various context composition strategies on the quality of code completion, evaluated across three setups: a pre-trained Code LLM, the same model fine-tuned with a specific context composition strategy, and the model after context window extension. To address these questions, the thesis describes a context composition framework for extracting relevant information from software repositories, as well as a fine-tuning pipeline for adapting Code LLMs to project-specific data. The outcomes characterize the role of context composition in enhancing code completion quality and provide insights into its integration within repository-level training procedures.

Part I

Conceptual Framework

Chapter 1

Code Completion

The upcoming chapter provides a comprehensive examination of code completion. It begins with a definition of the task, then delves into the motivations for advancing code completion techniques, highlighting their impact on productivity and learning. Finally, the categorization of code completion based on various criteria is presented.

1.1 Task Definition

Code completion, also called *code suggestion* or *autocompletion*, is a functionality within integrated development environments that enables developers to utilize an automated continuation of the code they are writing. Typically, it is invoked by a trigger model, which aims to anticipate the user's requirement for this functionality. The point of this invocation, whether it is referenced as temporal or spatial, is called a *trigger point*.

In this work, code completion is regarded as a task rather than a feature, as this perspective highlights the task-methodology relationship and emphasizes the technical aspects over marketing considerations. In addition, the term *completion* is used as a shorthand throughout the text.

1.2 Motivation

The task of code completion is a pivotal component of contemporary software development, offering substantial benefits that enhance both the productivity and efficiency of developers. This section elucidates the motivations for investigating this task and advancing existing solutions.

By reducing the amount of typing required, code completion allows developers to focus more on the logic and structure of their code rather than the syntax. Studies have shown that AI-powered code completion tools can significantly reduce task execution times, with some reporting up to a 55.8% reduc-

tion in controlled experiments (Peng et al. 2023). These tools also significantly impact developer productivity by reducing the cognitive load associated with coding tasks, achieved through contextually relevant suggestions that streamline the coding process (Weber et al. 2024). Furthermore, the integration of AI in code completion tools has been linked to increased developer satisfaction and efficiency, as it allows developers to complete tasks more swiftly and with greater accuracy (Bakal et al. 2025).

For students and novice programmers, code completion tools serve as a valuable learning aid. They provide real-time feedback and suggestions, helping learners understand programming concepts and syntax more effectively. This educational aspect is highlighted in studies where students reported enhanced learning experiences and increased confidence in their coding abilities (Takerngsaksiri et al. 2023). Moreover, code completion tools provide accurate suggestions that help reduce typo errors and other common mistakes. This is particularly beneficial for new developers or those working with unfamiliar codebases, as it helps them adhere to coding standards and best practices.

Recent advancements in deep learning have enabled the personalization of code completion tools, allowing them to adapt to specific organizational or individual coding styles. This personalization not only improves the relevance of suggestions but also enhances the overall user experience, making these tools more effective and user-friendly (Giagnorio et al. 2025).

Due to the simple and atomic formulation of code completion, it possesses the important property of being fundamental to other AI-powered code assistance features. For instance, code generation can be viewed as a specific extension of code completion, and code editing is a sequence of code generations. This implies that most enhancements achieved through code completion research propagate further, adding new layers of improvements in the field.

In summary, code completion stands as a foundational and multifaceted tool within modern software development. Its benefits extend beyond mere convenience, offering measurable improvements in efficiency, learning, and code quality. As advancements in AI continue to refine its capabilities, code completion not only streamlines day-to-day programming tasks but also acts as a cornerstone for broader innovations in AI-assisted software engineering.

1.3 Taxonomy of Code Completion

The concept of code completion lacks a strict definition within the field, as its interpretation has evolved with the various approaches employed to address it over time (see the Related Work section of the Ciniselli et al. (2021) for more details). This section provides a comprehensive overview of the diverse characteristics of code completion and highlights the specific focus of this thesis.

To grasp the meaning of the subsequent text, it is crucial to understand the term *token*, which refers to any subsequence of characters with a finite length. The utility of this term is further justified in Section 2.2.1.

1.3.1 Granularity

Completion can be applied under the different degrees of granularity, which refers to the scope and detail of the produced code. At the most basic level, *next-token completion* involves predicting the subsequent token, such as a keyword, operator, or identifier. This level of granularity is useful for fine-grained suggestions that assist developers in writing code efficiently. Moving up in complexity, *single line completion* involves predicting the continuation of the given line of code based on the current context, which requires a broader understanding of the code's structure and logic. At the highest level, *code block completion* involves generating entire blocks of code, such as functions or classes, which necessitates a comprehensive understanding of the codebase and its architecture. This level of granularity is akin to code generation, where the model not only completes existing code but also creates new, coherent code structures. Each level of granularity serves different purposes and can be leveraged depending on the specific needs of the development task at hand.

For this work, single-line completion is selected because it presents a greater challenge for modern models compared to next-token prediction and serves as a fundamental baseline task for evaluating the performance of the proposed methods.

1.3.2 Context

In the realm of code completion, a distinction is drawn between file-level and repository-level approaches, each characterized by its scope and contextual depth. *File-level code completion* operates within the confines of a single file, leveraging the immediate context such as local variable declarations, function definitions, and imports. This approach is effective for small, isolated scripts but is limited in its ability to capture the broader interactions present in larger projects. Conversely, *repository-level code completion* extends its reach to encompass the entire codebase, integrating information from multiple files and modules. This broader context is essential for understanding complex dependencies and interactions that span across the repository, enabling more accurate and contextually relevant code suggestions. The need for greater context in repository-level completion is driven by the intricate and interconnected nature of modern software systems, where a comprehensive understanding of the entire codebase is crucial for effective code completion.

Repository-level completion has been selected as the focus of this thesis because it offers a broader research landscape and addresses a significant need within the field.

1.3.3 Suffix Awareness

The completion scenario can be categorized based on the directionality of the prediction relative to the trigger point. The *left-to-right* scenario involves generating code predictions using only the context preceding the trigger point, which is typical in traditional code completion systems. This approach is effective for straightforward code continuations but may lack the ability to consider the broader context. Conversely, the *left-and-right* scenario, also known as *bidirectional* completion, leverages both the preceding and succeeding context around the trigger point. This method allows for more informed predictions by considering the entire file, thus enhancing the accuracy and relevance of the suggestions.

Modern models for code completion employ the Fill-in-the-Middle (FIM) approach to gain suffix awareness capability. Although this thesis describes this method in Section 3.3, it does not expand on it and instead focuses on the more fundamental left-to-right scenario. This choice is driven by the practical part of this work and its design constraints, which are motivated in Section 6.2.1.

1.3.4 Line Start Availability

Another criterion for categorizing completion systems is whether the initial portion of the target line has already been completed by the user. In practical applications, this scenario is quite common. However, for the purpose of model evaluation, it is more straightforward to assess the *full line completion*, where the initial part of the target line is not provided. This approach serves as a lower bound for assessing completion quality, as the ability to predict the entire line demonstrates a stronger capability.

Due to the aforementioned reasoning, the full line code completion is further considered throughout this thesis.

1.3.5 Programming Language Support

The code completion systems can support either a single programming language or multiple languages simultaneously.

In the practical part of this thesis, multilingual models are employed with a focus on a single programming language, specifically Python[1]. This design choice is motivated by resource constraints.

To finalize the terminology used in subsequent parts of this thesis regarding this task, the *completion* refers to a full, single-line, repository-level code completion without suffix awareness.

---

[1] https://www.python.org/

Chapter 2

Standard LM

This chapter provides the necessary theoretical foundation to understand general language modeling, without specifically focusing on the code completion task. It begins with the formulation and presents a probabilistic perspective on the models employed to represent language. Subsequently, it explores the Transformer architecture and the training of such models. The chapter concludes with a discussion on the inference and sampling processes of trained models.

In this chapter, the term *language* refers to an arbitrary natural language, as opposed to a formal one.

2.1 Definition

Consider a natural language denoted as $\mathcal{L}$, which is a set of all possible sentences in this language. A sentence $\boldsymbol{x}_{1:T}$ of length T is a sequence of words $(x_1, x_2, \ldots, x_T)$, where each word is an element of the *vocabulary V*. Given that some sentences are more likely to occur than others, there exists a joint probability distribution $P : \mathcal{L} \to [0,1]$ over variable-length sequences from $\mathcal{L}$. The model p_θ with parameters θ (also called *weights*) that represents this distribution is called a *language model*, and the task of creating such a model is referred to as *language modeling*.

Although natural languages are infinite due to their recursive structure, they have a finite number of words, meaning $K = |V| \in \mathbb{N}$. To demonstrate this, consider that word lengths are finite, which implies that $\mathcal{L}$ certainly contains a word with the maximum length M. Let the number of characters involved in word formation be N. A very rough upper bound estimate for the number of words in $\mathcal{L}$ can be calculated using the formula N^M, which is finite because both N and M are finite.

The chain rule of probability can be applied to represent P as follows:

$$P(\boldsymbol{x}_{1:T}) = P(x_1) \cdot P(x_2 \mid x_1) \cdot P(x_3 \mid x_2, x_1) \cdots = \prod_{t=1}^{T} P(x_t \mid \boldsymbol{x}_{1:t-1}) \quad (2.1)$$

This indicates that language modeling is equivalent to estimating the probability of each word in the sentence given all the preceding words (i.e., the *context*), where the probability distribution over each word is a K-dimensional categorical distribution. This sequential prediction process, where each word depends only on the prior subsequence, is called *autoregressive language modeling*.

2.2   Text Representation

The proper representation of text is critical to effective language modeling design. Treating text as a sequence of words is natural and intuitive for humans, but not the best choice for machines.

2.2.1   Tokens

The first reason why words may not be the optimal choice for representing discrete units of text is the Out-of-Vocabulary (OOV) problem, which refers to the presence of words (or tokens) that are absent from a language model's vocabulary. This challenge arises from the inherently dynamic nature of language, which lacks fixed boundaries. As language evolves, some words become obsolete while new ones emerge. Additionally, the text corpora used for training language models often contain typographical errors and other artifacts of human language.

The second reason is that the language models require statistical information about the co-occurrence of words to capture the semantics of the language. It becomes a problem with rare words whose representations in models lack a learning signal. This phenomenon can be lucidly demonstrated with Zipf's law, which states that in a given corpus, the frequency of any word is inversely proportional to its rank in the frequency table (Estoup 1916). This implies that a small number of words are used very frequently, while the majority are rare. Consequently, language models trained on word-level representations may struggle to learn these infrequent words, leading to poor generalization.

One can propose employing character-level text representation. Although this approach completely resolves the OOV problem, it introduces its own limitations. A small vocabulary necessitates processing longer sequences with smaller semantic representation of its units, which introduces additional computational overhead to internally combine these units to form a more integral semantic representation of text.

Thus, there is a trade-off between the size of the vocabulary and the size of input sequences. Minimizing one leads to an increase in the other. A large

vocabulary means that some of its units are used very rarely, and their representations lack a learning signal. A small vocabulary means the utilization of more semantically poor units and a greater computational overhead to process longer sequences.

The optimal balance between character-level and word-level representations is subword chunks, generally called *tokens*. They offer a greater prospect of granularity in text division, which allows addressing the aforementioned trade-off on a more fine-grained level.

2.2.2 Byte Pair Encoding for Tokenization

The OOV problem persists at the token level. A variety of methods exist to address this issue, with one of the most popular being the Byte Pair Encoding (BPE) algorithm. Originally proposed by Gage (1994) as a compression algorithm, it was later adapted by Sennrich et al. (2015) for word segmentation with the aim of constructing a vocabulary.

The BPE process begins by initializing the vocabulary with individual characters and a special end-of-word symbol. Words are initially represented as sequences of these characters. Given some corpus of text, the algorithm iteratively identifies the most frequent pair of adjacent symbols and merges them into a new symbol. This process continues until a predefined number of merge operations is reached, resulting in a vocabulary composed of both characters and frequently occurring subword units. After the vocabulary is constructed, the same algorithm can be applied to tokenize the text.

This approach allows for a compact representation of text, reducing the sequence length while maintaining the ability to represent rare and unseen words. By balancing the trade-off between vocabulary size and sequence length, BPE enables language models to efficiently handle the dynamic nature of language, capturing both common and rare linguistic patterns.

2.2.3 Embeddings

After the text is tokenized, it is represented as a sequence of tokens $(x_1, x_2, \ldots, x_T)$, where $x_i \in \{1, 2, \ldots, K\}$ is a corresponding integer identifier. A parameterized language model p_θ should have the ability to process this sequence of integers. Since most effective models are based on Neural Networks, θ is a set of matrices that define some non-linear transformation over the continuous vector space. Therefore, tokens need to be embedded into the input vector space $\mathbb{R}^d$.

The most straightforward approach is to set $d = K$ and use *one-hot encoding*, which assigns a vector $\mathbf{x}_i = \text{one-hot}(x_i) \in \{0, 1\}^K$ full of zeros except for the position corresponding to the token, which is set to one. The resulting vectors capture the identity of the tokens but lack the interaction effects between them.

This issue is resolved by incorporating the first layer of the neural network as a trainable embedding matrix $\mathbf{E} \in \mathbb{R}^{d' \times K}$, which maps each token to a vector $\mathbf{E}\mathbf{x}_i$ in $\mathbb{R}^{d'}$, called an *embedding*, by multiplying its one-hot encoding with $\mathbf{E}$. Note that the sparse nature of one-hot representations allows this multiplication to be implemented using a lookup table by taking the i-th column of $\mathbf{E}$.

There are multiple ways to train the embedding matrix $\mathbf{E}$. The choice of method depends on the desired properties. In the case of language modeling, the primary requirement is the alignment of the embeddings with the rest of the model weights. Therefore, the embeddings are frequently initialized and trained jointly with the rest of the model parameters.

The trained embeddings capture the semantics of the tokens. This means that tokens occurring in similar contexts tend to have similar embeddings. This phenomenon is known as the *distributional hypothesis* (Harris 1954). The most common measure of similarity between two embeddings is their dot product or its normalized version, *cosine similarity*:

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \tag{2.2}$$

As the dot product suffers from the *curse of dimensionality*, the number of dimensions d' required to capture all semantic patterns is exponentially smaller than the vocabulary size K. This is achieved by the fact that two dissimilar vectors do not necessarily have to be orthogonal, as their dot product is already near zero.

2.3 Transformer Models

The most widely used architecture for language modeling is the Transformer model proposed by Vaswani et al. (2017). It is based on the Attention Mechanism, with its roots introduced by Bahdanau et al. (2014). Since then, a vast number of various architectural modifications have appeared. However, the core idea remains the same.

In the primary source paper, the authors proposed two parts of the architecture: the encoder and the decoder. Both of them, or their combination, are used depending on the specific task. In the case of language modeling (including code modeling), the decoder-only variant has gained widespread adoption. This part of the text describes only this type of Transformer.

2.3.1 Overview of the Forward Pass

Consider the sequence of context tokens $\boldsymbol{x}_{1:t-1} = (x_1, x_2, \ldots, x_{t-1})$ from Equation 2.1. To model the probability $p_\theta(x_t \mid \boldsymbol{x}_{1:t-1})$ of the next token x_t, the decoder-only Transformer first embeds the context tokens into vectors

$(\mathbf{Ex}_1, \mathbf{Ex}_2, \ldots, \mathbf{Ex}_{t-1})$ and then passes them through the stack of decoder layers to obtain the contextually enriched embeddings $(\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{t-1})$, which are called *hidden states*. It then takes the last hidden state and multiplies it by the weights matrix to get the *logits* $\mathbf{z}_t = \mathbf{W}\mathbf{h}_{t-1}$. The $\mathbf{W}$ is often called the *language modeling head*. The softmax function is then used to get the parameters of the categorical distribution over the vocabulary:

$$\mathbf{p}_t = \mathrm{softmax}(\mathbf{z}_t) = \frac{\exp(\mathbf{z}_t/\tau)}{\sum_{k=1}^{K} \exp(z_{t,k}/\tau)}, \tag{2.3}$$

where the exponent in the numerator and division are element-wise functions, and τ is a *temperature parameter* discussed in Section 2.5.1.

The resulting vector of estimated probabilities $\mathbf{p}_t$ can be used in multiple ways. First, to fulfill the original task of language modeling, the element corresponding to the next token x_t can be extracted from $\mathbf{p}_t$ and serve as $p_\theta(x_t \mid \boldsymbol{x}_{1:t-1})$ in Equation 2.1. Second, one can sample from the categorical distribution defined by $\mathbf{p}_t$ to get the next token and thus generate a new sequence from the model, which means that the language model can be used as a generative model. Both settings are commonly used. The first one is employed for the training process, while the second one is used for inference.

Before delving into the specifics of the decoder layer, two important points should be noted. First is that the initial token lacks context. Consequently, a special Beginning of Sequence token is utilized to provide context and serve as an embedding for $p_\theta(x_1)$. Second, for simplicity, the bias term is omitted in all the weight application formulas, as its inclusion is optional and contingent upon the engineer's design choices.

2.3.2   Decoder Layer

Denote the input embeddings as $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$, where $\mathbf{x}_i \in \mathbb{R}^{d_{\mathrm{model}}}$ for each $i \in \{1, 2, \ldots, T\}$, and let $\mathbf{X} \in \mathbb{R}^{T \times d_{\mathrm{model}}}$ represent their concatenated form. The decoder layer then applies the following operations:

$$\mathbf{X}^{(1)} = \mathbf{X} + \mathrm{MultiHead}(\mathrm{Norm}(\mathbf{X})), \tag{2.4}$$

$$\mathbf{X}^{(2)} = \mathbf{X}^{(1)} + \mathrm{MLP}(\mathrm{Norm}(\mathbf{X}^{(1)})), \tag{2.5}$$

where $\mathbf{X}^{(2)} \in \mathbb{R}^{T \times d_{\mathrm{model}}}$ is the output of the layer, and Norm, MultiHead, MLP are sub-layers discussed below.

These formulas differ from the original architecture described in Vaswani et al. (2017) as they employ *pre-normalization* (Baevski et al. 2018; Child et al. 2019; Wang et al. 2019). This modification enhances the stability of the training process by ensuring that the residual embedding stream is modified only through addition, which helps maintain well-behaved gradients at initialization (Xiong et al. 2020). To illustrate this concept in comparison to the original *post-normalization* structure, Figure 2.1 is provided.
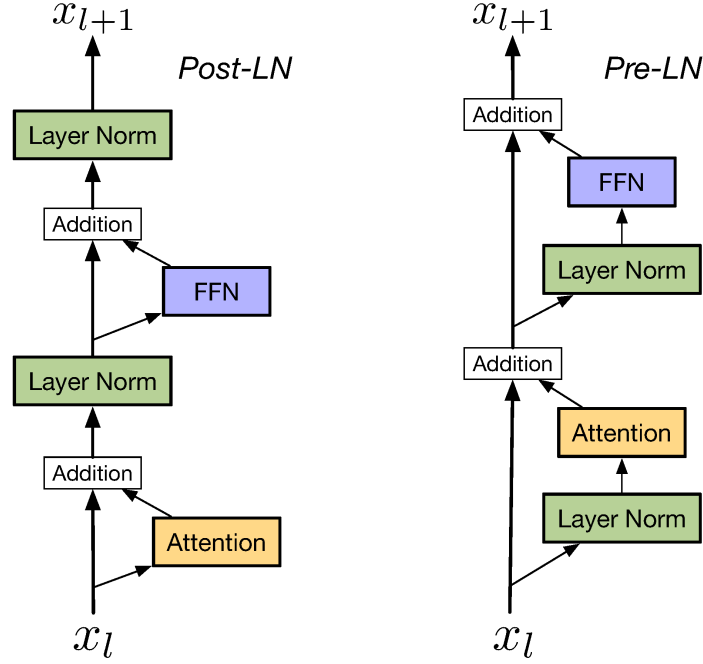
■ **Figure 2.1** Comparison of post-normalization and pre-normalization architectures in Transformer models

Source: Ding et al. (2021)

2.3.3 Normalization

The normalization layer stabilizes the training process by normalizing the input embeddings independently for each token. Various types of normalization layers exist. While the original architecture used Layer Normalization (Ba et al. 2016), Root Mean Square Layer Normalization (Zhang et al. 2019) has also been widely adopted.

2.3.4 Attention

Consider two additional architectural parameters d_k and $d_v = d_{\text{model}}$. The attention layer is described using three weight matrices: $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d_k \times d_{\text{model}}}$ and $\mathbf{W}_V \in \mathbb{R}^{d_v \times d_{\text{model}}}$. The output of the layer is calculated as follows:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\left[\mathbf{Q}\mathbf{K}^\top\right]_{\text{mask}}}{\sqrt{d_k}}\right)\mathbf{V}, \qquad (2.6)$$

where $\mathbf{V} = \mathbf{X}\mathbf{W}_V^\top$ and $\{\mathbf{Q}, \mathbf{K}\} = \text{RoPE}(\mathbf{X}\mathbf{W}_{\{Q,K\}}^\top)$ with RoPE denoting the Rotary Position Embedding discussed in Section 2.3.8. The softmax is applied

row-wise, and $[\,\cdot\,]_{\mathrm{mask}}$ is a causal masking operation that sets all elements above the main diagonal to $-\infty$, thereby preventing data leakage from future tokens.
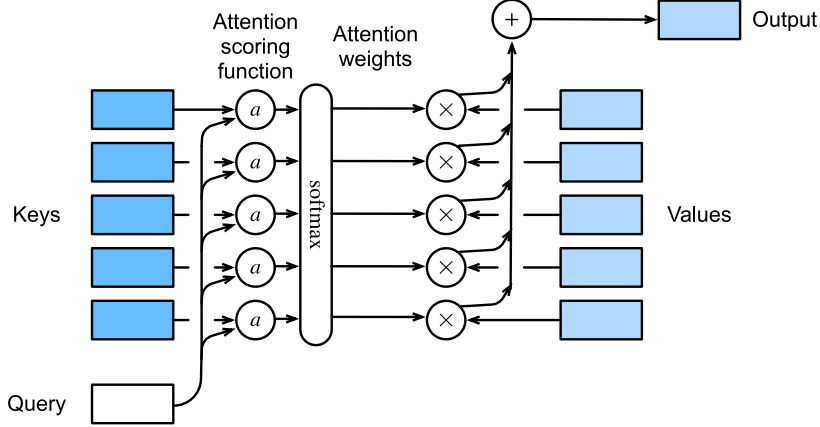
■ **Figure 2.2** Attention mechanism at the single token level

Source: Zhang et al. (2023a)

Attention functions as a layer that enables tokens to communicate with each other. Each token performs a query $\mathbf{q}_i$ to all preceding tokens and itself $\mathbf{k}_{j\leq i}$. The corresponding attention weight of the query-key match $\mathbf{q}_i^\top \mathbf{k}_j$ is normalized by the softmax and used as a soft dictionary lookup to retrieve the partial component of the value vector $\mathbf{v}_j$. The convex combination of all such value vectors $\mathbf{v}_{j\leq i}$ becomes the output of the attention layer. This operational flow for a single query is illustrated in Figure 2.2.

2.3.5 Multi-Head Attention

Multi-Head Attention is a modification of the basic attention mechanism. Instead of applying the attention mechanism once, the embedding processing is divided into multiple *heads* that perform attention in parallel. Given the number of heads h and an additional output projection weight matrix $\mathbf{W}_O \in \mathbb{R}^{hd_v \times d_{\mathrm{model}}}$, the output of the layer is calculated as:

$$\mathrm{MultiHead}(\mathbf{X}) = \tag{2.7}$$

$$\mathrm{MultiHead}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \mathrm{Concat}(\mathrm{head}_1,\ldots,\mathrm{head}_h)\mathbf{W}_O \tag{2.8}$$

$$\mathrm{where\ head_i} = \mathrm{Attention}(\mathbf{Q}_i,\mathbf{K}_i,\mathbf{V}_i), \tag{2.9}$$

where $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i$ are obtained by applying an independent set of weight matrices $\mathbf{W}_Q^{(i)}, \mathbf{W}_K^{(i)}, \mathbf{W}_V^{(i)}$ to the input embeddings $\mathbf{X}$, and $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are their concatenated versions.

In this version of the attention, d_v is not necessarily equal to d_{model}.

2.3.6 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a simple feed-forward neural network with two linear transformations and a non-linear activation function. It is applied to each token independently.

Originally, the Rectified Linear Unit (ReLU) was used as the activation function. However, other alternatives have empirically been shown to perform better. Two of these are the Gaussian Error Linear Unit (GELU) and the Sigmoid Linear Unit (SiLU) from Hendrycks et al. (2016). Another is the Swish Gated Linear Unit (SwiGLU) introduced in Shazeer (2020).

2.3.7 Dropout

Dropout is a widely adopted regularization technique that enhances the generalization of the model (Srivastava et al. 2014). During the training process, it randomly masks some of the activations of the preceding layer with a probability p_{dropout} and scales the remaining activations by a factor of $1/p_{\mathrm{dropout}}$ to maintain the output statistics of the layer. This process is conceptually illustrated in Figure 2.3.
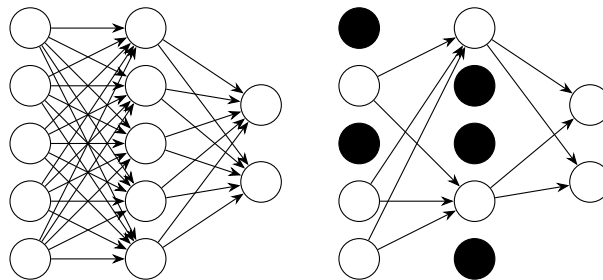
■ **Figure 2.3** Dropout applied to the activations of a multi-layer perceptron, with a dropout probability of $p_{\mathrm{dropout}} = 0.5$

Source: Labach et al. (2019)

This method is frequently applied in Transformers across all layers, including the dropout of the attention scores calculated before the application of the softmax function. (Vaswani et al. 2017; Zehui et al. 2019)

2.3.8 Rotary Position Embedding

Rotary Position Embedding is a modification built upon the vanilla Transformer to incorporate the positional information of tokens and address issues present in the originally proposed analogous approach. RoPE was introduced by Su et al. (2021) and quickly gained widespread adoption.

For each embedding $\mathbf{x}_m$ at position m in the sequence, RoPE proposes a rotation of its query and key vectors:

$$\{\mathbf{q}_m, \mathbf{k}_m\} = \mathrm{RoPE}(\mathbf{W}_{\{Q,K\}}\mathbf{x}_m, m) = \mathbf{R}^d_{\Theta,m}\mathbf{W}_{\{Q,K\}}\mathbf{x}_m \qquad (2.10)$$

using an orthogonal rotary matrix

$$\mathbf{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}, \quad (2.11)$$

where $d = d_k$ and $\Theta = \{\theta_i = \theta_{\mathrm{base}}^{-2(i-1)/d}, i \in \{1, 2, \ldots, d/2\}\}$ is a predefined set of frequencies derived from the *base frequency* hyperparameter θ_{base}. RoPE can be interpreted as a rotation of two-dimensional scalar chunks of the query and key vectors, determined by both the token's position within the sequence and the chunk's position within the vector, as illustrated in Figure 2.4.
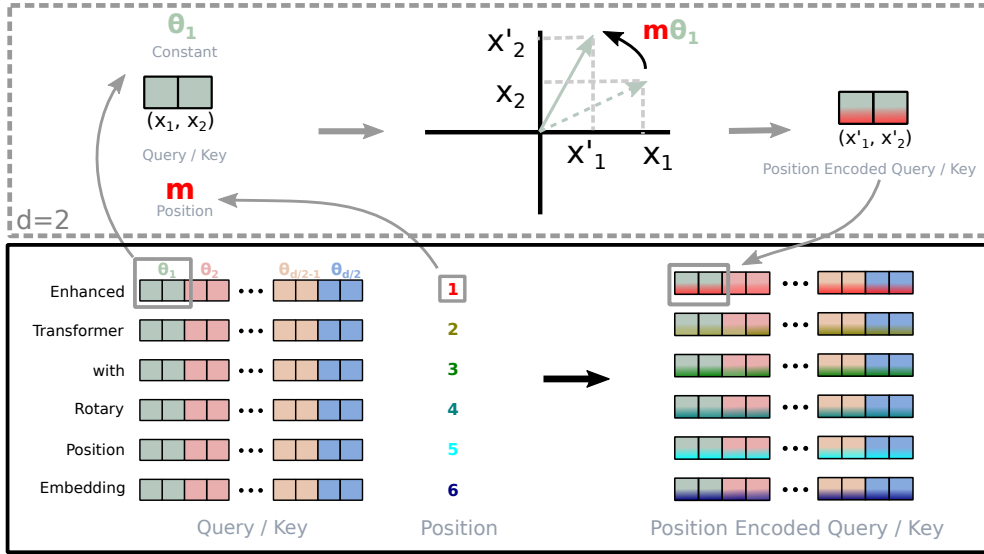
■ **Figure 2.4** Illustration of the effect of RoPE on the query and key vectors
Source: Su et al. (2021)

Note two important details regarding the application of RoPE. First, it requires d to be even. Second, it is applied across all heads in multi-head attention without repeating the rotation frequency h times, inherently adding more functional bias to heads working on higher frequencies (Barbero et al. 2024).

When multiplying query and key matrices in Equation 2.6, the two embeddings $\mathbf{x}_m$ and $\mathbf{x}_n$ derive the attention score decay based on the relative distance $m - n$ between them:

$$\mathbf{q}_m^\top \mathbf{k}_n = (\mathbf{R}_{\Theta,m}^d \mathbf{W}_Q \mathbf{x}_m)^\top (\mathbf{R}_{\Theta,n}^d \mathbf{W}_K \mathbf{x}_n) = \mathbf{x}_m^\top \mathbf{W}_Q \mathbf{R}_{\Theta,n-m}^d \mathbf{W}_K \mathbf{x}_n, \qquad (2.12)$$

which allows the model to capture the positional information of the tokens.

2.4   Training

To elaborate on the process of convergence from a randomly initialized model to one with desirable properties, this section provides an overview of the training process by deriving the direct objective and describing the optimization algorithm, along with associated batch-related terminology and learning rate scheduling.

2.4.1   Maximum Likelihood Estimation

Training the language model p_θ involves aligning the distribution it represents with the true data distribution. However, the later is unknown, and only a finite set $\mathcal{D} = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}\}$ of instances sampled from it is available. The training objective $\mathcal{L}_\mathcal{D}$, known as the *likelihood function*, is expressed as the probability of these samples being generated by the model. Assuming that samples are independent and identically distributed, $\mathcal{L}_\mathcal{D}$ can be decomposed into the product of individual probabilities:

$$\mathcal{L}_\mathcal{D}(\theta) = p_\theta(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}) = \prod_{i=1}^n p_\theta(\boldsymbol{x}^{(i)}) \to \max \qquad (2.13)$$

The method of estimating the parameters θ by maximizing this function is called Maximum Likelihood Estimation.

Due to several reasons, including underflow issues, the *log-likelihood function* is often used instead:

$$\ell_\mathcal{D}(\theta) = \log \mathcal{L}_\mathcal{D}(\theta) = \log \prod_{i=1}^n p_\theta(\boldsymbol{x}^{(i)}) = \sum_{i=1}^n \log p_\theta(\boldsymbol{x}^{(i)}) \qquad (2.14)$$

Optimizing both functions is equivalent, as the logarithm is a monotonically increasing function.

To take one step further, a minus sign can be added to focus on minimization instead of maximization, thereby formulating the *negative log-likelihood loss function*. Additionally, Equation 2.1 can be used to express the loss function as a sum of the log-likelihoods of the individual tokens:

$$L(\theta) = -\ell_\mathcal{D}(\theta) = -\sum_{i=1}^n \sum_{t=1}^{T_i} \log p_\theta(x_t^{(i)} \mid \boldsymbol{x}_{1:t-1}^{(i)}) \to \min \qquad (2.15)$$

This objective corresponds to the Cross-Entropy (CE) loss, which is often represented using the dot product of $\mathbf{p}_t^{(i)}$ from Equation 2.3 and the one-hot encoding of the target token $x_t^{(i)}$.

2.4.2 Gradient-Based Optimization

Various optimization algorithms have been developed to train neural networks. The most effective ones are based on first-order methods, which iteratively adjust the parameters by subtracting the stochastic gradient of the loss function with respect to the parameters. The degree of this adjustment is controlled by multiplicative factor of the gradients called *learning rate*. Gradients are calculated using a subsample of the training set, known as a *mini-batch* (or simply *batch*), with the application of automatic differentiation.

In the field of language modeling, the Adaptive Moment Estimation (Adam) (Kingma et al. 2014), a modification of Stochastic Gradient Descent, and its version supporting weight decay (Loshchilov et al. 2017) are the most popular algorithms. Adam maintains two moving average statistics, called the first and second *moments*, for each parameter, which correct the application of the gradient and accelerate convergence.

Training is sometimes conducted using a half-precision floating-point format to increase computational speed. However, this approach introduces the risk of underflow for small gradient values. To mitigate this issue, *gradient scaling* is employed. This technique multiplies the loss value by a constant factor, allowing the gradients to maintain appropriate magnitudes during backpropagation. When the optimization step is performed, the gradients are rescaled by the same factor to restore the original learning rate setting. The scaling factor is often determined automatically. (NVIDIA Corporation 2023)

Another common modification applied to gradients is *clipping*, which projects the gradient vector onto the surface of a hypersphere with a radius equal to a predefined value if its norm exceeds this threshold. This process prevents gradients from exploding in regions with sharp loss landscapes and stabilizes the training process. (Pascanu et al. 2013)

2.4.3 Batch-Related Terminology

Due to the operational nature of Graphics Processing Units (GPUs), multiple sequences must be organized to have the same length to be processed in parallel. This is achieved by either padding shorter sequences with special tokens or truncating longer ones. The model's scope of visibility is termed the *context window*, and its size is referred to as the *context length*.

In some cases, sequences are randomly shuffled, concatenated, and split to form batches without the need for padding or truncation. This technique is known as *batch packing*.

When hardware constraints limit batch size, *gradient accumulation* is utilized. Instead of using the entire batch for a single optimization step, it is divided into *micro-batches*, which are processed sequentially. The parameters are updated only after all the micro-batches have been processed.

2.4.4 Learning Rate Scheduling

Due to the design of Adam, a few iterations at the beginning of training are required to gather sufficient gradient statistics for better moment estimation. This requirement becomes even more critical in the context of the instability associated with Transformers. During this initial phase, using the full learning rate is inadvisable, as it may lead to unstable updates. To mitigate this issue while utilizing all available data, a *warm-up* period is introduced in the training process. Throughout this period, the learning rate is gradually increased, often linearly, from zero to the desired value over a predefined number of iterations.

After training with high learning rates, the model benefits from smaller parameter updates as it approaches the local optimum and requires more precise adjustments. This is achieved by decaying the learning rate over time.

To summarize, the learning rate, rather than being a constant hyperparameter, has evolved into a function of the iteration index, thereby opening a new dimension for controlling the training process.

2.5 Inference

This section introduces the fundamental concepts related to the inference process of language models, specifically sampling and stopping criteria.

2.5.1 Sampling

As previously mentioned, a language model represents a probability distribution over the vocabulary given the preceding context. The iterative process of sampling from this distribution and incorporating these new tokens into the model's context is referred to as *generation*.

Tokens are selected based on their associated probabilities $\mathbf{p}_t$ from the softmax output, as described in Equation 2.3. The *temperature* parameter modulates the sharpness of this distribution. There are two limiting cases, which are demonstrated in Figure 2.5: First, as $\tau \to +\infty$, the distribution becomes uniform, with all tokens having an equal probability of being selected. Second, as $\tau \to 0^+$, the selection becomes deterministic, choosing the most probable token. This is known as *greedy decoding* and corresponds to the application of the greedy search algorithm in a graph of all possible sequences, where the language model functions as a heuristic.

Logits: [0.50, -0.14, 0.65, 1.52, -0.23]
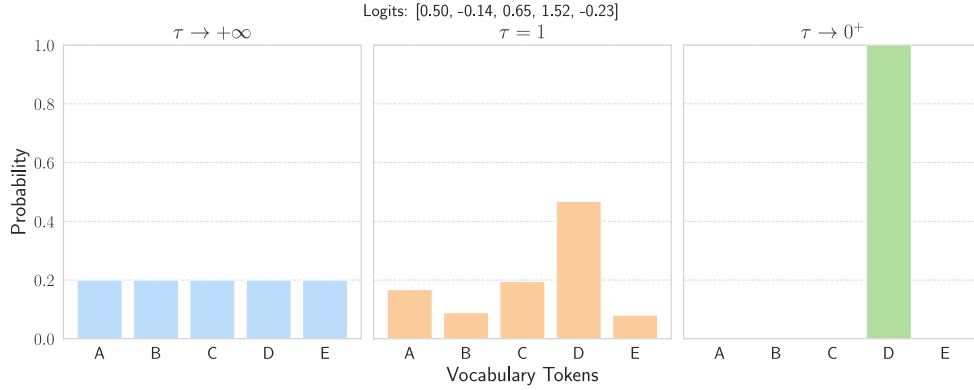
■ **Figure 2.5** Impact of the temperature parameter on the softmax output. The middle image demonstrates the standard effect of the softmax function on an array of logits. The images on the left and right show the limiting case of temperature adjustment.

To further enhance the quality of the generated text, the *top-k sampling* strategy is often employed. It restricts the vocabulary to the top-k most probable tokens, which helps prevent the model from outputting irrelevant tokens.

A similar concept is applied in the *nucleus sampling* strategy, also known as *top-p sampling* (Holtzman et al. 2019). It restricts the vocabulary to the smallest set of tokens whose cumulative probability exceeds a threshold parameter p.

2.5.2   Stopping Criteria

There are several methods to terminate the generation process.

Firstly, the model can signal the completion of the output sequence by generating an End of Sequence (EOS) token. This requires the EOS token to be included in the vocabulary, and the model must be trained to produce it. This is accomplished by appending the EOS token to the end training sequences.

Secondly, algorithmic stopping criteria can be employed. For example, the generation process can be halted when a maximum length threshold is exceeded. Another pertinent example, particularly in the context of code completion, is to cease generation when the model outputs a token containing a newline character.

Chapter 3

Completion-Centric LM

This chapter builds upon the topic of standard language modeling by delving into the specifics of the code completion task. It provides a description of the training stages of Code LLMs, gradient masking, FIM, and the evaluation of code completion models.

3.1 Training Stages

The training of modern Code LLMs begins with acquiring a model checkpoint that possesses a robust understanding of code. This can be achieved through two distinct types of training stages. One type involves taking a general-purpose pre-trained LLM and fine-tuning it on a file-level code dataset. This approach is utilized by models such as Code Llama (Rozière et al. 2023) and Qwen2.5-Coder (Hui et al. 2024). The other type involves training a model from scratch using a mixture of code and code-related natural language data. This pre-training approach is adopted by models like DeepSeek-Coder (Guo et al. 2024) and OpenCoder (Huang et al. 2024). Both types necessitate vast amounts of data, typically ranging from 2 trillion (2T) to 18 trillion (18T) tokens with context length of 4,096 (4K).

Once a strong model with a limited context size is trained, it often undergoes *repository-level pre-training* (sometimes referred to as *long-context fine-tuning*). The objective of this stage is to extend the context window, thereby enabling the model to comprehend a broader scope within a given repository. Unlike the initial training stage, repository-level pre-training requires significantly fewer tokens. However, it necessitates the application of context extrapolation methods and the utilization of longer input sequences. This phase typically involves between 8 billion (8B) and 300 billion (300B) training tokens, with the context window extended with 16K or 32K tokens.

Each input sequence in repository-level pre-training consists of two components: the *composed context* and the *completion file*. Various methods exist

for obtaining the former. In this thesis, the function responsible for this task is referred to as the *context composer*, or simply *composer*. This process involves retrieving and preprocessing a subset of files, which are then assembled to form a repository context.

Models trained using only the aforementioned stages are referred to as *base models*. To enhance their utility for various tasks, an additional instruction tuning phase is often conducted. However, the capabilities gained from this stage are not essential for the code completion task and are not discussed further in this thesis. Throughout this work, all mentioned models are assumed to be their base versions.

3.2 Gradient Masking

The learning signal for the model is derived from both the composed context and the completion file sources. The alignment of their distributions is contingent upon the specific choice of the composer. When learning the completion of all tokens, a mismatch in these distributions can introduce undesirable bias into the model.

For instance, learning to complete README files may be irrelevant if the model's primary objective is to excel solely in code completion. However, incorporating various file formats into the context is justified if they hold relevance to the completion task (e.g., including documentation).

This issue can be mitigated through the application of *gradient masking*. By setting the individual losses of non-completion tokens to zero, these tokens can be excluded from the gradient computation during training. The impact of this method is examined in Section 6.5.2.

3.3 Fill-in-the-Middle

Due to the autoregressive nature of decoder-only Transformers, they are unable to utilize future tokens in their context. Consequently, to account for both the prefix and suffix of the completion file, the Fill-in-the-Middle approach was proposed by Bavarian et al. (2022), with its origins in the works of Donahue et al. (2020), Aghajanyan et al. (2022), and Fried et al. (2022). This capability is particularly advantageous for the completion task, as code is frequently written in a non-sequential and chaotic order.

The primary concept of FIM involves randomly dividing a portion of training sequences into three segments: prefix, middle, and suffix. These segments are then concatenated using special tokens to form a new sequence order: prefix, suffix, and middle. Incorporating such augmented data into the pretraining process, introduces a new infilling capability to the model, albeit with a marginal performance degradation (Allal et al. 2023; Rozière et al. 2023).

3.4   Evaluation

The assessment of code completion models requires a multifaceted approach that captures both syntactic and functional aspects of generated code. This section surveys the principal methodologies for evaluating model outputs, highlighting the diversity of metrics and their relevance to practical development scenarios.

3.4.1   Metrics

A variety of metrics have been developed to quantify the performance of code completion systems. These measures reflect different perspectives on correctness, similarity, and usability, ranging from exact syntactic matches to more nuanced assessments of edit effort and semantic equivalence. The following overview presents the most prominent of them.

3.4.1.1   Exact Match

The Exact Match (EM) metric is a fundamental and widely used measure for evaluating code completion. It is defined as the ratio of correctly completed code lines to the total number of lines. This metric is particularly valued for its direct alignment with the objectives of code completion evaluation.

However, the Exact Match metric operates at a line-level granularity, which means it only indicates whether a line is completed correctly or not. This binary nature of the metric does not offer insights into the degree of deviation from the correct completion in cases where the completion is incorrect. To mitigate this limitation, the Exact Match metric is often used in conjunction with the more granular measures.

3.4.1.2   Edit Similarity

The generation of functionally correct code completions is valuable, yet the effort required to edit and adapt these generated lines is equally significant. Edit Similarity (ES) quantifies this effort by measuring the number of single-character edits (insertions, deletions, or substitutions) needed to transform the generated code into a reference (Svyatkovskiy et al. 2020). Mathematically, ES for two strings is expressed as:

$$\mathrm{ES}(\boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{\mathrm{lev}(\boldsymbol{x}, \boldsymbol{y})}{\max\{|\boldsymbol{x}|, |\boldsymbol{y}|\}}, \tag{3.1}$$

where $\mathrm{lev}(\boldsymbol{x}, \boldsymbol{y})$ represents the Levenshtein distance between the generated code $\boldsymbol{x}$ and the reference sample $\boldsymbol{y}$, and $|\cdot|$ denotes the length of the string.

This metric is crucial for evaluating code completion scenarios, as it provides a measure of the effort developers must exert to correct errors in the

generated code. Moreover, it has been shown that ES moderately correlates with the generation of functionally correct code (Dibia et al. 2022).

3.4.1.3 Cross-Entropy and Perplexity

As mentioned earlier, cross-entropy is employed as a loss function in the training process of language models. More specifically, it represents the average log-likelihood of the individual ground-truth tokens. It can also be interpreted as the average number of nats required to encode the model's predictions per token compared to the true distribution.

Perplexity (PPL) is the exponentiated form of cross-entropy, providing a more intuitive interpretation as the average number of choices among which the model is uncertain. For this reason, perplexity is often referred to as the weighted average branching factor of a language (Murphy 2022).

Both metrics are frequently used as proxies for assessing model quality. They are valuable for monitoring because these measures are consistently computed during training, being either derived from the loss function (PPL) or representing the loss itself (CE). However, they are too abstract to serve as primary metrics for specific tasks. Additionally, these measures are heavily influenced by vocabulary size and tokenization methods, which makes them unsuitable for comparing different models.

3.4.1.4 Top-k Accuracy

Top-k Accuracy is a metric that quantifies the frequency with which the model's top-k predictions align with the actual ground-truth completion. Within the scope of this thesis, Top-k Accuracy is considered as a token-level metric to provide a more granular evaluation of the model's performance.

3.4.1.5 Pass@k

All aforementioned metrics are syntax-based, meaning they evaluate the model's performance based on the syntactic match between the generated and reference completions. However, in real-world scenarios, there exists a wide variety of correct completions that may not be present in the dataset used for evaluation. The model might predict a completion that fulfills the user's functional requirements, even if the metric evaluates it as incorrect.

To address this issue, an unbiased estimation of the probability that at least one generated solution out of k passes all unit tests was proposed by Chen et al. (2021). This metric, known as pass@k, is calculated using the following expectation:

$$\text{pass@k} = \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \tag{3.2}$$

where $n \geq k$ is the number of samples generated by the model, and $c \leq n$ is the number of those samples that passed all unit tests. This metric reflects the probability of finding a correct solution within k attempts, which mirrors the iterative approach developers often take when exploring multiple solutions.

To apply this metric to single-line code completion, it is assumed that all other lines, except the target one, are present and functionally correct. This assumption does not hold for many use cases, such as when a user writes a function on the fly. Therefore, pass@k offers an optimistic assessment of completion capabilities, as the evaluation set contains one corrupted line per problem. In addition to this, Liu et al. (2024) introduced a comprehensive list of further limitations in their Appendix A.4 section. In summary, pass@k is more realistic for tasks with greater completion granularity or for code generation assessment.

3.4.1.6   Further Metrics

Several metrics for evaluating generated code have been migrated from machine translation and summarization fields, yet they often face limitations due to their unnatural application to code (Evtikhiev et al. 2022). This has prompted the research community to develop code-specific adaptations. However, identifying an optimal metric for code-oriented tasks remains an open challenge.

N**-gram Based Metrics:**   The Bilingual Evaluation Understudy (BLEU) (Papineni et al. 2002) is a precision-oriented metric that measures the proportion of n-grams in the candidate text that also appear in the reference snippet. BLEU4 is a specific instance of BLEU that considers n-grams up to a length of 4. This configuration is widely adopted because it balances the evaluation of both individual words and longer phrases. The Recall-Oriented Understudy for Gisting Evaluation (ROUGE) (Lin 2004) is a family of recall-based metrics originally developed for evaluating automatic summarization. ROUGE-N calculates n-gram recall between a candidate and a reference, while ROUGE-L uses the longest common subsequence to capture word order without requiring consecutive matches.

Character-Level Metric: The ChrF metric (Popović 2015) is an F-measure that evaluates character n-grams instead of word ones. It incorporates a β parameter to adjust the emphasis between precision and recall. ChrF represents an advancement over previously mentioned metrics by balancing precision and recall in a single measure. The character-level approach offers several advantages: it eliminates tokenization dependencies, implicitly captures morpho-syntactic information, and requires no additional linguistic tools.

Semantic-Enhanced Metrics: Metric for Evaluation of Translation with Explicit Ordering (METEOR) (Banerjee et al. 2005) improves upon simple n-gram matching by adding support for stemming, synonymy, and paraphrasing, along with a penalty for disordered n-grams. BERTScore (Zhang et al. 2019) leverages contextual embeddings from BERT (Bidirectional Encoder Representations from Transformers) to compute similarity between tokens in candidate and reference texts, enabling robust matching of paraphrases and semantic equivalents.

Code-Specific Metrics: RUBY (Tran et al. 2019) was specifically designed for code evaluation, integrating lexical, syntactic, and semantic representations of source code to better capture functional equivalence. It uses an ensemble approach that leverages the most reliable available representation: program dependence graphs, abstract syntax trees, and surface text. CodeBLEU (Ren et al. 2020) extends BLEU by adding weighted n-gram matching for programming language keywords, an abstract syntax tree match component to capture structural information, and a data-flow match to evaluate semantic correctness.

3.4.2 Evaluation Modes

Besides the chosen inference strategy, types of which were discussed in Section 2.5, there are multiple setups for evaluating the repository-level completion capabilities of the Code LLM.

During training, language models typically employ a technique known as *teacher forcing*. In this method, each subsequent token prediction is conditioned on the ground-truth tokens from the preceding sequence. This approach takes advantage of the parallelizable nature of Transformers, facilitating efficient training. Conversely, during evaluation, the model can be conditioned on its own previously generated tokens, offering a more realistic assessment of its capabilities. Despite this, teacher forcing remains prevalent during training for the purpose of collecting metrics and monitoring progress, as it does not require additional forward passes. It is crucial to recognize that teacher forcing can render some metrics meaningless, as they may assume conditioning on the model's output distribution. For example, ES necessitates the generation of a full line solely by the model without external guidance, as it compares entire lines rather than individual tokens. The same statement does not hold for EM, where a single incorrectly generated token results in the entire line being deemed incorrect.

For single-line code completion, the repository context can be composed based on the file prefix (and suffix in the case of FIM) or the entire file. The former method requires a separate composition and forward pass for each target line in the composition file, offering superior performance. In contrast, the latter requires only a single instance of both, but it risks data leakage of

the target line for the retrieval mechanism. Despite the potential issues of the second method, it is suitable for training as it does not necessitate gradient masking for the file prefix and provides more training tokens for loss computation.

Apart from training evaluation, a validation loop is employed periodically to monitor the decline in a model's ability to generalize to new data, a phenomenon known as *overfitting*, which occurs due to excessive memorization of the training data. To ensure that validation metrics are comparable to training ones, the same evaluation setup is often utilized. However, this approach differs from the final evaluation of the model on various benchmarks, which is conducted using scenarios that closely resemble real-world conditions.

3.4.3   Benchmarks

In recent years, the field of code completion has witnessed significant advancements in benchmarking, particularly for repository-level tasks. This section presents a chronological overview of major benchmarks that have shaped this domain.

CodeXGLUE, introduced by Lu et al. (2021), represents one of the first comprehensive benchmarks for code intelligence. It encompasses a collection of 10 tasks across 14 datasets, including code completion, which it evaluates using datasets like PY150 (Raychev et al. 2016) and GitHub Java Corpus (Allamanis et al. 2013). While CodeXGLUE established a foundation for code completion evaluation, it primarily focused on in-file completion without considering cross-file dependencies that are prevalent in real-world software development.

Addressing this limitation, Ding et al. (2022) proposed CoCoMIC, which pioneered the joint modeling of in-file and cross-file context for code completion. CoCoMIC's key innovation was the introduction of a Cross-File Context Finder (CCFINDER) that locates and retrieves relevant cross-file context. This approach significantly improved completion accuracy, particularly for Application Programming Interface (API) usage scenarios, highlighting the importance of repository-level understanding in code completion tasks.

Building upon these cross-file context concepts, Zhang et al. (2023b) introduced RepoEval, a benchmark constructed from high-quality real-world repositories. Unlike its predecessors, RepoEval explicitly focused on three levels of code completion granularity: line, API invocation, and function body completion scenarios. The benchmark also employs unit tests present in the repositories for evaluating function body completions. RepoEval's design brought the evaluation closer to real-world development practices.

RepoBench, proposed by Liu et al. (2023), address RepoEval's limitation of limited repository coverage by significantly expanding the evaluation dataset. It further enhanced the evaluation framework by decomposing the repository-level code completion process into three interconnected tasks: RepoBench-R (retrieval), RepoBench-C (code completion with both 2K and 8K context

lengths), and RepoBench-P (end-to-end pipeline). This modular approach enabled more targeted evaluations of each component while maintaining their interdependence in a complete system. RepoBench also expanded language coverage to include both Python and Java, representing a step toward multilingual evaluation. In addition, RepoBench demonstrated that incorporating cross-file contexts significantly improves code completion performance, even with randomly selected contexts.

CrossCodeEval (Ding et al. 2023) significantly broadened the multilingual scope by incorporating four popular programming languages: Python, Java, TypeScript, and C#. It introduced a rigorous methodology for creating examples that strictly require cross-file context for accurate completion, utilizing static analysis to pinpoint cross-file context usage within the current file. Their evaluations demonstrated that models performed poorly with only in-file context but improved significantly when cross-file context was included, suggesting BM25 (Robertson et al. 2009) as an effective retrieval method. This benchmark established a higher standard for evaluating cross-file contextual understanding in diverse programming languages. Later, CrossCodeLongEval (Wu et al. 2024a) expanded upon this foundation by addressing CrossCodeEval's limited task coverage, introducing chunk and function completion scenarios alongside line completion to provide a more comprehensive evaluation framework.

Advancing toward more realistic scenarios, Deng et al. (2024) proposed R^2C^2-Bench, which introduced a context perturbation strategy to simulate real-world repository-level code completion environments. This benchmark constructs candidate retrieval pools with both abstract and snippet contexts, capturing both coarse-grained global information and fine-grained local details. R^2C^2-Bench's comprehensive approach highlighted limitations in previous benchmarks, particularly regarding the coverage of diverse completion scenarios.

Long Code Arena (LCA), as introduced by Bogomolov et al. (2024), presented a comprehensive suite of six benchmarks, with a particular focus on evaluating repository-level capabilities across multiple code-related tasks, including project-level code completion. This component of LCA offers a more nuanced evaluation of completion tasks by categorizing target lines into six distinct types. Notably, the two most significant categories are *infile* and *in-project*, which respectively denote lines that utilize an API declared within the completion file and in other files within the repository. The dataset is stratified into four segments based on the total number of characters forming the overall context, thereby representing varying levels of complexity. To prevent data leakage, a traversal of Git history was conducted. In summary, Long Code Arena added an additional layer of diversity to repository-level code completion benchmarks, enhancing the evaluation landscape.

Wu et al. (2024b) proposed RepoMasterEval, which prioritized testing in authentic development conditions. Recognizing limitations in test suite quality

of previous benchmarks, RepoMasterEval employed mutation testing and manual test case crafting to ensure robust evaluation. Uniquely, this benchmark conducted an industrial study correlating model performance with online acceptance rates, demonstrating that RepoMasterEval scores can indeed predict real-world usability of code completion systems.

Further emphasizing developer experience, Codev-Bench (Pan et al. 2024) introduced a developer-centric evaluation framework based on business analysis from industrial code completion products. It redefined evaluation criteria to better align with developers' intent and desired completion behavior throughout the coding process. Codev-Bench leveraged an agent-based system to automate repository crawling, environment construction, and dynamic call chain extraction from existing unit tests, providing a more realistic assessment of code completion in modern software development.

Most recently, M^2RC-EVAL (Liu et al. 2024) significantly expanded the multilingual coverage to 18 programming languages, addressing the limited language scope of previous benchmarks. It introduced two types of fine-grained annotations (bucket-level and semantic-level) that enable comprehensive analysis of model performance across different code semantics and complexity levels. M^2RC-EVAL also created a massively multilingual instruction corpus to enhance repository-level code completion capabilities of existing models, establishing a new standard for comprehensive multilingual evaluation.

The evolution of these benchmarks reflects the field's progressive understanding of the complexities inherent in repository-level code completion, moving from isolated single-file evaluations to comprehensive assessments that consider cross-file dependencies, multilingual capabilities, and real-world usage patterns. These benchmarks collectively provide a robust framework for evaluating and improving code completion systems in increasingly realistic scenarios.

In-Context Learning

This chapter provides an overview of the in-context learning paradigm in relation to the thesis objectives. It begins with a definition, continues with a discussion of long context, and concludes with the relevance of in-context learning to the code completion task.

4.1 Definition

In-Context Learning (ICL) refers to the ability of a model to condition on a set of demonstrations provided in the input context in order to learn and adapt to unseen tasks without modifying its parameters (Brown et al. 2020). The input string is called a *prompt*, and typically includes a task description along with a set of examples. The process of constructing a prompt to better leverage ICL is referred to as *prompt engineering*.

This learning paradigm has proven highly effective, as it enables specialized approaches to solving tasks based solely on inference computations, while the model is pre-trained without awareness of task specificity.

4.2 Long Context

ICL introduces context length as a new scaling direction for LLMs (Kaplan et al. 2020), as shown in Figure 4.1. The terminology also encompasses the concepts of *zero-shot*, *one-shot*, and *few-shot prompting*, which refer to the number of demonstrations provided in the prompt. Increasing the number of demonstrations typically leads to improved downstream performance (Brown et al. 2020).

However, leveraging the advancements of this scaling direction requires addressing multiple challenges. The following overview develops a fundamental understanding of some of the directions associated with long context in the field.
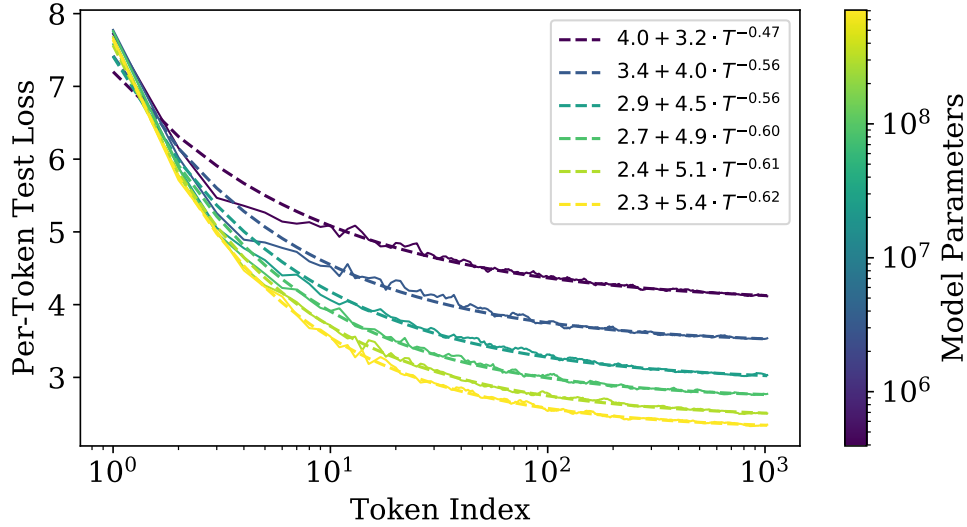
Figure 4.1 Power-law scaling of test loss with context token position
Source: Kaplan et al. (2020)

4.2.1 Quadratic Complexity of Attention

The original attention architecture of Transformer models exhibits quadratic computational and time complexity requirements, which depend on the number of tokens provided to the model (Vaswani et al. 2017). This undesirable property has prompted the development of numerous subquadratic and other efficient attention mechanisms. For instance, the survey by Tay et al. (2022) offers a detailed examination of this subject and visualizes the taxonomy of such architectures, as shown in Figure 4.2.

4.2.2 Retrieval-Augmented Generation

Not only the number of demonstrations but also their relevance to the task has been proven to be a crucial factor in ICL performance (Liu et al. 2021). Given a limited upper bound on the context length of the prompt, the task of finding the most relevant grounding information has become an important component of the respective research.

Closely related to this idea is the Retrieval-Augmented Generation (RAG) framework, which introduces a separate retrieval step in the prompt engineering procedure (Lewis et al. 2021). The objective of this mechanism is to provide the model with access to a non-parametric chunk of information related to the task at hand. The provided context can be either a factual knowledge supplement or the demonstrations required by the ICL setting.
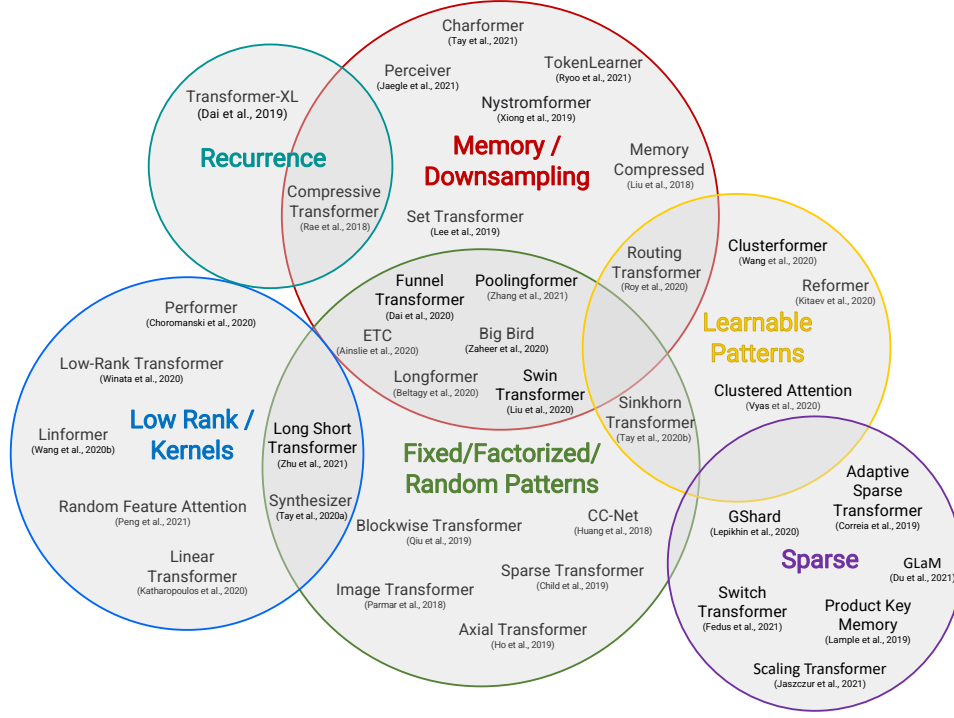
■ **Figure 4.2** Taxonomy of efficient Transformer architectures

Source: Tay et al. (2022)

RAG addresses the problem of overfitting and updates the outdated parameter-based memory of the model with a more relevant, query-conditioned context.

4.2.3   Context Extension

Since long context is an obstacle present at the architectural level, it affects both the inference and training modes of the model. The latter is particularly problematic, as it is more resource intensive. This issue is mitigated by dividing the model development pipeline into multiple stages, the specific cases for Code LLMs of which are discussed in Section 3.1. First, models are heavily trained on a small context window, which is then extended to larger ones. This approach drastically reduces training time and long context data requirements, thereby increasing the efficiency of the pipeline (Xiong et al. 2023).

Context extension is functionally linked to RoPE, as the only novel information passed to the model with longer contexts is the greater positional span. This necessitates the introduction of an extrapolation property for this information. However, this cannot be directly achieved by simply training on longer sequences, as RoPE has been shown to be a universal approximator and suffers greatly on ranges outside of the original context window

(Chen et al. 2023). To address this issue, multiple methods have been proposed (Chen et al. 2023; Rozière et al. 2023; Peng et al. 2023; Xiong et al. 2023), the main idea of which is to shift from the landscape of the extrapolation problem to that of interpolation.

The most fundamental modification in the model's architecture between short and long training stages is an Adjustment of the Base Frequency (ABF) parameter of RoPE, as concurrently proposed by Rozière et al. (2023) and Xiong et al. (2023). More specifically, θ_{base} is scaled by one or two orders of magnitude to diminish the decay of the attention scores related to the increased relative distance $m - n$ between tokens.

4.3 Code Completion Prompting

ICL emerges during the training of large language models via the next-token prediction objective as a function of the number of parameters and the data size used for pre-training (Hahn et al. 2023). Thus, its presence is found in base models, which are used to perform the code completion task.

In this thesis, individual demonstrations are not distinguished, and the prompt is treated as a set of concatenated text snippets, which consist of the content of repository text files and are organized according to RAG principles, where the completion file serves as a query.

Part II

Applied Research

The investigation of the final research question, RQ.B2, in this thesis was previously published by Sapronov et al. (2025) in the "Tiny Papers" track at the Deep Learning for Code (DL4C) workshop during the International Conference on Learning Representations (ICLR) 2025. This thesis should be regarded as an extended version of that paper, authored by the same individual, and does not constitute plagiarism of either their own or others' work. All differences are highlighted and justified below.

Chapter 5

Technical Foundation

In this chapter, we provide a comprehensive description of the context composition framework, which is applied to preprocess the data as reported in the subsequent section. We then present the fine-tuning pipeline, a tool that integrates this data to facilitate the project adaptation of Code LLMs. Additionally, a list of tools necessary for executing these implementations is included.

The contributions outlined in this chapter are: the context composition framework with 13 different composers implemented, a set of pre-composed training datasets, and a fine-tuning pipeline.

5.1 Context Composition Framework

We begin this section with an explanation of the independent building blocks of the context composition framework and their integration to form a complete compositional strategy. The chapter then enumerates all the composers utilized in the experimental section, accompanied by their descriptions. Finally, we offer an overview of the additional parameter applied to modify the retrieval order and include a commentary that invites further exploratory investigation into the framework.

5.1.1 Building Blocks

As stated in Section 3.1, we define the context composer as a function that takes a repository snapshot and a completion file and returns a string that represents the project context. The investigation of ICL capabilities requires a diverse array of such functions. Therefore, we conclude that the most effective strategy to achieve this diversity is to decompose the context composers into atomic pieces that can be repeatedly utilized. We propose seven abstract building blocks for this framework:

1. **File Filtering** determines which files to include in the subsequent data processing pipeline. For instance, it can filter out files with empty content or files with certain extensions.

2. **File Preprocessing** modifies the content of the files. For example, it can extract code from markdown files.

3. **File Chunking** changes the granularity of the data from files to text chunks. The simplest form of chunking is identity, which maintains the file-grained structure.

4. **Chunk Ranking** determines the relevance of the chunks for the completion file. It can take multiple forms, ranging from simple heuristics and sparse embedding comparisons to more sophisticated dense approaches.

5. **Chunk Sorting** defines the rules for ordering the chunks based on their ranks. The most common scenario is to sort the chunks in ascending order of their ranks, placing the most relevant chunks at the end of the list.

6. **Chunk Assembling** joins sorted chunks into a single string using a specific template. For example, the assembler can prepend a comment with path information of the chunk's source file and join them using a separator string.

7. **Context Postprocessing** modifies the context as a unified string. For instance, line dropout can be applied by this block.

Each of these blocks has its own abstract class and serves as a foundation for concrete implementations. This design allows for several different transformations associated with a single block type to be part of a single context composer, eliminates code duplication and allows for greater flexibility in composer creation. We present the directed graph in Figure 5.1 that visualizes the order in which these instances can be applied.

5.1.2 List of Context Composers

All context composers utilized in the Research Investigation chapter adhere to two standard preprocessing steps: the removal of empty files and the normalization of all line endings to the Line Feed character. These shared preprocessing steps ensure consistency across all experiments. We present the following list to provide a comprehensive overview of the context composers implemented and employed in this research, along with a justification of their design choices.

1. **File-Level** produces an empty context. We use this composer as a baseline for comparison and to highlight the differences introduced by the repository context.
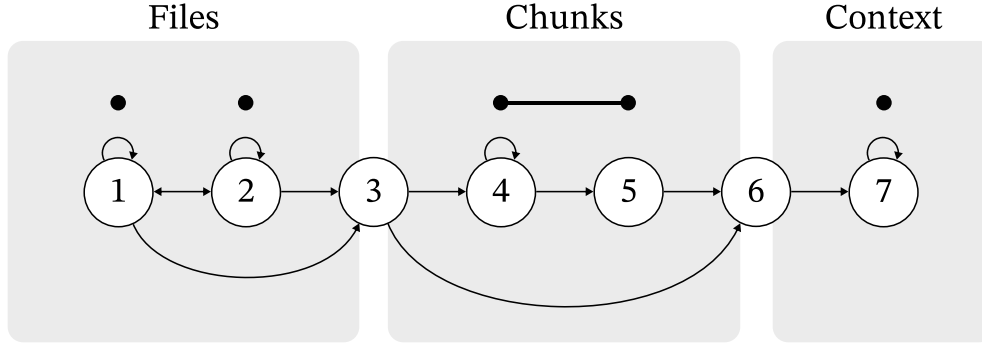
■ **Figure 5.1** Order of composer blocks. Nodes represent block types: ① File Filtering, ② File Preprocessing, ③ File Chunking, ④ Chunk Ranking, ⑤ Chunk Sorting, ⑥ Chunk Assembling, and ⑦ Context Postprocessing. Edges indicate the permissible sequence of block application. The data format is denoted by three frames: Files, Chunks, and Context. The bold subgraph highlights blocks that may be omitted.

2. **Path Distance** constructs the context using only files with the `.py` extension. The selected files are sorted in descending order according to their path distance from the completion file. For files with identical path distances, a secondary sorting criterion is applied using the Intersection over Union (IoU) metric, which equals to the number of lines shared with the completion file divided by the total number of unique lines in both files. The IoU calculation considers lines with leading and trailing whitespace removed and includes only those lines that are at least five characters in length after whitespace removal. We regard this composer as a representative of simple sparse RAG systems and designate it as a secondary baseline.

3. **Lines IoU** is similar to the Path Distance method, but omits the initial sorting by path distance. Instead, files are ranked directly using the IoU metric. We add this composer to the framework as it expands upon the Path Distance and redefines its relevance function, thereby strengthening the grounding of the research conclusions.

4. **Code Chunks** removes all docstrings, comments, and import statements from the context produced by Path Distance. We include this composer to assess the impact of non-functional code elements in context.

5. **Half-Memory** begins with the context generated by Path Distance. Each line is independently removed with a dropout probability of 0.5, maintaining the overall saturation of the context window. By presenting this composer, we test the model's ability to extract useful information from dysfunctional code.

6. **Declarations** extends Path Distance by filtering out all non-declarative elements, retaining only function and class declarations. We consider this

composer to evaluate whether the declarations are sufficient to achieve a functional understanding of the code. It is important to note that this type of context includes a larger number of ICL demonstrations of inferior quality compared to Path Distance.

7. **Text Chunks** uses Path Distance as the base composer, but removes all code from the context, leaving only docstrings and comments. We employ this composer to address the setting opposite to that of Code Chunks.

8. **Text Files** constructs the context using files with the extensions `.json`, `.yaml`, `.yml`, `.sh`, `.md`, `.txt`, and `.rst`. The selected files are grouped in ascending order of relevance: [`.json`], [`.yaml`, `.yml`], [`.sh`], [`.md`, `.txt`, `.rst`]. Within each group, files are further sorted in descending order according to their path distance from the completion file. This composer enables us to assess the significance of the context based on standalone documentation and configuration files.

9. **Random Files** constructs the context by randomly ordering all files from the repository snapshot. We introduce this composer to encompass the scenario of a non-empty repository context that is independent of the completion file.

10. **Random `.py`** selects only files with the `.py` extension and arranges them in a random order. We design this composer to resemble Random Files, ensuring that it includes only inlier ICL demonstrations for the completion file.

11. **Random Tokens** constructs the context by sampling a sequence of non-special tokens at random, with each token selected independently and with equal probability. We intend for this composer to test the model's robustness to such context and to represent the scenario of a full context window devoid of any (relevant) repository context.

12. **Duplication** constructs the context by repeatedly concatenating the content of the completion file until the maximum context window size is reached. We utilize this composer as a sanity check for our experiments, to examine the effect of data leakage, and to evaluate the model's capability to copy ground-truth completions.

13. **Leak** begins with the context produced by Path Distance. The completion file is randomly divided into five segments at newline characters, which then disjointedly replace context lines at random positions, approximately preserving the original token count. By including this composer, we inspect a more moderate data leakage scenario and evaluate the model's ability to extract ground-truth lines. (Sapronov et al. 2025)

5.1.3 Order Modifications

To increase the diversity of experiments conducted to address RQ.B1 and RQ.B2, we consider two additional file ordering modifications alongside the *original* mode, as illustrated in Figure 5.2:

- *reversed*: The retrieved files that fit within the model's context window are reordered in reverse. Consequently, the most relevant file appears at the beginning of the context window. We employ this mode to investigate the impact of demonstration permutations on ICL capabilities.

- *irrelevant*: The order of retrieved files is reversed prior to the context length cut-off, resulting in a complete reversal of the context, with the most irrelevant files positioned at the end of the context string. It enables us to evaluate the significance of irrelevant repository context in light of our research questions.
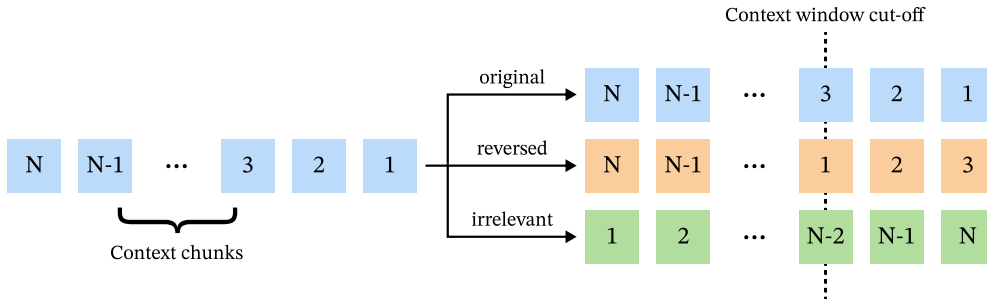
Figure 5.2 Three modes of context ordering. The numbers in boxes indicate the rank of the chunks, with smaller numbers denoting higher relevance.

5.1.4 Examples

In addition to the blocks used to create composers employed in our experiments, we provide further exemplary implementations in the attached code, totaling 30 distinct blocks. The repository also includes a demonstrative Jupyter Notebook that describes these blocks and offers concrete usage examples of the package. We hope that this foundation advances further research in this area and enhances the clarity of our experiments.

5.2 Training Data

The dataset was provided by JetBrains Research, which initialized this project, and is based on the methodology outlined in Bogomolov et al. (2024). It was

constructed by traversing the GitHub histories of Python repositories and applying permissive license filtering to sample completion files and their corresponding parent commits. Each data point consists of a pair: a list of Python completion files and a repository snapshot that captures the state of the repository at the time the completion files were added. The snapshot includes all text files except for the completion files themselves. To prevent contamination of the benchmark used in the evaluation, any repositories present in the benchmark were excluded from the training data.

To delineate the scope of this work, we ought to note that the corpus described thus far was entirely provided by JetBrains Research and was not generated by the author of this thesis. Conversely, the subsequent processing steps applied to this dataset represent the original contributions of the author.

We apply multiple filtering criteria to obtain the dataset with greater relevance and quality. First, all commits made prior to 2010 are excluded. Second, completion files with lengths outside the closed interval [800, 25000] characters are removed. Third, to eliminate redundancy, a simple deduplication strategy is employed on completion files based on the file name and the name of the repository to which they belong. Finally, up to 1000 of the most recently updated unique completion files are selected from each repository. The remaining repository snapshot is retained without additional processing. (Sapronov et al. 2025)

The resulting corpus comprises 1,640 repositories, 160,801 commits, and 361,052 completion files. The completion files contain a total of 1.7 billion characters, while the repository snapshot files contain 4.8 trillion characters. To ensure the reproducibility of our research, we provide the identifiers for all data points within this corpus.

A subset of 2560 samples is randomly selected to form the validation set. During this process, we ensure that the repositories included in the training and validation sets do not overlap, and that no more than five different completion files are sourced from the same repository. After sampling for validation, the remaining data is sufficient to cover more than 250,000 unique completion files. We then apply the pre-composition procedure using the context composers listed in Section 5.1.2. This allows us to perform this operation only once and reuse the smaller produced datasets for several experiments. Moreover, instead of saving the entire context string, which can reach the length of the total number of characters used in the repository, we apply a 16K-token cut-off, ensuring that the resulting datasets range from 4 to 17 GB in Parquet format. A demonstration of the first five data points of each dataset is provided in the accompanying thesis repository.

It is important to note that the pre-composition of both the training and validation sets is based on the entire completion file. This approach is motivated by the inefficiency of selecting target lines to possess file prefixes during the training and validation processes. For better clarity, one can consider training on a single line from each completion file; this results in the degrada-

tion of the gradient approximation proportional to the number of completion lines, or an increase in the number of forward and backward passes to the same degree.

5.3 Fine-Tuning Pipeline

For the purposes of this research, we developed a fine-tuning pipeline aimed at adapting Code LLMs to the repository-level context.

From the data perspective, it supports the integration of pre-composed datasets obtained from the aforementioned context composition framework. We design the train-validation split to eliminate data leakage from samples gathered from the same repositories. By enabling configurable tokenization we ensure context window saturation and facilitate the application of gradient masking.

The pipeline integrates models from the Hugging Face Hub[1]. It supports adapters that can augment the model architecture, freeze a subset of parameters, or modify the optimizer initialization. Models are dynamically loaded onto a single device based on hardware availability. Efficient attention implementation and parameter data types are selected using the same principle.

We implement the training loop in pure PyTorch[2], as it is a highly adaptable framework that provides a substantial level of flexibility and performance. The pipeline supports cosine learning rate scheduling with linear warmup, gradient scaling, clipping, and accumulation. The user can periodically invoke two validation loops: one on the original dataset split and another on an additional provided dataset. The latter allows metrics to be grounded on baseline composition strategy instead of solely monitoring the in-distribution capabilities of the model.

During the training process, the pipeline produces a set of configurable metrics and statistics for both the training and validation data. These include cross-entropy, Exact Match, Top-k Accuracy, learning rate tracking, epoch and token number counting. We log all these values to ensure adequate monitorability of the experiments. Furthermore, we decompose their values based on the types of tokens to which they are applied and denote these types as follows:

- **Attached:** Tokens whose losses participate as leaf nodes in automatic differentiation for gradient computation.

- **Detached:** Tokens whose losses are zeroed out during gradient masking and serve as a complement to the attached tokens.

- **Completion:** Tokens of the completion file.

---

[1] https://huggingface.co/
[2] https://pytorch.org/

- ▪ **Context:** Tokens of the repository context.

- ▪ **Full:** All tokens.

In addition to statistics and metrics, the pipeline provides logging of the standard output and error streams. Checkpointing of the model and optimizer states is also supported.

The entire pipeline utilizes a modular structure, which enhances code maintainability and extendability. We design it in an end-to-end manner, requiring only a pre-composed dataset, a set of YAML configuration files, and a single command line from the user. We accompany all components that exhibit stochastic behavior with a seed parameter, ensuring the reproducibility of all experiments.

5.4 Tools

In the context composition framework, we integrate Hugging Face tokenizers from the Transformers[3] library to compute token-related statistics, and Tree-sitter[4] to decompose Python code into its parts. Both libraries are employed in the concrete implementations of the blocks, although their usage can be omitted without disrupting the main abstract framework. The OmegaConf[5] configuration system is utilized to manage the initialization of the composers from YAML configuration files.

The pipeline inherits requirements from the context composition framework as it utilizes its functions. Additionally, we employ Datasets[6] and Accelerate[7] from the Hugging Face ecosystem to provide peripheral support to the Transformers library. Pandas[8] is used to load pre-composed datasets from the Parquet format. We extend the configurative capabilities of OmegaConf and enhance the pipeline's usability by using the Hydra[9] package. PyTorch, as mentioned in the previous section, is utilized to implement the training loop. The tqdm[10] library is employed to display progress bars for all prolonged processes. Weights & Biases[11] is used to track metrics and statistics in real-time.

We use a single 8×H200 node, with each GPU possessing 141 GB of VRAM, to conduct all experiments and execute evaluation script.

---

[3]`https://github.com/huggingface/transformers`
[4]`https://tree-sitter.github.io/tree-sitter/`
[5]`https://github.com/omry/omegaconf`
[6]`https://github.com/huggingface/datasets`
[7]`https://github.com/huggingface/accelerate`
[8]`https://github.com/pandas-dev/pandas`
[9]`https://github.com/facebookresearch/hydra`
[10]`https://github.com/tqdm/tqdm`
[11]`https://github.com/wandb/wandb`

Chapter 6

Research Investigation

In this chapter, we describe the research aspect of the thesis. The list of research questions is presented first, followed by an elaboration of the unified experimental design and individual discussions. Supplementary results that emerged during the investigation, along with a prospective view on future work, are included at the end of the chapter in their respective sections.

The contributions outlined in this chapter are: a unified experimental design, four research questions answered along with empirical evidence, a detailed examination of the effects of compositions, a reported case of resource efficiency for repository-level pre-training, and other supplementary results that emerged during the research.

6.1 Research Questions

To investigate the nature of in-context learning capabilities of Code LLMs, we divide four research questions into two distinct settings: in-distribution and out-of-distribution context length. The former implies an investigation of the model's capabilities within a pre-trained model's context window and covers the following research questions:

RQ.A1 Composition Impact on Inference: Does the quality improvement of code completion depend on the composition strategy employed during model inference?

RQ.A2 Fine-Tuning on Compositions: Does fine-tuning a pre-trained Code LLM with a specific context composer enhance the subsequent quality of code completion?

We consider the latter setting to explore the repository-level pre-training stage and to observe phenomena related to context window extension. We define the following research questions related to these objectives:

RQ.B1 Effect of Context Extension: Does the repository-level pre-training affect the in-context learning abilities of the model developed during earlier stages?

RQ.B2 Influence of Composition on Context Extension: Does the quality improvement of code completion depend on the context composition approach used during the repository-level pre-training stage?

6.2 Experimental Design

In this section, we describe the unified experimental design for all four research questions, including training and evaluation components. Specific details are deferred to the subsequent sections dedicated to each research question separately.

6.2.1 Training

We utilize DeepSeek-Coder-Base 1.3B (Guo et al. 2024) to address RQ.A1 and RQ.A2, as it served as a robust foundation for code completion research at the inception of this work. It supports a context window size of 16K tokens, allowing us to leverage a substantial portion of the repository data. For experiments concerning RQ.B1 and RQ.B2, we employ OpenCoder-1.5B-Base (Huang et al. 2024), the only modern Code LLM released without undergoing a repository-level pre-training stage. This model supports a maximum context window size of 4K tokens, providing an opportunity to explore context extension fine-tuning. For this purpose, we adjust the base frequency of RoPE from $\theta_{\text{base}} = 10{,}000$ to $\theta_{\text{base}} = 500{,}000$.

An input sequence is derived from each row of the pre-composed dataset by independently tokenizing the context string and the completion file. Through this process, we ensure that the completion sequence does not exceed 4,096 tokens and that the total length of the concatenated input remains within 16,384 tokens. To enforce these constraints, we apply truncation from the left for the context and from the right for the completion. Given that most composed contexts exhibit high token saturation, we maintain a context-to-completion token ratio exceeding $3 : 1$. (Sapronov et al. 2025) This approach is employed for all setups with a 16K context length. The file-level training applies the same approach but omits the context string, resulting in a maximum context length of 4K tokens. We also note that no data packing techniques are applied. Furthermore, to ensure a uniform setup across all research questions and experiments, FIM is not used, as it is not supported by OpenCoder.

We use consistent hyperparameters across all experiments, as they have been established as optimal for both models. Specifically, the optimization process is conducted using the AdamW optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a weight decay of 0.01. A batch size of 128 is employed, with a micro-batch size of 1 to accommodate hardware constraints. To ensure stable training, gradient clipping is applied with a maximum gradient Euclidean norm of 2. The learning rate is managed using a cosine decay scheduler with a linear warm-up phase, where the maximum learning rate is set to 5×10^{-5}. The warm-up phase lasts for 256 iterations, after which the learning rate follows a cosine decay schedule for 3244 additional iterations, reaching a minimum value of 5×10^{-8}. (Sapronov et al. 2025)

We apply model checkpointing and validation loops every 128 optimization steps. We also monitor the metrics for both the original composed validation dataset split and the Path Distance baseline composer with a length truncation of 16K tokens. This approach enables us to track both the in-distribution and out-of-distribution dynamics of model capabilities throughout the training.

Early stopping is applied after 512 iterations, as we find this sufficient to address the stated research questions. This, in combination with the context window size, results in the utilization of approximately 73 million training tokens for fine-tuning on the File-Level and 1 billion tokens for all other composers. The order of the data points is deterministically shuffled in the same manner for all runs to maintain consistency for subsequent comparisons.

6.2.2 Evaluation

The Project-Level Code Completion task from the Long Code Arena benchmark (Bogomolov et al. 2024), detailed in Section 3.4.3, is selected to establish an evaluation setup. We focus on the *large-context set*, emphasizing two primary line types: *inproject* and *infile*. Our choice is motivated by the need for context window saturation and a more precise distinction between the target lines. The most reportable metric for code completion, Exact Match, is employed to evaluate the models' performance on these line types separately and is presented on a scale from 0 to 100. We obtain completion lines from the model using a greedy decoding strategy in order to keep the evaluation process as simple as possible.

We select two baseline composers from the aforementioned list: File-Level (FL) and Path Distance (PD). For RQ.A2, RQ.B1, and RQ.B2, we denote the composer used to obtain a model checkpoint as the original composer (Or).

In contrast to the training phase, we utilize only file prefixes instead of integral completion files to derive the context for evaluation and to better reflect real-world usage scenarios. This approach involves invoking composers for each target line independently, thereby generating multiple distinct context

strings for the same completion file. Duplication and Leak composers are exceptions to this rule due to their requirement for the full completion file content. These composers are outlined separately in the following sections.

Differences in metric values compared to Sapronov et al. (2025) are evident. This is because this thesis employs a separate evaluation script, independent of the proprietary one used within JetBrains Research, allowing us to publish all code under the permissive MIT license. Two factors contribute to these differences. First, we use teacher forcing, which combines effectively with the EM metric and enables a greater number of evaluation runs at a reduced cost. Second, the EM version used here does not remove whitespace prior to calculation. These factors result in a stricter metric assessment of the model's performance. Since all modifications are applied uniformly and all numbers are reported under the same setup, the metric values' bias is consistent and does not affect the conclusions drawn from the experiments.

6.3 In-Distribution Context Length

In this section, we address the first pair of research questions: RQ.A1 and RQ.A2. The evaluation and training setups employed here do not presume overstepping the context window boundaries. To conduct all experiments on long context sequences, we use DeepSeek-Coder-Base 1.3B, as it is sufficiently pre-trained on repository-level data and supports a context length of 16K tokens.

6.3.1 Composition Impact on Inference

To measure the impact of context composition strategies on the quality of code completion during Code LLM inference, we evaluate the DeepSeek-Coder-Base 1.3B model (Guo et al. 2024) using all previously listed composers. The results are presented in Table 6.1.

A noticeable difference is observed, allowing us to infer that the context provided to the model during inference has a significant impact on its performance. Some detailed findings are as follows:

1. In-context learning enables the model to leverage any form of meaningful context to improve *infile* completion, indicating that API declarations are not the sole source of metric improvements and that project-level information provides auxiliary grounding for generation.

2. The Lines IoU approach proposes the most effective relevance function for retrieval-augmented generation among the presented methods, demonstrating that the model can achieve the same level of *inproject* performance as *infile*, given the context of sufficient quality.

Inference Composer	inproject	infile
File-Level	25.5	31.1
Path Distance	42.6	44.6
Lines IoU	**46.4**	**46.2**
Code Chunks	43.0	45.0
Half-Memory	33.9	36.8
Declarations	27.8	32.3
Text Chunks	26.1	31.2
Text Files	25.7	31.8
Random Files	28.1	33.1
Random `.py`	29.9	33.5
Random Tokens	23.1	29.6
Duplication	86.8	86.8
Leak	78.5	78.3

■ **Table 6.1** Exact Match scores of DeepSeek-Coder-Base 1.3B benchmarked on the LCA

3. The examination of Declarations indicates that function and class declarations alone do not provide a strong foundation for in-context learning capabilities; the functional code within their bodies is more important.

4. A comparison between Random `.py` and Text Files suggests that configuration and documentation files are less important for the model than randomly selected code examples in the context. The same observation applies to code comments and documentation strings produced by Text Chunks.

5. Even randomly selected files enhance the code completion capabilities of the model.

6. Random Tokens degrade performance only slightly compared to the File-Level composer, indicating that the model is highly, though not completely, robust to the presence of noisy context.

7. The model is capable of effectively copying (Duplication) and extracting (Leak) ground-truth lines from the context without requiring any additional training.

6.3.2 Fine-Tuning on Compositions

To estimate the impact of the context composition strategy employed during model fine-tuning, we subject the same model from the previous section to training with various composers. After fine-tuning, we benchmark the model to produce Table 6.2.

Fine-Tuning Composer	inproject			infile		
	FL-16K	PD-16K	Or-16K	FL-16K	PD-16K	Or-16K
No Fine-Tuning	25.5	42.6	—	31.1	44.6	—
File-Level	25.5	42.4	25.5	31.0	44.7	31.0
Path Distance	25.4	39.7	39.7	31.3	43.7	43.7
Lines IoU	25.6	40.3	43.8	31.4	43.7	45.0
Code Chunks	25.5	39.0	39.3	31.5	42.6	42.5
Half-Memory	25.0	38.4	32.1	31.3	42.9	35.2
Declarations	25.6	40.7	27.1	31.3	43.8	32.3
Text Chunks	25.5	39.2	24.3	31.3	42.7	30.0
Text Files	25.6	40.0	23.7	31.5	43.5	30.1
Random Files	25.8	39.8	25.3	31.3	42.7	31.0
Random .py	25.2	40.1	27.1	30.8	43.5	31.7
Random Tokens	25.1	35.5	23.9	31.0	38.3	29.8
Duplication	25.6	42.8	86.9	31.1	45.0	86.9
Leak	25.0	41.1	79.2	30.6	43.1	78.9

Table 6.2 Exact Match scores of DeepSeek-Coder-Base 1.3B fine-tuned with different context composition strategies. The rows indicate the composer used to obtain the model checkpoint. The columns represent different evaluation setups: PD for Path Distance, FL for File-Level, and Or for the composer used during fine-tuning. All sequences are truncated to 16K tokens. The blank cells refer to the Table 6.1.

The marginal differences in metric values suggest that this particular model is sufficiently trained and is not significantly affected by subsequent fine-tuning.

One might argue that the number of training steps is insufficient. To address this concern, we monitor the learning curves during training and observe that the metric improvements are too small to justify further scaling of this experimental direction. For instance, the best linear slope, 1.28×10^{-5} of EM, is achieved by the Lines IoU composer during the first 512 iterations. Applying a linear extrapolation of this slope implies that the model would require more iterations to gain 5 EM points than permitted by the learning rate scheduler. Furthermore, since the proxy metric for EM is cross-entropy loss, whose learning curves follow power-law scaling (Kaplan et al. 2020), this extrapolation remains extremely optimistic.

Additionally, we explored different setups to address the issue of the model's robustness to this type of fine-tuning. First, a different training dataset with lower data diversity was used earlier, which resulted in overfitting after the first epoch. Second, we experimented with other hyperparameter choices without success. Third, to eliminate researcher proficiency bias, the experiments were independently reproduced from scratch by another person, which is beyond the scope of this thesis.

6.4   Out-of-Distribution Context Length

In this section, we address the second pair of research questions: RQ.B1 and RQ.B2. For both of them, we use OpenCoder-1.5B-Base and apply context window extension fine-tuning with sequences exceeding the original context window size of 4K tokens. The only exception is the File-Level composer, which we use to support our findings on the resource efficiency of repository-level pre-training under the in-distribution context length composer.

6.4.1   Effect of Context Extension

We assess the impact of repository-level pre-training on the previously obtained in-context learning capabilities of the model using the composition strategies accessible to OpenCoder-1.5B-Base, which has an original context size limitation of 4K tokens. The summarized results are presented in Table 6.3. For the extended version, which includes the effects observed with *reversed* and *irrelevant* order modifications, refer to Table A.1.

Context Extension Composer	inproject		infile	
	FL-4K	PD-4K	FL-4K	PD-4K
No Extension				
$\theta_{\text{base}} = 10{,}000$	26.1	37.1	32.7	38.9
$\theta_{\text{base}} = 500{,}000$	13.4	15.9	14.7	12.6
File-Level				
$\theta_{\text{base}} = 10{,}000$	26.2	37.2	33.2	38.6
$\theta_{\text{base}} = 500{,}000$	25.9	36.6	33.3	38.6
Path Distance	25.8	37.6	33.4	39.0
Other Composers	25.7 – 26.2	36.8 – 37.5	32.7 – 33.4	38.5 – 39.3
Duplication	19.2	28.4	24.2	26.2
Leak	25.1	35.1	31.2	35.2

■ **Table 6.3** Exact Match scores of OpenCoder-1.5B-Base under different evaluation setups. The original model's performance is denoted by No Extension. The other rows represent the checkpoints obtained with the respective composers. Context extension is performed with 4K-token sequences for the File-Level and 16K-token sequences for all other composers. The column notations are consistent with the previous table.

We draw three main observations from the table to address RQ.B1:

1. ABF without additional fine-tuning significantly impairs the ICL capabilities.

2. The context extension procedure effectively restores model performance after RoPE frequency scaling. Even the File-Level composer, which utilizes input sequences up to 4K tokens, enables the model to adapt to the new θ_{base} value and maintain the same performance level as the original model.

3. Data leakage of ground-truth lines impedes the recovery of the model's ICL capabilities. This effect is particularly pronounced for the Duplication composer.

6.4.2 Influence of Composition on Context Extension

In this section, we address the final research question RQ.B2 by evaluating the checkpoints obtained after context extension fine-tuning, using various composers to construct the training context. Our conclusions are drawn from Table 6.4, which is a subset of the comprehensive Table A.1.

| Context Extension Composer	inproject			infile		
	FL-4K	PD-16K	Or-16K	FL-4K	PD-16K	Or-16K
No Extension						
$\theta_{\text{base}} = 10,000$	26.1	0.0	—	32.7	0.0	—
$\theta_{\text{base}} = 500,000$	13.4	8.3	—	14.7	3.6	—
File-Level						
$\theta_{\text{base}} = 10,000$	26.2	0.0	24.4	33.2	0.0	29.4
$\theta_{\text{base}} = 500,000$	25.9	44.9	26.1	33.3	44.7	33.5
Path Distance	25.8	48.7	48.7	33.4	49.2	49.2
Lines IoU	25.9	48.7	51.4	33.4	48.9	50.2
Code Chunks	25.7	48.3	48.1	33.0	48.3	48.8
Half-Memory	25.8	47.8	37.6	32.7	48.0	40.1
Declarations	26.1	47.3	28.2	32.9	46.9	35.0
Text Chunks	26.2	47.9	27.2	32.9	47.8	33.4
Text Files	25.9	47.3	27.1	33.3	48.1	34.0
Random Files	26.1	48.5	29.6	33.0	48.7	35.5
Random .py	25.7	48.8	32.5	32.8	48.8	36.0
Random Tokens	26.1	45.0	26.3	32.9	45.1	33.3
Duplication	19.2	34.8	95.3	24.2	28.1	94.2
Leak	25.1	45.2	86.9	31.2	43.8	85.8

■ **Table 6.4** Exact Match scores collected from the long-context evaluation of composer choices employed during the repository-level pre-training stage of OpenCoder-1.5B-Base. All clarifying remarks are consistent with the previous table. Blank cells indicate the absence of a repository-level pre-training stage in OpenCoder's development pipeline.

To summarize the outcomes of our study, we present the following observations:

1. As expected, the original model is unable to utilize the 16K context window. The preliminary base frequency adjustment provides only a negligible improvement and necessitates an additional optimization phase.

2. The context composition strategy employed during the repository-level pre-training stage has only a marginal impact on the final model quality. This finding suggests that adaptation to the RoPE adjustment is the primary driver of long-context improvements.

3. It is possible to significantly reduce computational requirements while still achieving competitive results at the repository-level pre-training stage. For instance, file-level training remains highly effective, even without any repository context and with a context window size limited to 4k tokens.

4. The Duplication composer proves its uncompetitiveness even when compared to the entirely irrelevant context obtained from Random Tokens. This suggests that the data must present at least some degree of challenge to the model in order to foster the development of additional capabilities.

The main practical contribution of this research is the identification of a low-resource approach to repository-level pre-training. Context extension fine-tuning can be restricted to the Path Distance composer, requiring only 1B training tokens, while the File-Level presents an even more resource-efficient alternative, necessitating just 73M training tokens. We further demonstrate the utility of this setup by comparing it with DeepSeek-Coder-Base 1.3B, which used 8.4B tokens for repository-level pre-training, and Qwen2.5-Coder-1.5B (Hui et al. 2024), which employed 300B tokens during the same stage. Figure 6.1 visualizes this comparison, and Figure 6.2 illustrates the performance scaling beyond the training context window for all four models.

We note that the same model's extrapolation behavior, as shown in Figure 6.2, was partially reported by Rozière et al. (2023). Specifically, when the model is initialized with RoPE from scratch and trained on sequences up to N tokens, it is unable to extrapolate beyond its pre-training length. However, after applying ABF and optimizing for a few steps on sequences up to $M > N$ tokens, the model learns to extrapolate not only beyond N tokens but also for values greater than M. While we confirm this finding, the core novelty of our evaluation lies in the case where $M = N$, which demonstrates the preservation of described properties.
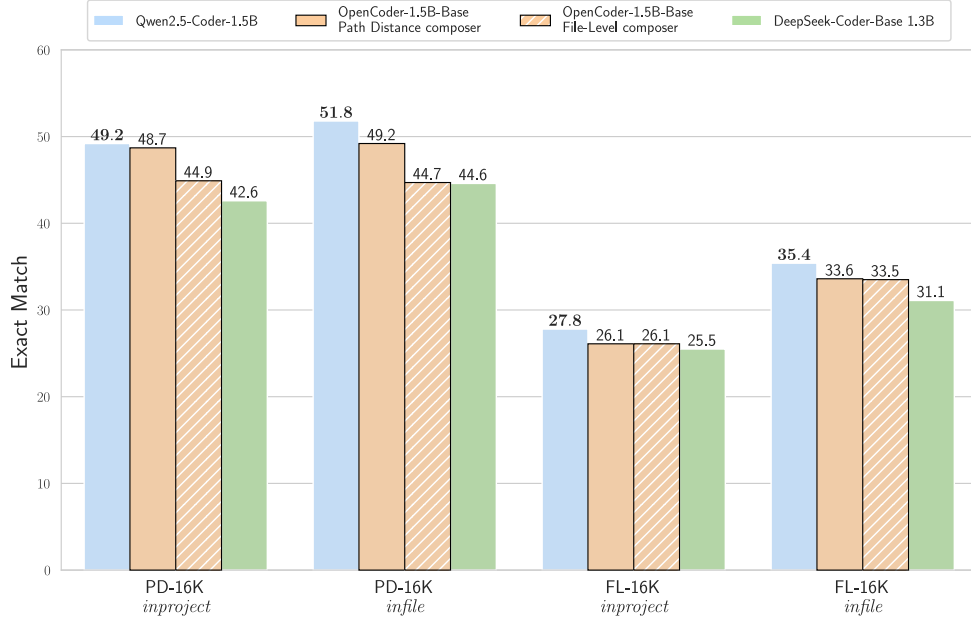
■ **Figure 6.1** Comparison of our two main checkpoints with other existing Code LLMs of the same weight class. The PD-16K setup is used to produce the metric assessment.

6.5 Supplementary Results

To diversify the research, in this section we provide a supplementary analysis of the experiments focused on investigating the effects of order modifications and gradient masking.

6.5.1 Influence of Order Modifications

The same conclusions regarding the preservation of ICL capabilities after the repository-level pre-training stage and the impact of composition on context extension apply to both the *reversed* and *irrelevant* order modifications. For example, although the checkpoint obtained with Lines IoU *irrelevant* context extension during inference yields an EM score that is half that of its *original* and *reversed* counterparts, it achieves equivalent performance when the composer is switched to Path Distance. We attach Table A.1 to present all metric values related to this observation.

6.5.2 Gradient Masking

To avoid training on out-of-distribution lines, we consistently apply gradient masking to non-completion tokens when training with composers that produce distributions dissimilar to the completion file. Although this design choice
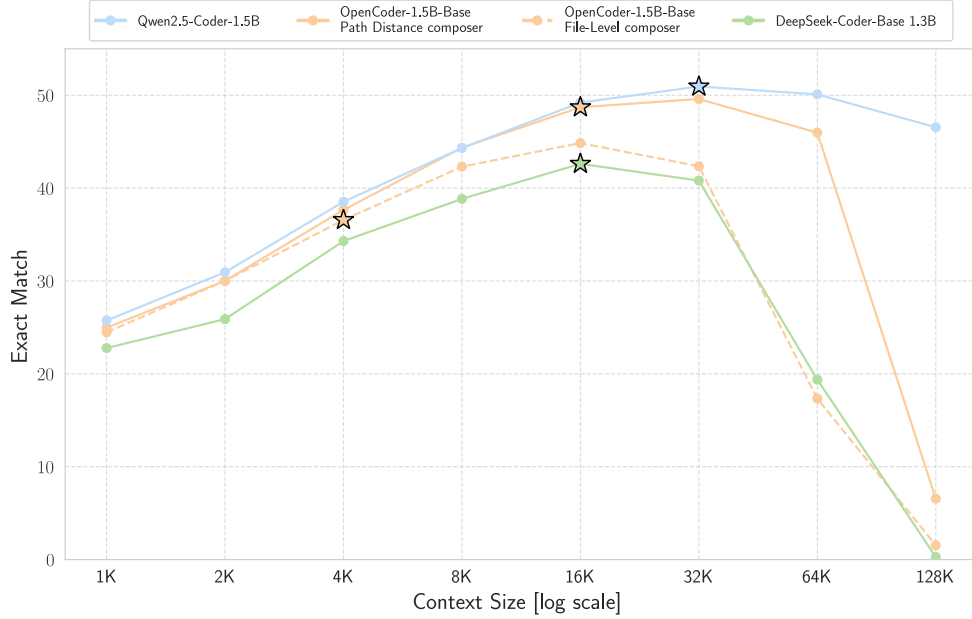
■ **Figure 6.2** Evaluation of the performance scaling beyond the context extension window. The *inproject* line type from the LCA benchmark is selected for visualization; the corresponding Figure A.1 presents results for the *infile* category. "1K" refers to 1,024 tokens. The ☆ markers denote the context length used during repository-level pre-training stage.

is intuitive, the applicability of this technique to runs with inlier composers remains an open question.

To examine the impact of including all losses in the optimization process for the inlier composers, we conduct training runs both with and without masking. Table A.2 shows the results of this experiment for the DeepSeek-Coder-Base 1.3B, and Table A.3 for the OpenCoder-1.5B-Base model. We then consider all these values except those for the Duplication composer to perform a one-sided paired t-test with the alternative hypothesis that the mean Exact Match of models trained with gradient masking is lower than that for full loss training. The reported p-value is 1.9×10^{-5}, so we reject the null hypothesis at a significance level of $\alpha = 0.05$. Figure 6.3 visualizes the distribution of these metric differences.

Another observation is that disabling gradient masking eliminates the detrimental properties of the Duplication composer. We attribute this to the model being required to perform well on the first instance of the completion file presented in the context, which does not have access to the ground-truth lines to copy from.
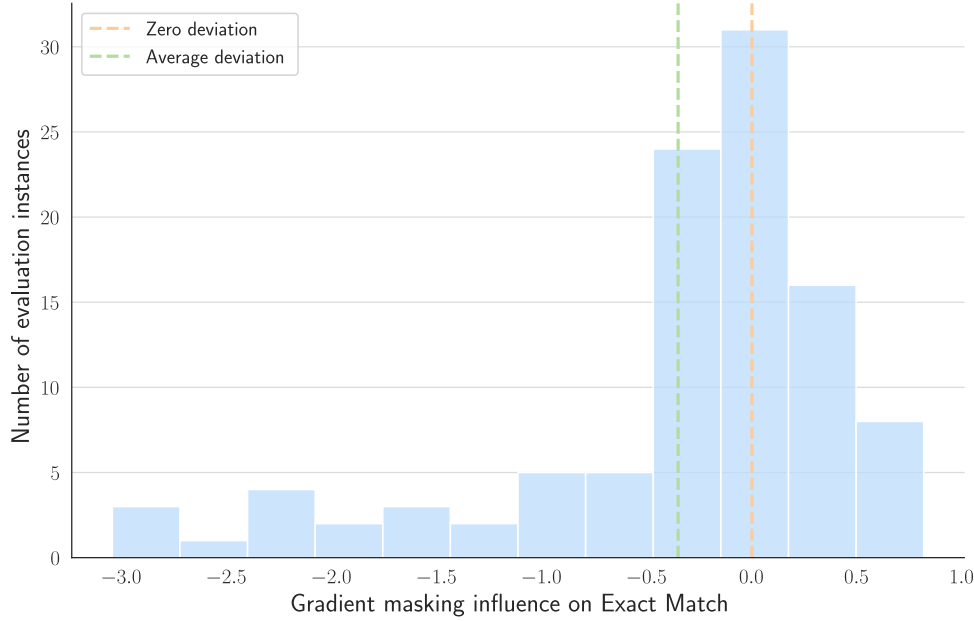
Figure 6.3 Histogram of the Exact Match differences between paired runs on inlier composers, with the Duplication composer excluded. Negative values indicate cases where gradient masking reduces model performance. The sample mean is -0.35 EM points on a scale from -100 to 100.

6.6 Limitations and Future Work

Although we have addressed all research questions and inferred various insights, several directions for scaling the conducted experiments and new prospects for extending the work further remain open.

The empirical results provided are based on the utilization of two Code LLMs, which narrows the generalizability of our research. This limitation is particularly pronounced for RQ.B1 and RQ.B2, as OpenCoder is the only modern Code LLM released without repository-level pre-training, leaving no alternatives yet to scale this direction. We conclude that a better practice for model providers would be to release all major model checkpoints obtained throughout the development pipeline, as this would foster the growth of the field by outsourcing less compute-intensive exploration to the open research community.

The evidence of a repository-level pre-training setup that requires fewer resources and achieves competitive results suggests significant potential for future exploration of the optimal recipe for this stage. First, a mixed approach combining different composers to benefit from their individual factors can be explored. Second, other RoPE extension methods can be tested using the same

experimental design. Third, a proper metric for token efficiency can be developed to assess the trade-off between the compute used and the performance achieved.

Another important line of future work is to verify the findings on other code-related tasks and test whether they can be transferred to the Natural Language Processing domain.

Conclusion

In conclusion, the objectives of this thesis have been achieved. The Conceptual Framework established a theoretical framework and provided an overview of the current state of project adaptation for the code completion task. This chapter supplies the necessary background to assess the practical contributions of the work, which are detailed in Applied Research. Specifically, four research questions regarding the impact of context composition strategies on code completion quality were addressed, including the examination of three different settings: inference-only, fine-tuning, and context extension. To accomplish this, a context composition framework and a fine-tuning pipeline were implemented.

The conducted research yielded several significant findings regarding repository-level code completion. First, the results substantiate the crucial role of repository context in enhancing completion quality under the inference of a pre-trained Code LLM setup. Second, DeepSeek-Coder-Base 1.3B possesses sufficient training, with its in-context learning capabilities exhibiting only marginal variability following downstream fine-tuning on different context composition strategies. Third, it was demonstrated that in-context learning capabilities are not affected by a properly conducted repository-level pre-training stage and are maintained after the model acquires a comprehension of long-context data. Fourth, the context composition strategy employed during the aforementioned stage demonstrates minimal impact on final model quality, suggesting that RoPE adjustment serves as the primary driver of long-context improvements.

Additional observations were also inferred beyond the originally stated research questions. Most notably, the computational requirements of the repository-level pre-training stage can be substantially reduced while maintaining competitive results not only with the extended context window but also with the original context length and a relatively small number of training tokens.

Overall, this thesis contributes to the evolution of code completion models by underscoring theoretical foundations in the conceptual framework and offering valuable empirical insights for practitioners through comprehensive research on context composition techniques for repository-level understanding.

Appendix

A.1 Supplementary Figures and Tables
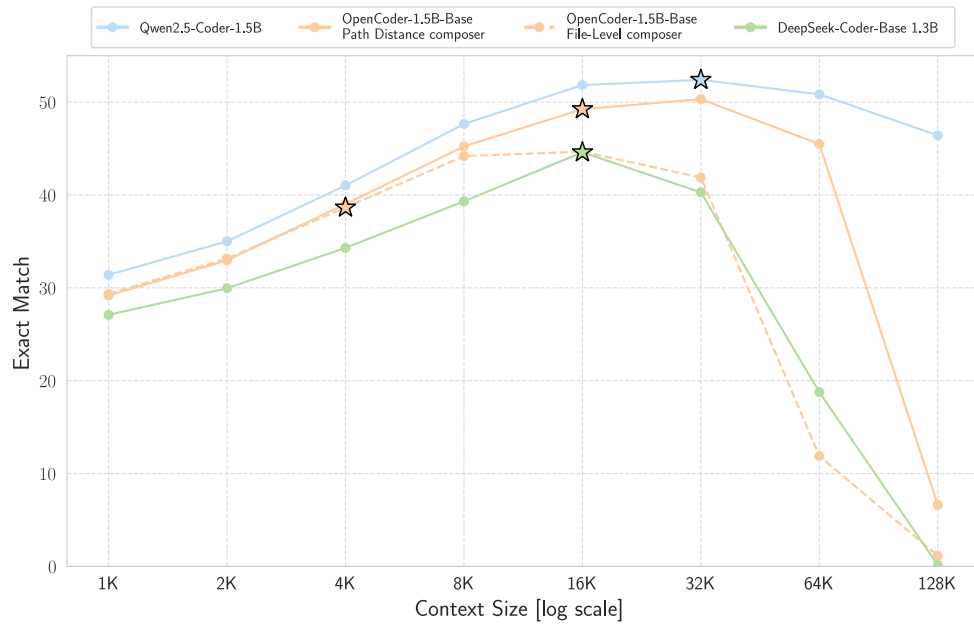
■ Figure A.1 Evaluation of the performance scaling beyond the context extension window. The *infile* line type from the LCA benchmark is selected for visualization; the corresponding Figure 6.2 presents results for the *inproject* category. "1K" refers to 1,024 tokens. The ☆ markers denote the context length used during repository-level pre-training stage.

Context Extension Composer	inproject				infile			
	FL-4K	PD-4K	PD-16K	Or-16K	FL-4K	PD-4K	PD-16K	Or-16K
No Extension								
$\theta_{\text{base}} = 10{,}000$	26.1	37.1	0.0	—	32.7	38.9	0.0	—
$\theta_{\text{base}} = 500{,}000$	13.4	15.9	8.3	—	14.7	12.6	3.6	—
File-Level								
$\theta_{\text{base}} = 10{,}000$	26.2	37.2	0.0	24.4	33.2	38.6	0.0	29.4
$\theta_{\text{base}} = 500{,}000$	25.9	36.6	44.9	26.1	33.3	38.6	44.7	33.5
Path Distance	25.8	37.6	48.7	48.7	33.4	39.0	49.2	49.2
reversed	25.9	37.1	48.6	48.9	33.0	39.2	48.7	49.4
irrelevant	26.0	37.7	48.3	27.4	33.0	38.9	48.3	33.7
Lines IoU	25.9	37.4	48.7	51.4	33.4	39.2	48.9	50.2
reversed	25.9	37.3	48.4	51.2	33.2	39.1	48.5	50.5
irrelevant	26.3	37.5	48.0	27.2	32.9	38.7	48.2	33.6
Code Chunks	25.7	37.5	48.3	48.1	33.0	38.9	48.3	48.8
reversed	26.3	37.5	48.1	48.2	33.0	38.9	48.6	48.9
irrelevant	25.7	37.6	47.8	27.6	32.3	38.4	47.6	33.4
Half-Memory	25.8	37.1	47.8	37.6	32.7	38.8	48.0	40.1
reversed	25.5	37.1	47.5	36.9	32.9	39.0	47.8	39.5
irrelevant	25.6	36.8	47.5	27.1	32.4	38.4	47.8	33.7
Declarations	26.1	37.0	47.3	28.2	32.9	38.9	46.9	35.0
reversed	25.9	37.3	47.4	29.0	33.1	38.8	47.0	35.0
irrelevant	26.1	37.0	46.9	28.4	32.7	39.0	46.9	34.5
Text Chunks	26.2	37.3	47.9	27.2	32.9	39.3	47.8	33.4
reversed	26.1	37.2	47.4	27.4	32.8	38.8	47.6	33.5
irrelevant	25.8	37.1	47.6	26.9	33.1	38.8	47.8	34.0
Text Files	25.9	37.0	47.3	27.1	33.3	39.2	48.1	34.0
reversed	26.0	36.8	47.1	27.1	33.2	38.8	48.0	33.6
irrelevant	25.9	37.3	47.6	27.1	33.2	38.9	48.0	34.0
Random Files	26.1	37.0	48.5	29.6	33.0	39.2	48.7	35.5
Random .py	25.7	37.3	48.8	32.5	32.8	38.8	48.8	36.0
Random Tokens	26.1	36.8	45.0	26.3	32.9	38.5	45.1	33.3
Duplication	19.2	28.4	34.8	95.3	24.2	26.2	28.1	94.2
Leak	25.1	35.1	45.2	86.9	31.2	35.2	43.8	85.8
reversed	24.5	35.2	44.8	89.0	30.8	35.0	43.7	87.6
irrelevant	24.8	35.3	45.0	85.7	31.2	35.2	43.8	85.1

■ **Table A.1** Extended table presenting the evaluation results for OpenCoder-1.5B-Base, which underwent the repository-level pre-training stage. A more detailed description of the evaluation setup is provided in Section 6.2.2.

Fine-Tuning	inproject			infile		
Composer & Loss	FL-16K	PD-16K	Or-16K	FL-16K	PD-16K	Or-16K
No Fine-Tuning	25.5	42.6	—	31.1	44.6	—
Path Distance						
Masked loss	25.4	39.7	39.7	31.3	43.7	43.7
Full loss	25.7	42.6	42.6	31.3	44.8	44.8
Lines IoU						
Masked loss	25.6	40.3	43.8	31.4	43.7	45.0
Full loss	25.5	42.5	46.2	31.5	45.1	46.5
Code Chunks						
Masked loss	25.5	39.0	39.3	31.5	42.6	42.5
Full loss	25.8	42.1	42.0	31.5	44.9	44.8
Random .py						
Masked loss	25.2	40.1	27.1	30.8	43.5	31.7
Full loss	25.7	41.7	28.9	31.7	45.0	33.7
Duplication						
Masked loss	25.6	42.8	86.9	31.1	45.0	86.9
Full loss	25.7	42.2	86.9	31.7	44.7	87.0

Table A.2 Exact Match scores for DeepSeek-Coder-Base 1.3B fine-tuned on composers generating inlier repository context for the completion file under two training setups: with and without gradient masking

Context Extension Composer & Loss	inproject				infile			
	FL-4K	PD-4K	PD-16K	Or-16K	FL-4K	PD-4K	PD-16K	Or-16K
No Extension								
$\theta_{base} = 10{,}000$	26.1	37.1	0.0	—	32.7	38.9	0.0	—
$\theta_{base} = 500{,}000$	13.4	15.9	8.3	—	14.7	12.6	3.6	—
Path Distance								
Masked loss	25.8	37.6	48.7	48.7	33.4	39.0	49.2	49.2
Full loss	26.1	37.4	48.6	48.6	32.8	39.2	49.0	49.0
Path Distance, *reversed*								
Masked loss	25.9	37.1	48.6	48.9	33.0	39.2	48.7	49.4
Full loss	26.3	37.3	48.7	48.6	33.1	39.0	48.9	48.6
Path Distance, *irrelevant*								
Masked loss	26.0	37.7	48.3	27.4	33.0	38.9	48.3	33.7
Full loss	26.0	37.2	48.4	26.7	33.1	38.8	48.3	33.4
Lines IoU								
Masked loss	25.9	37.4	48.7	51.4	33.4	39.2	48.9	50.2
Full loss	26.2	37.6	48.5	50.8	32.7	39.1	48.9	50.0
Lines IoU, *reversed*								
Masked loss	25.9	37.3	48.4	51.2	33.2	39.1	48.5	50.5
Full loss	26.0	37.3	48.3	50.5	33.1	39.3	48.8	50.1
Lines IoU, *irrelevant*								
Masked loss	26.3	37.5	48.0	27.2	32.9	38.7	48.2	33.6
Full loss	26.1	37.2	48.4	26.8	33.2	38.9	48.5	33.5
Code Chunks								
Masked loss	25.7	37.5	48.3	48.1	33.0	38.9	48.3	48.8
Full loss	26.3	37.3	48.6	47.7	32.8	39.0	49.1	48.8
Code Chunks, *reversed*								
Masked loss	26.3	37.5	48.1	48.2	33.0	38.9	48.6	48.9
Full loss	26.3	37.6	48.9	47.6	33.1	39.2	48.9	48.7
Code Chunks, *irrelevant*								
Masked loss	25.7	37.6	47.8	27.6	32.3	38.4	47.6	33.4
Full loss	26.2	37.3	47.9	27.6	33.5	39.1	48.6	34.0
Random .py								
Masked loss	25.7	37.3	48.8	32.5	32.8	38.8	48.8	36.0
Full loss	26.0	37.4	48.9	32.4	33.0	39.2	49.1	36.0
Duplication								
Masked loss	19.2	28.4	34.8	95.3	24.2	26.2	28.1	94.2
Full loss	25.7	36.6	45.9	96.0	32.8	38.6	45.0	95.4

■ **Table A.3** Exact Match scores for repository-level pre-trained OpenCoder-1.5B-Base on composers generating inlier repository context for the completion file under two training setups: with and without gradient masking

Bibliography

Sapronov, Maksim; Glukhov, Evgeniy, 2025. On Pretraining For Project-Level Code Completion. In: *ICLR 2025 Third Workshop on Deep Learning for Code.* Available also from: `https://openreview.net/forum?id=t9RN9 WX4Ic`.

Aghajanyan, Armen; Huang, Bernie; Ross, Candace; Karpukhin, Vladimir; Xu, Hu; Goyal, Naman; Okhonko, Dmytro; Joshi, Mandar; Ghosh, Gargi; Lewis, Mike; Zettlemoyer, Luke, 2022. *CM3: A Causal Masked Multimodal Model of the Internet* [online] [visited on 2025-05-16]. Available from arXiv: `2201.07520 [cs.CL]`.

Allal, Loubna Ben; Li, Raymond; Kocetkov, Denis; Mou, Chenghao; Akiki, Christopher; Ferrandis, Carlos Munoz; Muennighoff, Niklas; Mishra, Mayank; Gu, Alex; Dey, Manan; Umapathi, Logesh Kumar; Anderson, Carolyn Jane; Zi, Yangtian; Poirier, Joel Lamy; Schoelkopf, Hailey; Troshin, Sergey; Abulkhanov, Dmitry; Romero, Manuel; Lappert, Michael; Toni, Francesco De; Río, Bernardo García del; Liu, Qian; Bose, Shamik; Bhattacharyya, Urvashi; Zhuo, Terry Yue; Yu, Ian; Villegas, Paulo; Zocca, Marco; Mangrulkar, Sourab; Lansky, David; Nguyen, Huu; Contractor, Danish; Villa, Luis; Li, Jia; Bahdanau, Dzmitry; Jernite, Yacine; Hughes, Sean; Fried, Daniel; Guha, Arjun; Vries, Harm de; Werra, Leandro von, 2023. *SantaCoder: don't reach for the stars!* [Online] [visited on 2025-05-16]. Available from arXiv: `2301.03988 [cs.SE]`.

Allamanis, Miltiadis; Sutton, Charles, 2013. Mining source code repositories at massive scale using language modeling. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 207–216. Available from DOI: `10.1109/MSR.2013.6624029`.

Ba, Jimmy Lei; Kiros, Jamie Ryan; Hinton, Geoffrey E., 2016. *Layer Normalization* [online] [visited on 2025-05-16]. Available from arXiv: `1607.06450` `[stat.ML]`.

Baevski, Alexei; Auli, Michael, 2018. *Adaptive Input Representations for Neural Language Modeling* [online] [visited on 2025-05-16]. Available from arXiv: `1809.10853` `[cs.CL]`.

Bahdanau, Dzmitry; Cho, Kyunghyun; Bengio, Yoshua, 2014. *Neural Machine Translation by Jointly Learning to Align and Translate* [online] [visited on 2025-05-16]. Available from arXiv: `1409.0473` `[cs.CL]`.

Bakal, Gal; Dasdan, Ali; Katz, Yaniv; Kaufman, Michael; Levin, Guy, 2025. *Experience with GitHub Copilot for Developer Productivity at Zoominfo* [online] [visited on 2025-05-16]. Available from arXiv: `2501.13282` `[cs.SE]`.

Banerjee, Satanjeev; Lavie, Alon, 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In: *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ann Arbor, Michigan: Association for Computational Linguistics, pp. 65–72. Available also from: `https://www.aclweb.org/anthology/W05-0909`.

Barbero, Federico; Vitvitskyi, Alex; Perivolaropoulos, Christos; Pascanu, Razvan; Veličković, Petar, 2024. *Round and Round We Go! What makes Rotary Positional Encodings useful?* [Online] [visited on 2025-05-16]. Available from arXiv: `2410.06205` `[cs.CL]`.

Bavarian, Mohammad; Jun, Heewoo; Tezak, Nikolas; Schulman, John; McLeavey, Christine; Tworek, Jerry; Chen, Mark, 2022. *Efficient Training of Language Models to Fill in the Middle* [online] [visited on 2025-05-16]. Available from arXiv: `2207.14255` `[cs.CL]`.

Bogomolov, Egor; Eliseeva, Aleksandra; Galimzyanov, Timur; Glukhov, Evgeniy; Shapkin, Anton; Tigina, Maria; Golubev, Yaroslav; Kovrigin, Alexander; Deursen, Arie van; Izadi, Maliheh; Bryksin, Timofey, 2024. *Long Code Arena: a Set of Benchmarks for Long-Context Code Models* [online] [visited on 2025-05-16]. Available from arXiv: `2406.11612` `[cs.LG]`.

Brown, Tom B.; Mann, Benjamin; Ryder, Nick; Subbiah, Melanie; Kaplan, Jared; Dhariwal, Prafulla; Neelakantan, Arvind; Shyam, Pranav; Sastry, Girish; Askell, Amanda; Agarwal, Sandhini; Herbert-Voss, Ariel; Krueger, Gretchen; Henighan, Tom; Child, Rewon; Ramesh, Aditya; Ziegler, Daniel M.; Wu, Jeffrey; Winter, Clemens; Hesse, Christopher; Chen, Mark; Sigler, Eric; Litwin, Mateusz; Gray, Scott; Chess, Benjamin; Clark, Jack; Berner, Christopher; McCandlish, Sam; Radford, Alec; Sutskever, Ilya; Amodei, Dario, 2020. *Language Models are Few-Shot Learners* [online] [visited on 2025-05-16]. Available from arXiv: `2005.14165` `[cs.CL]`.

Chen, Mark; Tworek, Jerry; Jun, Heewoo; Yuan, Qiming; Oliveira Pinto, Henrique Ponde de; Kaplan, Jared; Edwards, Harri; Burda, Yuri; Joseph, Nicholas; Brockman, Greg; Ray, Alex; Puri, Raul; Krueger, Gretchen; Petrov, Michael; Khlaaf, Heidy; Sastry, Girish; Mishkin, Pamela; Chan, Brooke; Gray, Scott; Ryder, Nick; Pavlov, Mikhail; Power, Alethea; Kaiser, Lukasz; Bavarian, Mohammad; Winter, Clemens; Tillet, Philippe; Such, Felipe Petroski; Cummings, Dave; Plappert, Matthias; Chantzis, Fotios; Barnes, Elizabeth; Herbert-Voss, Ariel; Guss, William Hebgen; Nichol, Alex; Paino, Alex; Tezak, Nikolas; Tang, Jie; Babuschkin, Igor; Balaji, Suchir; Jain, Shantanu; Saunders, William; Hesse, Christopher; Carr, Andrew N.; Leike, Jan; Achiam, Josh; Misra, Vedant; Morikawa, Evan; Radford, Alec; Knight, Matthew; Brundage, Miles; Murati, Mira; Mayer, Katie; Welinder, Peter; McGrew, Bob; Amodei, Dario; McCandlish, Sam; Sutskever, Ilya; Zaremba, Wojciech, 2021. *Evaluating Large Language Models Trained on Code* [online] [visited on 2025-05-16]. Available from arXiv: 2107.03374 `[cs.LG]`.

Chen, Shouyuan; Wong, Sherman; Chen, Liangjian; Tian, Yuandong, 2023. *Extending Context Window of Large Language Models via Positional Interpolation* [online] [visited on 2025-05-16]. Available from arXiv: 2306.15 595 `[cs.CL]`.

Child, Rewon; Gray, Scott; Radford, Alec; Sutskever, Ilya, 2019. *Generating Long Sequences with Sparse Transformers* [online] [visited on 2025-05-16]. Available from arXiv: 1904.10509 `[cs.LG]`.

Ciniselli, Matteo; Cooper, Nathan; Pascarella, Luca; Mastropaolo, Antonio; Aghajani, Emad; Poshyvanyk, Denys; Di Penta, Massimiliano; Bavota, Gabriele, 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering*, pp. 1–1. ISSN 2326-3881. Available from DOI: 10.1109/tse.2021.3128234.

Deng, Ken; Liu, Jiaheng; Zhu, He; Liu, Congnan; Li, Jingxin; Wang, Jiakai; Zhao, Peng; Zhang, Chenchen; Wu, Yanan; Yin, Xueqiao; Zhang, Yuanxing; Su, Wenbo; Xiang, Bangyu; Ge, Tiezheng; Zheng, Bo, 2024. *R^2C^2-Coder: Enhancing and Benchmarking Real-world Repository-level Code Completion Abilities of Code Large Language Models* [online] [visited on 2025-05-16]. Available from arXiv: 2406.01359 `[cs.CL]`.

Dibia, Victor; Fourney, Adam; Bansal, Gagan; Poursabzi-Sangdeh, Forough; Liu, Han; Amershi, Saleema, 2022. *Aligning Offline Metrics and Human Judgments of Value for Code Generation Models* [online] [visited on 2025-05-16]. Available from arXiv: 2210.16494 `[cs.SE]`.

Ding, Ming; Yang, Zhuoyi; Hong, Wenyi; Zheng, Wendi; Zhou, Chang; Yin, Da; Lin, Junyang; Zou, Xu; Shao, Zhou; Yang, Hongxia; Tang, Jie, 2021.

CogView: Mastering Text-to-Image Generation via Transformers [online] [visited on 2025-05-16]. Available from arXiv: `2105.13290 [cs.CV]`.

Ding, Yangruibo; Wang, Zijian; Ahmad, Wasi Uddin; Ding, Hantian; Tan, Ming; Jain, Nihal; Ramanathan, Murali Krishna; Nallapati, Ramesh; Bhatia, Parminder; Roth, Dan; Xiang, Bing, 2023. *CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion* [online] [visited on 2025-05-16]. Available from arXiv: `2310.11248 [cs.LG]`.

Ding, Yangruibo; Wang, Zijian; Ahmad, Wasi Uddin; Ramanathan, Murali Krishna; Nallapati, Ramesh; Bhatia, Parminder; Roth, Dan; Xiang, Bing, 2022. *CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context* [online] [visited on 2025-05-16]. Available from arXiv: `2212.10007 [cs.CL]`.

Donahue, Chris; Lee, Mina; Liang, Percy, 2020. *Enabling Language Models to Fill in the Blanks* [online] [visited on 2025-05-16]. Available from arXiv: `2005.05339 [cs.CL]`.

Estoup, J. B., 1916. Les Gammes Stenographiques. *Institut Stenographique de France.*

Evtikhiev, Mikhail; Bogomolov, Egor; Sokolov, Yaroslav; Bryksin, Timofey, 2022. Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software.* Vol. 203, p. 111741. ISSN 0164-1212. Available from DOI: `10.1016/j.jss.2023.111741`.

Fried, Daniel; Aghajanyan, Armen; Lin, Jessy; Wang, Sida; Wallace, Eric; Shi, Freda; Zhong, Ruiqi; Yih, Wen-tau; Zettlemoyer, Luke; Lewis, Mike, 2022. *InCoder: A Generative Model for Code Infilling and Synthesis* [online] [visited on 2025-05-16]. Available from arXiv: `2204.05999 [cs.SE]`.

Gage, Philip, 1994. A new algorithm for data compression. *The C Users Journal archive.* Vol. 12, pp. 23–38. Available also from: `https://api.semanticscholar.org/CorpusID:59804030`.

Giagnorio, Alessandro; Martin-Lopez, Alberto; Bavota, Gabriele, 2025. *Why Personalizing Deep Learning-Based Code Completion Tools Matters* [online] [visited on 2025-05-16]. Available from arXiv: `2503.14201 [cs.SE]`.

Guo, Daya; Zhu, Qihao; Yang, Dejian; Xie, Zhenda; Dong, Kai; Zhang, Wentao; Chen, Guanting; Bi, Xiao; Wu, Y.; Li, Y. K.; Luo, Fuli; Xiong, Yingfei; Liang, Wenfeng, 2024. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence* [online] [visited on 2025-05-16]. Available from arXiv: `2401.14196 [cs.SE]`.

Hahn, Michael; Goyal, Navin, 2023. *A Theory of Emergent In-Context Learning as Implicit Structure Induction* [online] [visited on 2025-05-16]. Available from arXiv: `2303.07971 [cs.CL]`.

Harris, Zellig S., 1954. Distributional Structure. In: available also from: `https://api.semanticscholar.org/CorpusID:86680084`.

Hendrycks, Dan; Gimpel, Kevin, 2016. *Gaussian Error Linear Units (GELUs)* [online] [visited on 2025-05-16]. Available from arXiv: `1606.08415` `[cs.LG]`.

Holtzman, Ari; Buys, Jan; Du, Li; Forbes, Maxwell; Choi, Yejin, 2019. *The Curious Case of Neural Text Degeneration* [online] [visited on 2025-05-16]. Available from arXiv: `1904.09751` `[cs.CL]`.

Huang, Siming; Cheng, Tianhao; Liu, J. K.; Hao, Jiaran; Song, Liuyihan; Xu, Yang; Yang, J.; Liu, Jiaheng; Zhang, Chenchen; Chai, Linzheng; Yuan, Ruifeng; Zhang, Zhaoxiang; Fu, Jie; Liu, Qian; Zhang, Ge; Wang, Zili; Qi, Yuan; Xu, Yinghui; Chu, Wei, 2024. *OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models* [online] [visited on 2025-05-16]. Available from arXiv: `2411.04905` `[cs.CL]`.

Hui, Binyuan; Yang, Jian; Cui, Zeyu; Yang, Jiaxi; Liu, Dayiheng; Zhang, Lei; Liu, Tianyu; Zhang, Jiajun; Yu, Bowen; Lu, Keming; Dang, Kai; Fan, Yang; Zhang, Yichang; Yang, An; Men, Rui; Huang, Fei; Zheng, Bo; Miao, Yibo; Quan, Shanghaoran; Feng, Yunlong; Ren, Xingzhang; Ren, Xuancheng; Zhou, Jingren; Lin, Junyang, 2024. *Qwen2.5-Coder Technical Report* [online] [visited on 2025-05-16]. Available from arXiv: `2409.12186` `[cs.CL]`.

Kaplan, Jared; McCandlish, Sam; Henighan, Tom; Brown, Tom B.; Chess, Benjamin; Child, Rewon; Gray, Scott; Radford, Alec; Wu, Jeffrey; Amodei, Dario, 2020. *Scaling Laws for Neural Language Models* [online] [visited on 2025-05-16]. Available from arXiv: `2001.08361` `[cs.LG]`.

Kingma, Diederik P.; Ba, Jimmy, 2014. *Adam: A Method for Stochastic Optimization* [online] [visited on 2025-05-16]. Available from arXiv: `1412.6980` `[cs.LG]`.

Labach, Alex; Salehinejad, Hojjat; Valaee, Shahrokh, 2019. *Survey of Dropout Methods for Deep Neural Networks* [online] [visited on 2025-05-16]. Available from arXiv: `1904.13310` `[cs.NE]`.

Lewis, Patrick; Perez, Ethan; Piktus, Aleksandra; Petroni, Fabio; Karpukhin, Vladimir; Goyal, Naman; Küttler, Heinrich; Lewis, Mike; Yih, Wen-tau; Rocktäschel, Tim; Riedel, Sebastian; Kiela, Douwe, 2021. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* [online] [visited on 2025-05-16]. Available from arXiv: `2005.11401` `[cs.CL]`.

Lin, Chin-Yew, 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In: *Text Summarization Branches Out.* Barcelona, Spain: Association for Computational Linguistics, pp. 74–81. Available also from: `https://aclanthology.org/W04-1013/`.

Liu, Jiachang; Shen, Dinghan; Zhang, Yizhe; Dolan, Bill; Carin, Lawrence; Chen, Weizhu, 2021. *What Makes Good In-Context Examples for GPT-3?* [Online] [visited on 2025-05-16]. Available from arXiv: `2101.06804` `[cs.CL]`.

Liu, Jiaheng; Deng, Ken; Liu, Congnan; Yang, Jian; Liu, Shukai; Zhu, He; Zhao, Peng; Chai, Linzheng; Wu, Yanan; Jin, Ke; Zhang, Ge; Wang, Zekun; Zhang, Guoan; Xiang, Bangyu; Su, Wenbo; Zheng, Bo, 2024. *M²RC-EVAL: Massively Multilingual Repository-level Code Completion Evaluation* [online] [visited on 2025-05-16]. Available from arXiv: `2410.21157 [cs.CL]`.

Liu, Tianyang; Xu, Canwen; McAuley, Julian, 2023. *RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems* [online] [visited on 2025-05-16]. Available from arXiv: `2306.03091 [cs.CL]`.

Loshchilov, Ilya; Hutter, Frank, 2017. *Decoupled Weight Decay Regularization* [online] [visited on 2025-05-16]. Available from arXiv: `1711.05101 [cs.LG]`.

Lu, Shuai; Guo, Daya; Ren, Shuo; Huang, Junjie; Svyatkovskiy, Alexey; Blanco, Ambrosio; Clement, Colin; Drain, Dawn; Jiang, Daxin; Tang, Duyu; Li, Ge; Zhou, Lidong; Shou, Linjun; Zhou, Long; Tufano, Michele; Gong, Ming; Zhou, Ming; Duan, Nan; Sundaresan, Neel; Deng, Shao Kun; Fu, Shengyu; Liu, Shujie, 2021. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation* [online] [visited on 2025-05-16]. Available from arXiv: `2102.04664 [cs.SE]`.

Murphy, Kevin P., 2022. *Probabilistic Machine Learning: An introduction.* MIT Press. Available also from: `http://probml.github.io/book1`.

NVIDIA Corporation, 2023. *Mixed Precision Training* [online] [visited on 2025-05-13]. Available from: `https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html`.

Pan, Zhenyu; Cao, Rongyu; Cao, Yongchang; Ma, Yingwei; Li, Binhua; Huang, Fei; Liu, Han; Li, Yongbin, 2024. *Codev-Bench: How Do LLMs Understand Developer-Centric Code Completion?* [Online] [visited on 2025-05-16]. Available from arXiv: `2410.01353 [cs.SE]`.

Papineni, Kishore; Roukos, Salim; Ward, Todd; Zhu, Wei-Jing, 2002. BLEU: a method for automatic evaluation of machine translation. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics.* Philadelphia, Pennsylvania: Association for Computational Linguistics, pp. 311–318. ACL '02. Available from DOI: `10.3115/1073083.1073135`.

Pascanu, Razvan; Mikolov, Tomas; Bengio, Yoshua, 2013. *On the difficulty of training Recurrent Neural Networks* [online] [visited on 2025-05-16]. Available from arXiv: `1211.5063 [cs.LG]`.

Peng, Sida; Kalliamvakou, Eirini; Cihon, Peter; Demirer, Mert, 2023. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot* [online] [visited on 2025-05-16]. Available from arXiv: `2302.06590 [cs.SE]`.

Popović, Maja, 2015. chrF: character n-gram F-score for automatic MT evaluation. In: Bojar, Ondřej; Chatterjee, Rajan; Federmann, Christian; Haddow, Barry; Hokamp, Chris; Huck, Matthias; Logacheva, Varvara; Pecina, Pavel (eds.). *Proceedings of the Tenth Workshop on Statistical Machine Translation.* Lisbon, Portugal: Association for Computational Linguistics, pp. 392–395. Available from DOI: `10.18653/v1/W15-3049`.

Raychev, Veselin; Bielik, Pavol; Vechev, Martin, 2016. Probabilistic model for code with decision trees. *SIGPLAN Not.* Vol. 51, no. 10, pp. 731–747. ISSN 0362-1340. Available from DOI: `10.1145/3022671.2984041`.

Ren, Shuo; Guo, Daya; Lu, Shuai; Zhou, Long; Liu, Shujie; Tang, Duyu; Sundaresan, Neel; Zhou, Ming; Blanco, Ambrosio; Ma, Shuai, 2020. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis* [online] [visited on 2025-05-16]. Available from arXiv: `2009.10297 [cs.SE]`.

Robertson, Stephen; Zaragoza, Hugo, 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* Vol. 3, no. 4, pp. 333–389. ISSN 1554-0669. Available from DOI: `10.1561/1500000019`.

Rozière, Baptiste; Gehring, Jonas; Gloeckle, Fabian; Sootla, Sten; Gat, Itai; Tan, Xiaoqing Ellen; Adi, Yossi; Liu, Jingyu; Sauvestre, Romain; Remez, Tal; Rapin, Jérémy; Kozhevnikov, Artyom; Evtimov, Ivan; Bitton, Joanna; Bhatt, Manish; Ferrer, Cristian Canton; Grattafiori, Aaron; Xiong, Wenhan; Défossez, Alexandre; Copet, Jade; Azhar, Faisal; Touvron, Hugo; Martin, Louis; Usunier, Nicolas; Scialom, Thomas; Synnaeve, Gabriel, 2023. *Code Llama: Open Foundation Models for Code* [online] [visited on 2025-05-16]. Available from arXiv: `2308.12950 [cs.CL]`.

Sennrich, Rico; Haddow, Barry; Birch, Alexandra, 2015. *Neural Machine Translation of Rare Words with Subword Units* [online] [visited on 2025-05-16]. Available from arXiv: `1508.07909 [cs.CL]`.

Shazeer, Noam, 2020. *GLU Variants Improve Transformer* [online] [visited on 2025-05-16]. Available from arXiv: `2002.05202 [cs.LG]`.

Srivastava, Nitish; Hinton, Geoffrey; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan, 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research.* Vol. 15, no. 56, pp. 1929–1958. Available also from: `http://jmlr.org/papers/v15/srivastava14a.html`.

Su, Jianlin; Lu, Yu; Pan, Shengfeng; Murtadha, Ahmed; Wen, Bo; Liu, Yunfeng, 2021. *RoFormer: Enhanced Transformer with Rotary Position Embedding* [online] [visited on 2025-05-16]. Available from arXiv: `2104.09864 [cs.CL]`.

Svyatkovskiy, Alexey; Deng, Shao Kun; Fu, Shengyu; Sundaresan, Neel, 2020. *IntelliCode Compose: Code Generation Using Transformer* [online] [visited on 2025-05-16]. Available from arXiv: `2005.08025 [cs.CL]`.

Takerngsaksiri, Wannita; Warusavitarne, Cleshan; Yaacoub, Christian; Hou, Matthew Hee Keng; Tantithamthavorn, Chakkrit, 2023. *Students' Perspective on AI Code Completion: Benefits and Challenges* [online] [visited on 2025-05-16]. Available from arXiv: 2311.00177 [cs.SE].

Tay, Yi; Dehghani, Mostafa; Bahri, Dara; Metzler, Donald, 2022. *Efficient Transformers: A Survey* [online] [visited on 2025-05-16]. Available from arXiv: 2009.06732 [cs.LG].

Tran, Ngoc; Tran, Hieu; Nguyen, Son; Nguyen, Hoan; Nguyen, Tien, 2019. Does BLEU Score Work for Code Migration? In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, pp. 165–176. Available from DOI: 10.1109/icpc.2019.00034.

Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N.; Kaiser, Lukasz; Polosukhin, Illia, 2017. *Attention Is All You Need* [online] [visited on 2025-05-16]. Available from arXiv: 1706.037 62 [cs.CL].

Wang, Qiang; Li, Bei; Xiao, Tong; Zhu, Jingbo; Li, Changliang; Wong, Derek F.; Chao, Lidia S., 2019. *Learning Deep Transformer Models for Machine Translation* [online] [visited on 2025-05-16]. Available from arXiv: 1906.0 1787 [cs.CL].

Weber, Thomas; Brandmaier, Maximilian; Schmidt, Albrecht; Mayer, Sven, 2024. Significant Productivity Gains through Programming with Large Language Models. *Proc. ACM Hum.-Comput. Interact.* Vol. 8, no. EICS. Available from DOI: 10.1145/3661145.

Wu, Di; Ahmad, Wasi Uddin; Zhang, Dejiao; Ramanathan, Murali Krishna; Ma, Xiaofei, 2024a. *Repoformer: Selective Retrieval for Repository-Level Code Completion* [online] [visited on 2025-05-16]. Available from arXiv: 2403.10059 [cs.SE].

Wu, Qinyun; Peng, Chao; Gao, Pengfei; Hu, Ruida; Gan, Haoyu; Jiang, Bo; Tang, Jinhe; Deng, Zhiwen; Guan, Zhanming; Gao, Cuiyun; Liu, Xia; Yang, Ping, 2024b. *RepoMasterEval: Evaluating Code Completion via Real-World Repositories* [online] [visited on 2025-05-16]. Available from arXiv: 2408.0 3519 [cs.SE].

Xiong, Ruibin; Yang, Yunchang; He, Di; Zheng, Kai; Zheng, Shuxin; Xing, Chen; Zhang, Huishuai; Lan, Yanyan; Wang, Liwei; Liu, Tie-Yan, 2020. *On Layer Normalization in the Transformer Architecture* [online] [visited on 2025-05-16]. Available from arXiv: 2002.04745 [cs.LG].

Xiong, Wenhan; Liu, Jingyu; Molybog, Igor; Zhang, Hejia; Bhargava, Prajjwal; Hou, Rui; Martin, Louis; Rungta, Rashi; Sankararaman, Karthik Abinav; Oguz, Barlas; Khabsa, Madian; Fang, Han; Mehdad, Yashar; Narang, Sharan; Malik, Kshitiz; Fan, Angela; Bhosale, Shruti; Edunov, Sergey; Lewis, Mike; Wang, Sinong; Ma, Hao, 2023. *Effective Long-Context Scaling of*

Foundation Models [online] [visited on 2025-05-16]. Available from arXiv: `2309.16039` [`cs.CL`].

Zehui, Lin; Liu, Pengfei; Huang, Luyao; Chen, Junkun; Qiu, Xipeng; Huang, Xuanjing, 2019. *DropAttention: A Regularization Method for Fully-Connected Self-Attention Networks* [online] [visited on 2025-05-16]. Available from arXiv: `1907.11065` [`cs.CL`].

Zhang, Aston; Lipton, Zachary C.; Li, Mu; Smola, Alexander J., 2023a. *Dive into Deep Learning* [online] [visited on 2025-05-16]. Available from arXiv: `2106.11342` [`cs.LG`].

Zhang, Biao; Sennrich, Rico, 2019. *Root Mean Square Layer Normalization* [online] [visited on 2025-05-16]. Available from arXiv: `1910.07467` [`cs.LG`].

Zhang, Fengji; Chen, Bei; Zhang, Yue; Keung, Jacky; Liu, Jin; Zan, Daoguang; Mao, Yi; Lou, Jian-Guang; Chen, Weizhu, 2023b. *RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation* [online] [visited on 2025-05-16]. Available from arXiv: `2303.12570` [`cs.CL`].

Contents of the attachment