Advancing Large Language Models for Code Using Code-Structure-Aware Methods

Linyuan Gong
Alvin Cheung, Ed.
Dawn Song, Ed.
Sida Wang, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-50
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-50.html

May 13, 2025

Advancing Large Language Models for Code Using Code-Structure-Aware Methods

By

Linyuan Gong

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Alvin Cheung, Chair
Professor Xiaodong Song
Doctor Sida Wang

Spring 2025

Advancing Large Language Models for Code Using Code-Structure-Aware Methods

Abstract

Advancing Large Language Models for Code Using Code-Structure-Aware Methods

by

Linyuan Gong

Doctor of Philosophy in Computer Science

University of California, Berkeley

Associate Professor Alvin Cheung, Chair

Large language models (LLMs) have transformed code-related tasks. However, most code LLMs ignore structural patterns of programming languages. This dissertation studies code-structure-aware LLMs by proposing novel methodologies, benchmarks, and pretraining strategies, showing that explicit structural modeling significantly enhances coding capability of LLMs.

First, we introduce ADELT, a transpiler that decouples code structure conversion from API keyword translation. ADELT achieves state-of-the-art transpilation without parallel data, showing the importance of structural awareness.

To rigorously evaluate structural understanding, we present SAFIM, a benchmark for syntax-aware FIM tasks. Evaluating 15 LLMs, we challenge the idea that "big model = good performance", and show that pretraining strategies and data quality are more important. We establish SAFIM as a foundational tool for future research.

We then propose two structure-aware pretraining paradigms. AST-T5 integrates abstract syntax trees (ASTs) into T5-like encoder-decoder models, outperforming baselines in code repair and transpilation. For decoder-only architectures, AST-FIM uses AST-guided masking to better address the tradeoff between FIM and left-to-right (L2R) generation, surpassing traditional methods on infilling tasks while retaining L2R generation capability.

Collectively, we show that code structure awareness enhances code generation, understanding, and transformation ability of LLMs. Our contributions—spanning transpilation frameworks, evaluation benchmarks, and pretraining techniques—provide a roadmap for integrating code structures into LLMs.

To my wife, Weiminghui Ji.
To my mother, Xianghong Lin.
In ever-present memory, my father, Xuechao Gong.

Contents

List of Figures

List of Tables

Acknowledgments

It is with immense gratitude that I acknowledge people who have supported me throughout my doctoral journey.

First and foremost, I extend my deepest gratitude to my PhD advisor, Professor Alvin Cheung. Alvin granted me the freedom to explore my research interests while providing invaluable guidance on research direction, paper writing, and academic presentations. His patient and unwavering support, especially during hard times, played a crucial role in my growth and progress.

I am also profoundly grateful to Professor Dawn Song, who served as my co-advisor during my first year and as a member of my dissertation committee. Professor Song's unique vision and insightful perspectives on research problems often provided me with a fresh lens through which to consider my projects.

My sincere thanks go to Sida Wang and Professor Koushik Sen for serving on my dissertation and qualifying exam committees. Their valuable feedback has significantly shaped my research and this dissertation. I am particularly grateful to Sida Wang, who made the AST-FIM project possible and offered very hands-on guidance in both AST-FIM and SAFIM.

I was fortunate to collaborate with several talented individuals. I thank Mostafa Elhoushi for his contributions to the AST-T5, SAFIM, and AST-FIM projects; Jiayi Wang for our work on ADELT; and Xinyun Chen for our work on PlotCoder.

In 2022, I had the privilege of being a summer research intern at Microsoft Research. I am grateful to my intern hosts, Professor Chenyan Xiong and Xiaodong Liu, for their guidance on our research project. I also thank my collaborators: Payal Bajaj, Yiqing Xie, Jianfeng Gao, Xia Song, and Carlton Shen.

Throughout my PhD studies, I have been fortunate to learn from and alongside many bright peers. I would like to thank Zhuohan Li, Xiaoxuan (Lily) Liu, Sahil Bhatia, and Zheng Liang for the enriching discussions and support.

My research journey began before Berkeley, and I am thankful for the brilliant advisors and collaborators who guided my initial steps. My heartfelt appreciation goes to my undergraduate advisors at Peking University, Professor Liwei Wang and Professor Di He. I also thank Guolin Ke, Tao Qin, Bian Jiang, Professor Tieyan Liu, Professor Pradeep Ravikumar, and Professor Jeremy C. Weiss for their mentorship. My early collaborations with Zhenhui Xu, Shuxin Zheng, Juyong Kim, Yilun Xu, and Ruyi Ji were also valuable experiences.

Finally, I owe my deepest gratitude to my family. To my mother, Xianghong Lin, for

her unconditional love and unwavering belief in me. To my father, Xuechao Gong, who passed away a few years ago; his memory continues to inspire me. I also thank my grandparents, aunt, uncle, and cousins for their support of our family during the difficult time of my father's passing. Last but certainly not least, to my wonderful wife, Weiminghui Ji, whom I married during my second year of PhD studies: your love, happiness, and inspiration have been my bedrock.

Chapter 1

Introduction

1.1 Background

Traditional neural program synthesis, before rise of large language models (LLMs), typically involves a multi-step process. First, a domain-specific language (DSL) tailored to the specific task is defined. Second, a neural network is trained to synthesize code within this DSL. Third, the DSL-generated code is translated into executable code. In this paradigm, the neural network's architecture is intrinsically linked to the DSL. For example, DeepCoder [5] uses the DSL to encode the current program state, with the neural network selecting valid actions within the DSL's defined state space. Other approaches, such as PlotCoder [6], does not explicitly define a DSL but still uses a vocabulary specialized for the target task in their neural network. While effective for narrow domains, these approaches were limited by their reliance on manually designed DSLs and task-specific adaptations.

The advent of LLMs has transformed the landscape of neural program synthesis. LLMs are pretrained on vast datasets comprising both natural language and code (e.g. the entire Wikipedia and all public repositories in GitHub). Generative pretraining equips LLMs with ability to directly generate code, without using DSL intermediaries. By unifying code and text into token sequences, LLMs achieve remarkable zero-shot generalization: given human instructions, LLMs can adeptly perform previously unseen tasks.

However, existing LLMs treat code as plain text, ignoring its inherent structured nature. This gap motivates the central question of this dissertation: How can we equip LLMs to understand and leverage code structure to advance program synthesis?

1.2 Definition of Code Structures

Code structure is the formal relationships between code elements, usually represented by trees or graphs. The main form of code structure considered in this dissertation is the *syntactic structure*, represented by Abstract Syntax Trees (ASTs) of code. An AST delineates a hierarchical arrangement of syntax nodes. For example, a function definition typically contains statements; a statement contains expressions; an expression contains identifiers or literals.

Beyond ASTs, other code-related structures can also be useful. These include control-flow graphs (CFGs), which map the potential execution paths within a program. Furthermore, structures spanning multiple files, such as file paths or inter-file dependencies, can be relevant. However, we mainly focus on syntactical structures represented by ASTs, because ASTs can be efficiently generated by a simple static parser—no execution or tracing is required.

1.3 The Need for Structure Awareness

While LLMs have shown impressive capabilities in generating code for simple tasks, such as HumanEval and MBPP which often involve short, self-contained Python functions, the performance of LLMs tends to degrade when faced with more complex real-world coding scenarios. This dissertation argues that incorporating an understanding of code structure is crucial for overcoming these limitations.

Several observations highlight the shortcomings of current LLMs that treat code merely as sequences of tokens. Firstly, they often struggle with low-resource programming languages (e.g. COBOL [7] and Verilog [8]), where limited training data hinders their ability to learn syntax and semantics. Secondly, they often struggle with tasks involve less popular or highly specialized API functions, which are underrepresented in pretraining corpora. Thirdly, they often struggle when handling long contexts, especially those spanning multiple files and requiring an understanding of inter-dependencies.

The potential benefits of structure awareness are supported by existing research. Studies have shown that models explicitly designed to leverage code structure can achieve better results on general-purpose coding tasks. For example, models like GraphCodeBERT [9] and StructCoder [10] have shown improved performance by incorporating structural information. Moreover, in domain-specific applications, such as hardware design, structure-aware approaches like VerilogCoder [11] have shown notable improvements. Such successes underscore the value of structural priors in

compensating for data scarcity and complexity.

In Chapter 2, we illustrate this principle through ADELT, a transpiler designed to convert code between deep learning frameworks. Standard LLMs struggle with generating correct accurate framework-specific APIs. ADELT addresses this by decoupling the problem, using two specialized models: one for transpiling the code skeleton and another for mapping API keywords. Crucially, its Abstract Syntax Tree (AST)-aware decoupling process allows these models to specialize on different structural code components without mutual interference. This AST-aware strategy significantly improves transpilation pass@1 rates over standard LLMs, showing how structure-awareness can bridge critical gaps in their capabilities.

In summary, current LLMs often struggle in complex or low-resource coding scenarios. The evidence from related work, and as will be further explored through examples like ADELT in this dissertation, suggests that equipping LLMs with an understanding of code structure is a promising solution.

1.4 Evaluation of Structure Awareness

Having established the importance of structure awareness for LLMs in program synthesis, an important question arises: How can we effectively evaluate the extent to which these models understand and leverage code structure? The answer differs for encoder-only and decoder-only model architectures.

One prominent approach for evaluating structure awareness in encoder-only models is through *attention probing*. For example, CodeSyntax [12] parses code into ASTs. Based on these ASTs, they define significant token relationships, such as the link between an if keyword and its associated else keyword. By analyzing attention maps, structural awareness is quantified. This technique reveals that models like GraphCodeBERT [9], which are explicitly designed to incorporate structural information, exhibit greater structure awareness compared to standard models like CodeBERT [13] that treat code as a flat sequence of tokens.

For decoder-only generative models, which directly synthesize code, a more direct evaluation of structure awareness is feasible through their generation performance on specifically designed tasks. Our work SAFIM (Chapter 3) introduces a syntax-aware fill-in-the-middle (FIM) benchmark, testing LLMs' ability to complete code blocks, conditional expressions, and other structured elements. Evaluations of 15 LLMs show that pretraining strategies and data quality outweigh model size in improving structural awareness. By providing a standardized evaluation platform, SAFIM not only measures progress but also guides future research toward more effective training

paradigms for code LLMs.

By using such targeted evaluations, we can gain deeper insights into how well LLMs understand code structure and identify areas for improvement. These evaluation methods are crucial for validating the effectiveness of the structure-aware training techniques proposed later in this dissertation.

1.5 Training Structure-Aware LLMs

Given the established need for structure awareness and methods to evaluate it, the subsequent challenge is to effectively train LLMs to be structure-aware. Our objective is to develop general-purpose, structure-aware LLMs, rather than models specialized for narrow tasks.

Previous research has explored integrating structural information into LLMs. Models like GraphCodeBERT [9] and StructCoder [10] show that incorporating code structure can lead to improved performance on various downstream tasks, sometimes rivaling or even surpassing larger models that treat code as plain text. However, these pioneering models have limitations in practical applications: users tend to use conventional LLMs that treat code as plain text. This preference is because models such as GraphCodeBERT and StructCoder typically gain structure awareness by directly encoding structural information into the model's input. Consequently, they require access to fully correct code structures during inference. This dependency becomes problematic as real-world user inputs often consist of incomplete or syntactically incorrect code snippets, which cannot be readily fed into parsers or code analysis tools.

Motivated by these limitations, this dissertation focuses on developing structure-aware LLMs that need no explicit structural information during inference. We ensure that our LLMs can serve as drop-in replacements for existing LLMs, seamlessly integrating into existing workflows without issues with malformed or partial code inputs. Simultaneously, this paradigm allows us to efficiently leverage code structure during the training phase at scale.

To explore this, we focus on two neural network architectures: T5-style encoder-decoder models and GPT-style decoder-only models.

Encoder-decoder models, often effective at smaller scales without requiring billions of parameters, provide an excellent initial proving ground for structure-aware techniques. Existing models like CodeT5 [14] have shown that T5-like models can perform a variety of tasks including code generation, transpilation, and code understanding

(e.g., classification), with particular strengths in code-to-code translation tasks. Our work aims to enhance such models with structural awareness. We introduce AST-T5 (Chapter 4), a novel pretraining paradigm designed to leverage ASTs to improve code generation, transpilation, and understanding. AST-T5 uses AST-Aware Segmentation to retain code structure in each context window, and an AST-Aware Span Corruption objective that trains the model to reconstruct various code structures. Crucially, AST-T5 avoids complex program analyses or architectural modifications, allowing it to integrate seamlessly with any systems that needs a standard encoder-decoder Transformer. Evaluations show that AST-T5 consistently outperforms similar-sized language models across diverse code-related tasks, including HumanEval and MBPP. Its structure-awareness makes AST-T5 particularly potent in code-to-code tasks, like automatic bug fixing and code transpilation.

Following the promising results with encoder-decoder architectures, we then address the question of whether AST-aware methods can effectively scale to larger, decoder-only models. This presents distinct challenges, as decoder-only models are trained autoregressively, typically optimizing a loss function over every token in a sequence. This differs from the span corruption objectives often used in encoder-decoder pre-training where only the masked part participate in loss calculation. Specifically, we study whether AST-aware masking strategies remain beneficial in the context of Fill-in-the-Middle (FIM) pretraining for decoder-only models.

Our work on AST-FIM, presented in Chapter 5, tackles this question. AST-FIM is a pretraining strategy that masks complete syntactic structures during FIM pretraining. Unlike random character FIM (Rand-FIM), AST-FIM ensures that masked spans align with AST nodes, creating coherent infilling targets. AST-FIM improves infilling performance on SAFIM and real-world code completion benchmarks significantly over standard FIM, with 1B and 8B models showing consistent gains. Also, AST-FIM provides less noisy training signals than Rand-FIM, so it offers similar performance as L2R models on standard L2R code generation tasks, while Rand-FIM models suffer from decreased performance.

1.6 Conclusion and Dissertation Outline

In summary, while LLMs have significantly advanced neural program synthesis by directly generating code, their common approach of treating code as plain text ignores the structural information in code. This dissertation points out that equipping LLMs with an understanding of code structure is important for overcoming their current limitations in handling complex, low-resource, or specialized coding tasks.

The subsequent chapters will delve into these contributions in detail. Chapter 2 will further illustrate the benefits of structure-awareness through the ADELT case study. Chapter 3 will present the SAFIM benchmark for evaluating structural understanding in generative models. Chapter 4 will introduce the AST-T5 pretraining methodology for encoder-decoder architectures, and Chapter 5 will detail the AST-FIM approach for large-scale decoder-only models. Through these explorations, this dissertation seeks to show that by systematically incorporating code structure, we can significantly enhance the capabilities and reliability of LLMs for a wide range of program synthesis challenges.

Chapter 2

ADELT: Transpilation Between Deep Learning Frameworks

We propose the **Ad**versarial **DE**ep **L**earning **T**ranspiler (**ADELT**), a novel approach to source-to-source transpilation between deep learning frameworks. ADELT uniquely decouples code skeleton transpilation and API keyword mapping. For code skeleton transpilation, it uses few-shot prompting on large language models (LLMs), while for API keyword mapping, it uses contextual embeddings from a code-specific BERT. These embeddings are trained in a domain-adversarial setup to generate a keyword translation dictionary. ADELT is trained on an unlabeled web-crawled deep learning corpus, without relying on any hand-crafted rules or parallel data. It outperforms state-of-the-art transpilers, improving pass@1 rate by 16.2 pts and 15.0 pts for PyTorch-Keras and PyTorch-MXNet transpilation pairs respectively. We provide open access to our code at `https://github.com/gonglinyuan/adelt`.

2.1 Introduction

The rapid development of deep learning (DL) has led to an equally fast emergence of new software frameworks for training neural networks. Unfortunately, maintaining a deep learning framework and keeping it up-to-date is not an easy task. Many deep learning frameworks are deprecated or lose popularity every year, and porting deep learning code from a legacy framework to a new one is a tedious and error-prone task. A *source-to-source transpiler between DL frameworks* would greatly help practitioners overcome this difficulty.

Two promising solutions to source-to-source transpilation between deep learning frameworks are unsupervised neural machine translation (NMT) [15] and large lan-

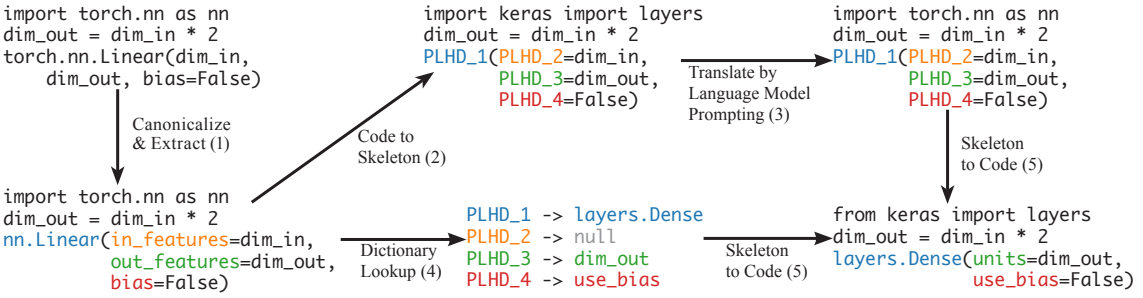


Figure 2.1: **An example of ADELT's pipeline:** an import statement in the code skeleton is transpiled from PyTorch to Keras by a language model via few-shot prompting; a linear fully-connected layer is transpiled by removing the argument `in_features` and renaming other API keywords according to the learned dictionary. The number (1 to 5) near each arrow label corresponds to the step number in Section 2.2.

guage models (LLMs). NMT treats deep learning code as a sentence for training sequence-to-sequence [16] models, but its applicability is limited due to the scarcity of parallel corpora and its notable data hunger. On the other hand, LLMs like GPT-3 [17], pretrained on web crawl data, offer potential, performing translation tasks in a few-shot or zero-shot manner. Our early experiments with GPT-4 show its potential in few-shot transpilation of deep learning programs. However, such models struggle with API-specific details, inaccurately handling function names and parameter mappings.

That said, most deep learning framework code is structured: each type of layers has its own constructor, and constructing a network involves calling each layer's constructor in a chaining manner. By leveraging the structures of programming languages, we can *decouple* the transpilation of skeletal codes from the mapping of API keywords. The transpilation of skeletal codes is the easier part, and LLMs already do a great job. We only need a separate algorithm to translate the *API keywords*, i.e., the function and parameter names to complete the transpilation.

In this chapter, we present ADELT (Figure 2.1), a method motivated by this insight to transpile DL code. The canonicalized source code is decoupled into two parts: the code skeleton and the API keywords. ADELT transpiles the code skeleton using a pretrained LLM by few-shot prompting. Each API keyword occurrence is then embedded into a vector by PyBERT, a BERT pretrained on Python code. This vector is both the textual and the contextual representation of the API keyword. ADELT

then leverages domain-adversarial training to learn a generator that maps the vector to an aligned embedding space. The alignment is enforced by a two-player game, where a discriminator is trained to distinguish between the embeddings from the source DL framework and those from the target DL framework. The API keyword embeddings are trained jointly with the generator as the output embedding matrix of a softmax classifier on the aligned embedding space. After generating a synthetic API keyword dictionary from the embeddings using a two-step greedy algorithm, ADELT then looks up each API keyword occurrence in the dictionary and puts them back into the transpiled code skeleton.

In summary, this chapter makes the following contributions:

- We introduce ADELT, a robust solution for transpilation between deep learning frameworks without training on any labeled data. Outperforming large language models, ADELT excels across various transpilation pairs, achieving pass@1 rate of 73.0 and 70.0 for PyTorch-Keras and PyTorch-MXNet transpilations, respectively. These scores surpass those of the state-of-the-art LLM, GPT-4, by 16.2 and 15.0 points respectively.
- For training, we construct a PyTorch-Keras-MXNet corpus of deep learning code from various Internet sources, containing 49,705 PyTorch modules, 11,443 Keras layers/models, and 4,785 MXNet layers/models. We then build an evaluation benchmark for PyTorch-Keras and PyTorch-MXNet transpilation. The benchmark evaluates both our API keyword mapping algorithm and the overall source-to-source transpilation.

2.2 Method

ADELT (**A**dversarial **DE**ep **L**earning **T**ranspiler) is an algorithm that transpiles code from a source deep learning framework into an equivalent one in a target framework, by transpiling the skeletal code using a pretrained large language model, and then looking up each keyword in a dictionary learned with unsupervised domain-adversarial training. ADELT applies the following steps to each piece of input code, which we illustrate using the example shown in Figure 2.1:

1. Extract *API calls* from the source code. Such API calls can be automatically extracted with the Python's built-in `ast` library. We then convert each API call into its canonical form, where each layer/function has a unique name, and all of its arguments are converted to keyword arguments. Finally, we extract

all *API keywords* from the canonicalized API call, where an *API keyword* is the name of a layer/function or the name of a keyword argument.

2. Transform the program into its *code skeleton* by replacing each API keyword occurrence with a distinct placeholder.

3. Transpile the code skeleton, where all API keywords are replaced by placeholders, into the target DL framework using a pretrained big LM (e.g., Codex).

4. Look up each API keyword in the *API keyword dictionary*, and replace each keyword with its translation. To generate the API keyword dictionary, we first learn the API embeddings using domain-adversarial training based on contextual embeddings extracted by PyBERT (a BERT pretrained on Python code and then fine-tuned on deep learning code). Next, we calculate the cosine similarity between the embedding vectors. Then we generate the API keyword dictionary using a hierarchical algorithm.

5. Put each API keyword back into the transpiled code skeleton to generate the final output.

We describe each of these steps next in detail.

2.2.1 Canonicalization & API Keyword Extraction

We first parse the source code into an *abstract syntax tree (AST)* with the Python `ast` module. Then, canonicalization and API call extraction are applied to the AST.

Canonicalization. We canonicalize each API call using the following steps during both domain-adversarial training (Section 2.2.3) and inference. Each step involves a recursive AST traversal.

1. Unify the different import aliases of each module into the most commonly used name in the training dataset. For example, `torch.nn` is converted to `nn`.

2. Unify different aliases of each layer/function in a DL library into the name in which it was defined. We detect and resolve each alias by looking at its `__name__` attribute, which stores the callable's original name in its definition.[1] For example, `layers.MaxPool2D` is converted to `layers.MaxPooling2D`.

[1] `https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy`

3. Convert each positional argument of an API call into its equivalent keyword argument. Sort all keyword arguments according to the order defined in the function signature. This is done by linking the arguments of each API call to the parameters of its API signature using the `bind` method from Python's `inspect` module.[2]

API keyword extraction. We define *API keyword* as the name of a layer/function or the name of a keyword argument. Once the input code is canonicalized, we locate each API keyword in the AST and then unparse the AST into the canonicalized source code.

2.2.2 Skeletal Code Transpilation

After canonicalizing the source program, ADELT then replaces all API keywords with a placeholder, turning the source program into its *code skeleton*. Each placeholder has textual form `PLACEHOLDER_i`, where $i = 1, 2, 3, \ldots$. The code skeleton is then translated by Codex using few-shot prompting. The full prompt for this step is shown in Section 2.6.4.

2.2.3 Domain-Adversarial Training

Once the code skeleton is transpiled, we then transpile API keywords. We train the aligned embeddings of API keywords in a domain-adversarial setting. In Section 2.2.4, the embeddings will be used to generate a dictionary that maps an API keyword of the source deep learning framework $\mathcal{X}^{(1)}$ to an API keyword in the target DL framework $\mathcal{X}^{(2)}$.

Figure 2.2 illustrates the domain-adversarial approach of ADELT, and Algorithm 1 shows the pseudocode. A generator maps the contextual representations extracted by PyBERT into hidden states (line 5-8). The alignment of hidden states from different DL frameworks is enforced by the adversarial loss induced by the discriminator (line 17-21), so that output embeddings learned with these hidden states (line 11-14) are also aligned. Next, we describe each step in detail:

Each training example is a pair of API keyword occurrences with their context in the training corpus, denoted by $(x^{(1)}, x^{(2)})$. Each keyword occurrence $x^{(l)}$ is tokenized and encoded as multiple *byte pair encoding (BPE)* [18] tokens. In our unsupervised setting, $x^{(1)}$ and $x^{(2)}$ are independent samples from $\mathcal{X}^{(1)}$ and $\mathcal{X}^{(2)}$ in the training dataset, respectively, and they are not necessarily translations of each other.

[2] `https://docs.python.org/3/library/inspect.html#inspect.Signature.bind`

Algorithm 1 Pseudo-code for domain-adversarial training.

```
for (x_1, y_1), (x_2, y_2) in loader:
  # N samples from X_1, X_2 respectively
  # y_1, y_2: API keyword ids

  h_1 = B(x_1).detach()  # contextual embedding
  h_2 = B(x_2).detach()  # no gradient to PyBERT
  z_1 = G(h_1)  # generator hidden states
  z_2 = G(h_2)  # z_1, z_2: N x d

  # dot product of z_l and output embeddings
  logits_1 = mm(z_1, E_1.view(d, m_1))
  logits_2 = mm(z_2, E_2.view(d, m_2))
  L_CE_1 = CrossEntropyLoss(logits_1, y_1)
  L_CE_2 = CrossEntropyLoss(logits_2, y_2)

  # discriminator predictions
  pred_1 = D(z_1)
  pred_2 = D(z_2)
  labels = cat(zeros(N), ones(N))
  L_D = CrossEntropyLoss(pred_1, labels)
  L_G = CrossEntropyLoss(pred_2, 1 - labels)

  # joint update of G and E_l
  #   to minimize L_CE_l
  optimize(G + E_1 + E_2, L_CE_1 + L_CE_2)
  optimize(D, L_D)  # train the discriminator
  optimize(G, L_G)  # train the generator
```

`B`: PyBERT used as the contextual embedder. `G`, `D`: the generator $\mathcal{G}$ and the discriminator $\mathcal{D}$. `E_l`: a d by m_l matrix, where the i-th column vector is the output embedding of API keyword $w_i^{(l)}$. `mm`: matrix multiplication; `cat`: concatenation

$$\begin{aligned}\mathcal{L}_{\mathcal{D}} = &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 1 | \mathcal{G}(\mathbf{h}^{(1)}))] \\ &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 2 | \mathcal{G}(\mathbf{h}^{(2)}))] \\ \mathcal{L}_{\mathcal{G}} = &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 2 | \mathcal{G}(\mathbf{h}^{(1)}))] \\ &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 1 | \mathcal{G}(\mathbf{h}^{(2)}))]\end{aligned} \tag{2.1}$$

$$\mathcal{L}_{\text{CE}}^{(l)} = -\mathbb{E}_{(x,y)\sim\text{data}^{(l)}} \left[\log \frac{\exp(\mathbf{z} \cdot \mathbf{e}_y^{(l)})}{\sum_{k=1}^{m^{(l)}} \exp(\mathbf{z} \cdot \mathbf{e}_k^{(l)})} \right] \tag{2.2}$$

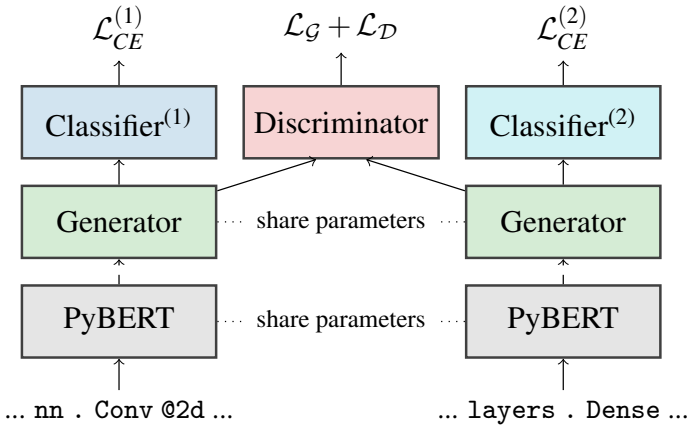


Figure 2.2: ADELT's domain-adversarial training with contextual embeddings from a PyBERT. The generator and the PyBERT are shared between different DL frameworks. We do not fine-tune the PyBERT during adversarial training.

PyBERT. *PyBERT* is our pretrained Transformer [19, 20] for Python code [13, 21, 22]. Given a sequence of BPE tokens that represent an API keyword with its context $x^{(l)}$, PyBERT outputs a sequence of vectors—one vector in $\mathbb{R}^{d_b}$ for each token, where d_b is the hidden dimension size of PyBERT. We average-pool all BPE tokens of the keyword and get a single d_b-dimensional vector as the contextual embedding $\text{PyBERT}(x^{(l)})$ of the API keyword. We denote the contextual embedding of $x^{(1)}, x^{(2)}$ by $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}$ respectively.

Generator and discriminator. We define two multi-layer perceptrons, a generator and a discriminator. A generator $\mathcal{G}$ encodes the contextual embeddings $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}$ into hidden states $\mathbf{z}^{(1)}, \mathbf{z}^{(2)} \in \mathbb{R}^d$, and a discriminator $\mathcal{D}$ is trained to discriminate between $\mathbf{z}^{(1)}$ and $\mathbf{z}^{(2)}$. The generator is trained to prevent the discriminator from making accurate predictions, by making $\mathcal{G}(\text{PyBERT}(\mathcal{X}^{(1)}))$ and $\mathcal{G}(\text{PyBERT}(\mathcal{X}^{(2)}))$ as similar as possible. Our approach is inspired by domain-adversarial training [23], where domain-agnostic representations of images or documents are learned for domain adaptation. In our case, a domain is represented by a DL framework.

Formally, we define the probability $\Pr_{\mathcal{D}}(\text{pred} = l|\mathbf{z})$ that a hidden state $\mathbf{z}$ is from the DL framework l predicted by the discriminator. Note that $\mathbf{z}^{(1)} = \mathcal{G}(\mathbf{h}^{(1)})$ and $\mathbf{z}^{(2)} = \mathcal{G}(\mathbf{h}^{(2)})$. The discriminator loss and the generator loss are computed as the binary cross entropy against the true label and the reversed label, respectively, as shown in Equation (2.1).

Output embeddings. Our goal is to learn an embedding for each API keyword, but the contextual embedding of each keyword occurrence varies with its context. So we instead train a d-dimensional vector $\mathbf{e}_i^{(l)}$ for each API keyword $w_i^{(l)}$, such that $\mathbf{e}_i^{(l)}$ is similar to the generator hidden states $\mathbf{z}_j^{(l)}$ of this keyword's occurrences and dissimilar to the hidden states $\mathbf{z}_k^{(l)}$ of any other keyword's occurrences. $\mathbf{e}_i^{(l)}$ is considered the *output embedding* of the API keyword $w_i^{(l)}$. With similarity computed using dot product, our optimization objective is shown in Equation (2.2), equivalent to the cross-entropy loss of $m^{(l)}$-way softmax-based classification.

Adversarial training. During each training iteration, the generator and discriminator are trained successively to minimize $\mathcal{L}_\mathcal{G}$ and $\mathcal{L}_\mathcal{D}$ respectively with mini-batch stochastic gradient descent. Minimizing the adversarial loss equals to minimizing the distance between two distributions of hidden states [24]. Therefore, the API keywords from the different DL frameworks will be mapped to an aligned embedding space.

Also, we jointly update the generator and the output embeddings to minimize $\mathcal{L}_{\text{CE}}^{(l)}$ with mini-batch SGD. The joint optimization is crucial, as updating the generator to minimize $\mathcal{L}_{\text{CE}}^{(l)}$ ensures that each generator hidden state $\mathbf{z}^{(l)}$ preserves enough information to recover its original API keyword. As a result, the output embeddings $\{\mathbf{e}_i^{(1)}\}_{i=1}^{m^{(1)}}$ and $\{\mathbf{e}_j^{(2)}\}_{j=1}^{m^{(2)}}$ are also aligned, as they are trained with vectors $\mathbf{z}^{(l)}$ from the aligned embedding space.

We do not fine-tune PyBERT during domain-adversarial training, as fine-tuning PyBERT makes the generator disproportionally strong that results in training divergence.

2.2.4 Hierarchical API Dictionary Generation

ADELT calculates a *scoring matrix* using the aligned API keyword embeddings trained in Section 2.2.3. The entry in the i-th row and the j-th column of the matrix is the cosine similarity between $w_i^{(1)}$ and $w_j^{(2)}$, denoted by $s_{i,j}$. Given the scoring matrix, we need to generate an API keyword dictionary that maps each API keyword in one deep learning framework to an API keyword in another DL framework.

Greedy match is used to generate a dictionary in word translation of natural languages [25], where each source word is matched to the target word with the highest similarity score.

Structure of API keywords. Unlike words in NL, API keywords are *structured*: API keywords can be classified into two types based on their associated AST node: *callables names* (names of functions or classes), and *parameter names* (names of keyword arguments). In dictionary generation, we do not allow callable names to be translated to parameter names. We only allow parameter names to be translated to callable names in a special case when the weight passes a threshold. In this case, this parameter will be dropped and generate a new API call (the last case in Table 2.2). Another structural property is that the matching of parameters depends on the matching of callables.

Hierarchical API dictionary generation algorithm leverages the structure of API keywords to generate a dictionary: **Step 1.** Consider each callable and its parameters as a group and compute the *group similarity* between each pair of groups, by summing up similarity scores in the greedy matching of parameter names, plus the similarity between two callable names. **Step 2.** Match groups greedily based on group similarity scores calculated in step 1.

2.3 Experiments

We evaluate the effectiveness of ADELT on the task of transpilation between PyTorch, Keras, and MXNet and compare our method with baselines.

2.3.1 Skeletal Code Transpilation

We use Codex [26], a LLM trained on public GitHub code, to transpile code skeletons. As an autoregressive language model trained on massive web data, Codex can handle translation tasks via prompting with few-shot demonstrations. Our prompt design aligns with Codex's code translation setup, comprising a single input-output example and three instructions to keep placeholders unchanged. Section 2.6.4 provides further details on this.

2.3.2 Training Setup

DL corpus. We consider 3 data sources **GitHub**, **JuiCe** [27], **Kaggle** [28] to build our DL corpus:

- **GitHub**: The GitHub public dataset available on Google BigQuery.[3] We keep `py` and `ipynb` files that contain `torch`, `keras`, or `mxnet` in the `main` and `master` branch of the repository (69GB of clean Python code before filtering, 2.5GB after filtering).
- **JuiCe**: A code generation dataset [27] based on `ipynb` files from GitHub. JuiCe contains many files absent in the public dataset on Google BigQuery, since the latter is a selected subset of GitHub (10.7GB of clean Python code).
- **Kaggle**: All files in KGTorrent [28], a dataset of Jupyter Notebooks from Kaggle[4] (22.1GB of clean Python code).

We tokenize all Python source code and extract subclasses of `torch.nn.Module`, `keras.layers.Layer`, or `keras.Model`. Then, we canonicalize (Section 2.2.1) the code of each class definition. We byte-pair encode [18], merge, and deduplicate codes from all sources. Finally, we collect all files into our *DL Corpus* containing 49,705 PyTorch modules, 11,443 Keras layers/models, and 4,785 MXNet layers/models.

PyBERT is our Transformer encoder pretrained with the masked language modeling (MLM) [20] objective on all open-source Python files from the GitHub dataset. We consider two model sizes: PyBERTsmall (6-layer, 512-d) and PyBERTbase (12-layer, 768-d). Detailed pretraining hyperparameters are described in Section 2.6.1.

Adversarial training. The generator and discriminator of ADELT are multilayer perceptrons. We search the learning rate and batch size according to the unsupervised validation criterion *"average cosine similarity"* [25], which measures the consistency between learned API keyword embeddings and generated keyword translations. Other hyperparameters are set based on previous studies [25] with details described in Section 2.6.2.

2.3.3 Evaluation Benchmark

Our method is evaluated through the task of transpiling code snippets from one DL framework to another. Our benchmark consists of two parts: first, we use heuristics to identify potential matching pairs in the corpus, which were then refined through manual curation to ensure a solid evaluation benchmark; the second part of the

[3] `https://console.cloud.google.com/marketplace/details/github/github-repos`
[4] `https://kaggle.com`

benchmark includes a set of expert-transpiled examples, each accompanied by unit tests. For detailed methodology and statistics, please refer to Section 2.6.3.

We report results in three evaluation metrics:

- **F1 score** quantifies the overlap between the predicted and ground truth outputs. In this context, we treat each prediction or ground truth as a bag of function calls. For each test case, we determine the number of exactly matched calls n_{match}, predicted calls n_{pred}, and ground truth calls n_{truth}. We define the F1 score for a particular example as $2n_{\text{match}}/(n_{\text{pred}} + n_{\text{truth}})$, and report the average F1 scores across all test cases.
- **Exact Match (EM) score** is a more rigorous metric that evaluates whether a model's transpilation is exactly equivalent to the ground truth for each code snippet. It's calculated as the proportion of exact matches to the total number of examples in the eval set.
- **Pass@1** assesses the proportion of examples for which the first transpilation attempt by the model successfully passes all the unit tests. These unit tests, created by experts for each benchmark example, evaluate the correctness of transpilations by execution.

2.3.4 Evaluation of Skeletal Code Transpilation

Transpiling code skeletons of DL programs is an easy task, and Codex easily learned transpilation patterns via few-shot prompting. In our evaluation benchmark, the exact match score of skeletal code transpilation using Codex is 100%.

2.3.5 Comparison with Other Methods

We compare ADELT using PyBERTsmall and ADELT using PyBERTbase with the following baselines. We run all methods 5 times with random seeds `[10, 20, 30, 40, 50]`, and report the arithmetic average of all metrics.

End-to-end language models. We compare ADELT with end-to-end few-shot LLM baselines, including GPT-3, Codex, and GPT-4, where the entire piece of source code, instead of the code skeleton, is fed into the LLM to generate the transpiled target program. For source-to-source translation, we use the *"completion"* endpoint of `code-davinci-002` version for Codex and the *"chat"* endpoint of `gpt-4-0314`

Table 2.1: **Comparison between ADELT and other methods** on source-to-source transpilation. "ADELT (Small)" is ADELT with PyBERTsmall and "ADELT (Base)" is ADELT with PyBERTbase. There are two numbers in each table cell: the first one is for transpiling PyTorch to the other framework (Keras or MXNet), and the second one is for transpiling the other framework to PyTorch. Each number is the average of 5 runs with different random seeds.

	PyTorch-Keras						PyTorch-MXNet					
	F1		EM		Pass@1		F1		EM		Pass@1	
GPT-3 [17]	26.6	32.0	22.4	26.0	23.4	27.2	25.8	32.8	23.4	25.0	25.0	26.4
Codex [26]	59.9	67.1	51.5	54.6	53.4	57.6	57.4	69.0	53.2	56.2	54.2	57.6
GPT-4	67.7	74.9	55.6	64.6	56.8	66.0	60.3	71.8	54.0	60.2	55.0	60.8
Edit Distance (Cased)	31.2	30.1	20.3	16.8	20.3	16.8	37.7	35.7	22.8	21.0	22.8	21.0
Edit Distance (Uncased)	23.9	30.1	12.6	16.8	12.6	16.8	30.8	36.0	18.4	20.0	18.4	20.0
ADELT (Small)	79.0	76.7	70.8	67.6	70.8	67.6	76.7	70.6	66.6	63.0	66.6	63.0
ADELT (Base)	**83.4**	**79.3**	**73.0**	**71.6**	**73.0**	**71.6**	**80.0**	**72.1**	**70.0**	**63.8**	**70.0**	**63.8**

for GPT-4. In both cases, we give the LLM a natural language instruction and 5 examples as demonstrations. Details of the prompts are shown in Section 2.6.5.

Edit distance. We consider a rule-based baseline where we use edit distance [29] as the similarity measure between API keywords, in place of the similarity measures calculated from learned embeddings. We apply hierarchical API dictionary generation exactly as what we do in ADELT. We report the result of both cased and uncased setups for edit distance calculation.

The results in Table 2.1 show that **ADELT consistently outperforms other methods across all metrics**. Notably, ADELT outperforms GPT-4 by significant margins, achieving a 16.2 pts lead in pass@1 of PyTorch-Keras translations and a 5.6 pts lead in Keras-PyTorch. The difference is due to ADELT being based on an encoder-only PyBERT model that leverages larger corpora of the *source* DL framework (PyTorch) more effectively, unlike GPT-4, a decoder-only LLM that benefits from larger corpora for the *target* framework. Therefore, ADELT complements traditional end-to-end LLM methods. Additionally, ADELT runs much faster than GPT-4, as it uses a smaller LLM for transpiling code skeletons and a dictionary lookup step that requires only a tiny fraction of the time needed for full LLM inference.

2.3.6 Case Studies

Table 2.2 shows four examples of PyTorch-Keras transpilation together with hypotheses of Codex and ADELT (Base). Both Codex and ADELT transpile the `nn.Conv2d` to Keras correctly by dropping the first argument `in_channels`. ADELT does not translate the parameter names of `nn.Embedding` to `input_dim` and `output_dim` correctly, while Codex does. However, we notice that Codex sometimes relies on the argument ordering heuristic. In the example of `nn.MultiheadAttention`, where parameters have a different ordering in Keras than in PyTorch, Codex generates the wrong translation, but ADELT successfully constructs the correct mapping between parameters.

Also, in the `nn.Embedding` example, Codex continues to generate code about "positional embeddings" after finishing transpilation. The extra code generated by Codex is relevant to the context.[5] Still, the extra code should not be part of the translation. We have tried various ways to make Codex follow our instructions (see Section 2.6.5 for details). However, because Codex is an end-to-end neural language model, our means of changing its predictions are limited, and the result is highly indeterministic. In the end, Codex still occasionally generates extra arguments or unneeded statements.

On the other hand, we decouple neural network training from the transpilation algorithm. ADELT transpiles between deep learning frameworks using deterministic keyword substitution based on a learned API keyword dictionary. The transpiled code is always syntactically correct. If a mistake is found in the dictionary (e.g., the `nn.Embedding` example in Table 2.2), it can be corrected by simply modifying the dictionary.

Correcting the API keyword dictionary by humans requires much less effort than building the dictionary manually from scratch, as ADELT generates a high-quality dictionary. Developers can even add additional rules to the transpiler. The flexibility of our decoupled design makes ADELT far easier to be integrated into real-world products than end-to-end neural translators/LMs are.

The last case in Table 2.2 shows an example where an API call (`layers.Dense` with `activation="relu"`) should be transpiled to two calls (`nn.Linear` and `nn.ReLU`). One-to-many mapping is rare in transpilation between deep learning frameworks, but the capability to model such mapping reflects the generality of a transpiler to other APIs. Both ADELT and Codex fail to solve this example because this usage is rarely seen in the training data. Still, if we train ADELT on an additional synthetic dataset ("ADELT +" in Table 2.2. See Section 2.6.8 for details), it successfully solves

[5]The definition of positional embeddings usually follows the definition of word embeddings (`nn.Embedding(vocab_size, ...)`) in the source code of a Transformer model.

Table 2.2: **Examples from the evaluation dataset of the PyTorch-Keras transpilation task and the Keras-PyTorch transpilation task.** We show the source code, ground truth target code, and the outputs from Codex, ADELT, and ADELT +. ✓: the output is the same or equivalent to the ground truth. ✓: the output contains an equivalent of the ground truth, but it also contains incorrect extra code. ✗: the output is incorrect.

Source	`nn.Conv2d(64, 128, 3)`	Source	`nn.Embedding(vocab_size, embed_dim)`
Truth	`layers.Conv2D(filters=128, kernel_size=3)`	Truth	`layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)`
Codex ✓	`layers.Conv2D(128, 3)`	Codex ✓	`layers.Embedding(vocab_size, embed_dim)` `self.position_emb = layers.Embedding(...)`
ADELT ✓	`layers.Conv2D(filters=128, kernel_size=3)`	ADELT ✗	`layers.Embedding( embeddings_initializer=embed_dim)`

Source	`nn.MultiheadAttention( model_dim, num_heads=num_heads, dropout=attn_dropout)`	Source	`in_dim = 256`
			`out_dim = 512`
			`layers.Dense(out_dim, activation='relu')`

Truth	`layers.MultiHeadAttention( num_heads=num_heads, key_dim=model_dim, dropout=attn_dropout)`	Truth	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_dim, out_dim)`
			`nn.ReLU()`

Codex ✗	`layers.MultiHeadAttention( model_dim, num_heads, dropout=attn_dropout)`	Codex ✗	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_dim, out_dim)`

ADELT ✓	`layers.MultiHeadAttention( num_heads=num_heads, key_dim=model_dim, dropout=attn_dropout)`	ADELT ✗	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_features=in_dim, out_features=out_dim)`

		ADELT+ ✓	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_features=in_dim, out_features=out_dim)`
			`nn.ReLU()`

Table 2.3: **Ablation study results.** By default, ADELT is trained with the adversarial loss on contextual embeddings extracted by PyBERT, and then a dictionary is generated based on cosine similarity scores. We change one component of ADELT (Small) or ADELT (Base) in each experiment to assess its contribution.

	Keyword				Source Code	
	P@1		MRR		F1	
ADELT (Small)	82.9	90.0	87.0	94.0	79.0	76.7
ADELT (Base)	**87.1**	90.0	**89.7**	94.0	**83.4**	**79.3**
Domain-adversarial training						
w/o PyBERT (Small)	52.1	63.6	60.5	72.8	37.2	43.0
w/o PyBERT (Base)	45.0	54.6	56.8	66.0	33.0	36.3
w/o Adv Loss (Small)	80.4	88.6	85.3	93.1	65.8	73.6
w/o Adv Loss (Base)	86.3	90.5	89.3	94.3	78.2	72.3
Measure for dictionary generation						
Inner Product (Small)	81.3	79.6	86.3	85.4	74.6	73.2
Inner Product (Base)	85.4	**93.2**	88.8	**95.7**	80.2	78.8

this case, showing that our method can model one-to-many mappings when enough training data is available.

2.3.7 Ablation Studies

We conduct ablation studies on PyTorch-Keras transpilation to validate the contribution of each part of ADELT. As illustrated in Figure 2.1, ADELT consists of five steps. Steps 1 (canonicalization and extraction), 2 (code to skeleton), and 5 (skeleton to code) are deterministic and always yield 100% accuracy as they do not rely on machine learning models. The accuracy of Step 3 (code skeleton transpilation) has been discussed in Section 2.3.4. Therefore, this section focus on the primary error source: the *API keyword translation* in Step 4. This crucial step involves mapping API keywords between frameworks. To evaluate its accuracy, we construct a high-quality dictionary by manually translating the top 50 most frequent API keywords from PyTorch to Keras. We then evaluate the translation's efficacy using standard metrics: *precision@k* (for $k = 1, 5$) and the *mean reciprocal rank* (MRR) of correct translations.

The results are shown in Table 2.3, where we study the following variants of ADELT:

Necessity of contextual embeddings. In *"w/o PyBERT"*, we replace PyBERT with Word2Vec [30] embeddings of the same dimensions d_b trained on the same corpora. The result in Table 2.3 shows that this change significantly harms the performance of ADELT. This justifies the use of PyBERT, a high-quality pretrained representation of API keywords that can capture their contexts.

Contribution of adversarial loss. In *"w/o Adv Loss"*, we remove the adversarial loss during training. Instead, we only train the generator and the output embeddings with the cross-entropy loss in Equation (2.2). The result in Table 2.3 shows that adversarial training contributes $\sim$6 pts in source-to-source transpilation, showing the effectiveness of adversarial training.

Comparison of similarity measures. By default, ADELT uses cosine similarity as the similarity measure for API dictionary generation. Table 2.3 shows the results of using dot product (inner). Cosine similarity outperforms dot product by a small margin. This fact implies that the performance of ADELT is insensitive to the choice of similarity measure.

2.4 Related Work

Source-to-source transpilation. Classical source-to-source transpilers use supervised learning. Nguyen et al. [31] and Karaivanov et al. [32] develop Java-C# transpilers using parallel corpora of open-source code. The dependency on parallel corpora renders these methods inapplicable to transpilation between deep learning frameworks, as parallel corpora are difficult to get.

Drawing inspiration from unsupervised neural machine translation (NMT) [15], recent advancements have made unsupervised programming language translation possible [33]. Such approaches, however, require vast amounts of in-domain unlabeled corpora, as evidenced by Lachaux et al. [33] and Roziere et al. [34], who used 744GB of GitHub source code and 333k curated Java functions respectively. The scarcity of DL code hinders their effectiveness for transpilation between DL frameworks.

Metalift [35] is a transpiler generator for various domain-specific languages (DSLs) [36, 37, 38, 39, 40]. It synthesizes target code validated through formal verification. Metalift requires users to explicitly define the target DSL's semantics, while ADELT automatically learns transpilation mappings from data.

Language models are few shot learners. GPT-3 [17] is a language model with 175B parameters trained on massive web crawl data. GPT-3 can be applied to many NLP tasks without any task-specific training, with instructions and few-shot demonstrations specified purely via text interaction with the model. Codex [26] is a GPT-3 fine-tuned on publicly available code from GitHub, specialized for code generation tasks. GPT-4 is a LLM proficient in both code and NL trained using instruction finetuning. In constrast, the code generation step of ADELT is keyword substitution instead of autoregressive generation. ADELT outperforms GPT-3, Codex, and GPT-4 in PyTorch-Keras transpilation and PyTorch-MXNet transpilation.

Adversarial learning & cross-lingual word embedding. Conneau et al. [25] uses domain-adversarial [23] approach to align the distribution of two word embeddings, enabling natural language word translation without parallel data. The domain-adversarial training in ADELT is inspired by their approach, but we align the distributions of the hidden states of *keyword occurrences*.

2.5 Conclusion

In this chapter, we present ADELT, an novel code transpilation algorithm for deep learning frameworks. ADELT decouples code transpilation into two distinct phases: code skeleton transpilation and API keyword mapping. It leverages large language models (LLMs) for the transpilation of skeletal code, while using domain-adversarial training for the creation of an API keyword mapping dictionary. This strategic decoupling harnesses the strengths of two different model types. Through comprehensive evaluations using our specially curated PyTorch-Keras and PyTorch-MXNet benchmarks, we show that ADELT significantly surpasses existing state-of-the-art transpilers in performance.

2.6 Appendix

2.6.1 PyBERT pretraining Hyperparameters and Implementation Details

The models are pretrained with the RoBERTa [41] pipeline in `fairseq`[6] codebase. We pretrain each PyBERT on the GitHub dataset. On a NVIDIA DGX-2, it takes 8.2 hours and 23.1 hours to train $\text{PyBERT}_{\text{SMALL}}$ and $\text{PyBERT}_{\text{BASE}}$, respectively.

[6] `https://github.com/facebookresearch/fairseq`

Table 2.4: Pre-training hyperparameters of PyBERT

Hyperparameter	**PyBERT$_{\text{SMALL}}$**	**PyBERT$_{\text{BASE}}$**
Number of layers	6	12
Hidden size d_b	512	768
FFN inner hidden size	2048	3072
Attention heads	8	12
Attention head size	64	64
Dropout	0.1	0.1
Attention dropout	0.0	0.0
FFN dropout	0.0	0.0
Adam β_1	0.9	0.9
Adam β_2	0.98	0.98
Adam ϵ	1e-6	1e-6
Weight decay	0.01	0.01
Gradient clipping	-	-
Peak learning rate	5e-4	5e-4
Batch size	2,048	2,048
Warmup steps	10,000	10,000
Total steps	125,000	125,000

Table 2.4 shows the pretraining hyperparemters of PyBERT$_{\text{SMALL}}$ and PyBERT$_{\text{BASE}}$. We first pretrain each model on the Github dataset and then fine-tune it on our canonicalized PyTorch-Keras corpus. The learning rate is decayed according to the inverse square root schedule. We do not use early stopping — we use the last PyBERT checkpoint in ADELT.

2.6.2 Domain-Adversarial Training Hyperparameters

The generator and the discriminator of ADELT are multilayer perceptrons. The activation function is ReLU for the generator and Leaky-ReLU for the discriminator. Dropout and label smoothing are applied for regularization. We train our generator, discriminator, and API keyword embeddings with Adam [42] on 1,536,000 samples. There is a linear learning rate warmup over the first 10% of steps, and then we set the LR according to the invert square root decay rule. The learning rate scheduler

Table 2.5: The hyperparameters of domain-adversarial training

Generator activation	ReLU
Generator hidden size	2,048
Generator layers	1
Discriminator hidden size	2,048
Discriminator layers	1
Discriminator activation	LeakyReLU
Discriminator LeakyReLU slope	0.2
Dropout	0.1
Label smoothing	0.2
Warmup step ratio	10%
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	1e-8
Weight decay	0.001
Discriminator iterations per step	1
Total samples	1,536,000
Peak learning rate (Small)	2e-4
Batch size (Small)	128
Peak learning rate (Base)	5e-4
Batch size (Base)	256

uses linear warmup and inverse sqrt decay. The maximum learning rate is searched from `[2e-4, 5e-4, 1e-3]`, and the batch size is searched from `[64, 128, 256]` The peak learning rate and the batch size are searched according to the unsupervised validation criterion *"average cosine similarity"* [25] of the generated dictionary, which quantifies the consistency between the learned API keyword embeddings and the generated keyword translations. We set other hyperparameters according to prior works [25], shown in Table 2.5 (top). The learning rates and the batch sizes selected in the hyperparameter search are shown in Table 2.5 (bottom). The total number of training steps is "total samples" (1,536,000) divided by the searched batch size, which is 6,000 steps for ADELT (Small) and 12,000 steps for ADELT (Base).

2.6.3 Evaluation Data Collection

Our evaluation uses a parallel corpus of 100 examples derived from two distinct methods:

Heuristically Identified Parallel Examples: We sourced open-source projects on GitHub that benchmark various deep learning frameworks by replicating identical neural network architectures across those frameworks. Our parallel pair identification process relied on a heuristic comparing Python class names. Criteria for selection included pairs of PyTorch modules and Keras models/layers that (a) possessed identical class names and (b) achieved a BLEU score above 65. Following heuristic selection, we refined these pairs through manual extraction of code segments containing deep learning API calls, resulting in a corpus of 50 parallel examples. Then, human experts manually transpile those 50 PyTorch examples to MXNet, resulting in a corpus of 50 parallel examples for evaluating PyTorch-MXNet transpilation.

Expert-Transpiled Examples: The second set of 50 examples was assembled by selecting PyTorch module definitions from GitHub repositories with more than 1,000 stars and asking human experts to convert them into the Keras framework. The resulting PyTorch-Keras pairs tend to be longer and more challenging.

2.6.4 Details of Skeletal Code Transpilation

Table 2.6 shows by example how we transpile skeletal codes using Codex few-shot prompting.

1. Each API keyword in the canonicalized source program is replaced with an distinct placeholder, numbered from 1 to n (the number of API keywords). The program after this step is called the code skeleton of the source program.

2. We append the code skeleton to the natural language prompt, *# Translate from PyTorch to Keras*, and four input-output pairs. The first three input-output pairs prompt the model to keep placeholders unchanged during transpilation. Our experiments show that three input-output pairs are required for 100% skeletal code transpilation correctness. Also, Codex can generalize to an arbitrary number of placeholders even if only three is given. The last input-output pair is a real example of PyTorch-Keras skeletal code transpilation.

Table 2.6: Example inputs we give to Codex for skeletal code transpilation. We also show the expected outputs of the language model.

Canonicalized Source Program

```
import torch.nn as nn
dense = nn.Linear(in_features=dim_in, out_features=dim_out, bias=False)
```

Code Skeleton

```
import torch.nn as nn
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in, PLACEHOLDER_3=dim_out, PLACEHOLDER_4=False)
```

Codex Input

```
# Translate from PyTorch to Keras

# PyTorch
PLACEHOLDER_1

# Keras
PLACEHOLDER_1

# PyTorch
PLACEHOLDER_2

# Keras
PLACEHOLDER_2

# PyTorch
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def forward(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# Keras
import tensorflow.keras.layers as layers
class Model(layers.Layer):
    def __init__(self):
        super().__init__()
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def call(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# PyTorch
import torch.nn as nn
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in, PLACEHOLDER_3=dim_out, PLACEHOLDER_4=False)

# Keras
```

Expected Codex Output

```
import tensorflow.keras.layers as layers
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in, PLACEHOLDER_3=dim_out, PLACEHOLDER_4=False)
```

Target Program

```
import tensorflow.keras.layers as layers
dense = layers.Dense(units=dim_out, use_bias=False)
```

3. This entire piece of input is fed into Codex, and Codex will complete this input by generating tokens after `# Keras`. The output of Codex is considered as the code skeleton of the target program.

4. Each placeholder is replaced with the API keyword in the target DL framework, by querying each API keyword before replacement (step 1) in the API keyword dictionary learned with ADELT.

If the number of placeholders in the source skeleton and the number of placeholders in Codex's output do not match, it is considered a failed example in evaluation. However, in practice, the success rate of skeletal code transpilation is 100% in our experiments. We attribute that to the fact that skeletal code in DL programs, in comparison to arbitrary Python code, tend to be high structured with fairly predictable import statements, constructors, and how the different DL layers are constructed and connected to each other.

2.6.5 Evaluation Setup of LLMs

Following the practices in Brown et al. [17] and Chen et al. [26], we use the *"completion"* endpoint of GPT-3 or Codex and the *"chat"* endpoint of GPT-4 for transpilation. We input some text as a prompt with few-shot demonstrations, and the model will generate a completion that attempts to match the prompt. Table 2.7 shows two examples illustrating how we leverage GPT-3 or Codex for our task. Table 2.8 shows two examples illustrating how we leverage GPT-4 for our task.

For source-to-source transpilation, prompt engineering is straightforward. In the PyTorch-Keras transpilation example, we tell the model to "`# Translate from PyTorch to Keras`" and then give 5 demonstrations from our evaluation dataset. Next, we input a piece of source code and "`# Keras`" and let the model generate a code completion starting from the following line. For chat models like GPT-4, we formulate the prompt in Markdown format, and extract the contents first Markdown code block as the model's output. To prevent answers from being leaked to the language model, we do not allow any demonstration to share common API functions with the current evaluation example.

Prompt engineering of API keyword translation is trickier because there are two types of keywords. We represent callable names by one line containing its textual representation, and we represent parameter names by two lines, where the first line is the name of the callable that the parameter belongs to, and the second line is the name of the parameter. We give 10 demonstrations from our evaluation dataset.

Table 2.7: Example inputs we give to GPT-3 or Codex for source-to-source transpilation and API keyword translation. We also show the expected outputs of the language models.

Source-to-Source Transpilation	**Keyword Translation**

`# Translate PyTorch to Keras`
 	
`# PyTorch`
`max_len = 512`
`self.embed_tokens = nn.Embedding(`
`  n_words, dim_emb)`
`# Keras`
`max_len = 512`
`self.embed_tokens = layers.Embedding(`
`  n_words, dim_emb, input_length=max_len)`
 	
`# PyTorch`
`nn.Linear(dim_in, dim_out)`
`# Keras`
`layers.Dense(dim_out)`
 	
(2 demonstrations omitted)
 	
`# PyTorch`
`F.log_softmax(logits, dim=-1)`
`# Keras`
`tf.nn.log_softmax(logits, axis=-1)`
 	
`# PyTorch`
`nn.Conv2d(64, 128, 3)`
`# Keras`
`layers.`
	`# Translate PyTorch to Keras`
	
	`# PyTorch`
	`F.log_softmax`
	`# Keras`
	`tf.nn.log_softmax`
	
	`# PyTorch`
	`nn.MaxPool2d`
	`stride`
	`# Keras`
	`layers.MaxPooling2D`
	`strides`
	
	(7 demonstrations omitted)
	
	`# PyTorch`
	`F.relu`
	`# Keras`
	`tf.nn.relu`
	
	`# PyTorch`
	`nn.Conv2d`
	`out_channels`
	`# Keras`
	`layers.`

`Conv2D(128, 3)`	`Conv2D` `filters`

Table 2.8: Example inputs we give to GPT-4 for source-to-source transpilation and API keyword translation. We also show the expected outputs of the language models. Because GPT-4 outputs Markdown texts including both NL and code, we extract contents of the first Markdown code block as the output of the model.

Source-to-Source Transpilation

````
You are an expert in deep learning.
Transpile PyTorch to Keras:

PyTorch:

```python
max_len = 512
self.embed_tokens = nn.Embedding(
  n_words, dim_emb)
```

Keras:

```python
max_len = 512
self.embed_tokens = layers.Embedding(
  n_words, dim_emb, input_length=max_len)
```

PyTorch:

```python
nn.Linear(dim_in, dim_out)
```

Keras:

```python
layers.Dense(dim_out)
```

(2 demonstrations omitted)

PyTorch:

```python
F.log_softmax(logits, dim=-1)
```

Keras:

```python
tf.nn.log_softmax(logits, axis=-1)
```

PyTorch:

```python
nn.Conv2d(64, 128, 3)
```

Keras:
````

````
Conv2D(128, 3)
````

Keyword Translation

````
You are an expert in deep learning.
Transpile PyTorch to Keras:

PyTorch:

```python
F.log_softmax
```

Keras:

```python
tf.nn.log_softmax
```

PyTorch:

```python
nn.MaxPool2d
stride
```

Keras:

```python
layers.MaxPooling2D
strides
```

(7 demonstrations omitted)

PyTorch:

```python
F.relu
```

Keras:

```python
tf.nn.relu
```

PyTorch:

```python
nn.Conv2d
out_channels
```

Keras:
````

````
Conv2D
filters
````

Although GPT-3 and Codex have strong capabilities in generating code related to our prompt, we find that they sometimes fail to follow our instructions to transpile between deep learning frameworks. We discuss this problem in Section 2.3.5. We try several approaches to mitigate this issue:

1. Use the *Instruct* version of GPT-3/Codex.
2. Add a prefix to the input prompt based on simple rules. For example, if the source code starts with `nn.` in PyTorch, add `layers.` to the prompt and let the model generate a code completion after it. This trick is applicable to two examples shown in Table 2.7.
3. Mask the logits of tokens that usually leads to irrelevant generations. Specifically, we find that the model tends to generate irrelevant extra code after a line break or random comments. So we add a bias of -100 to the logits of the hash mark "`#`". We also add a bias of -100 to the logits of the line break if the source code contains no line breaks.

We find that these measures significantly improve the performance of GPT-3 and Codex on deep learning transpilation. All results of GPT-3 and Codex reported in Section 2.3.5 are from the LMs with all these tricks turned on.

Conversely, GPT-4 did not exhibit similar issues, and given that it does not support logits masking, we engaged it directly using the prompts in Table 2.8 on `gpt-4-0314` without additional alterations.

2.6.6 Cross-Domain Local Scaling (CSLS)

Cross-Domain Local Scaling (CSLS) is a similarity measure for creating a dictionary based on high-dimensional embeddings. CSLS was proposed by Conneau et al. [25] for word translation between natural languages. Empirical results by Conneau et al. [25] show that using a pairwise scoring matrix (e.g. cosine similarity, dot product) in dictionary generation suffers from the *hubness* problem [43], which is detrimental to generating reliable matching pairs as some vectors, dubbed *hubs*, are the nearest neighbors to many other vectors according to s, while others (anti-hubs) are not nearest neighbors of any point. This problem is observed in various areas [44, 45]. CSLS is proposed to mitigate the hubness problem.

We also conduct an experiment to verify the effectiveness of CSLS in API keyword translation between deep learning frameworks. Specifically, we denote by $\mathcal{N}_s^{(l)}(w)$ the *neighborhood* of API keyword w, a set consisting of K elements with the highest

similarity scores with w in DL framework $\mathcal{X}^{(l)}$. We calculate the average similarity score of $w_i^{(1)}$ to its neighborhood in DL framework $\mathcal{X}^{(2)}$ and denote it by $r_i^{(2)}$. Likewise, we denote by $r_j^{(1)}$ the average similarity score of $w_j^{(2)}$ to its neighborhood in DL framework $\mathcal{X}^{(1)}$. Then we define a new similarity measure CSLS of $w_i^{(1)}$ and $w_i^{(2)}$ by subtracting $r_i^{(2)}$ and $r_j^{(1)}$ from their (doubled) similarity score $s_{i,j}$, as shown in Equation (2.3).

$$\begin{aligned} r_i^{(2)} &= \frac{1}{K} \sum_{k \in \mathcal{N}_s^{(2)}(w_i^{(1)})} s_{i,k} \\ r_j^{(1)} &= \frac{1}{K} \sum_{k \in \mathcal{N}_s^{(1)}(w_j^{(2)})} s_{k,j} \\ \mathrm{CSLS}_{i,j} &= 2s_{i,j} - r_i^{(2)} - r_j^{(1)} \end{aligned} \tag{2.3}$$

CSLS can be induced from a parameter K and any similarity measure, including dot product and cosine similarity. Intuitively, compared with the score matrix of similarity measure s, the score matrix of CSLS assigns higher scores associated with isolated keyword pairs and lower scores of keywords lying in dense areas.

Given the (cosine similarity) scoring matrix scaled by CSLS, we then apply the hierarchical dictionary generation algorithm (Section 2.2.4) to generate the API keyword dictionary. We search K in $\{5, 10, 20\}$ according to the unsupervised evaluation metric, and the result is similar, where $K = 5$ gives a slightly better result. Table 2.9 shows the result of cosine-CSLS compared with cosine similarity.

Table 2.9 shows that replacing cosine similarity with cosine-CSLS-5 does not impact the F1 score of transpiling PyTorch to Keras significantly, but it hurts the F1 score of transpiling Keras to PyTorch. The reason is that the vocabulary of API keywords is smaller than a natural language vocabulary. Hubness is not a problem for generating API keyword dictionaries; instead, penalizing the top-K may hurt the performance when there are relatively few valid candidates (e.g. Keras-to-PyTorch transpilation). Therefore, we do not use CSLS for ADELT.

2.6.7 Full Results with Error Bars

Table 2.10 shows full results with error bars for PyTorch-Keras API keyword translation and source-to-source transpilation. The table includes the results of both the main comparison with GPT-3/Codex and ablation studies. We also add the results of GPT-3 and Codex on API keyword translation, where we randomly give the

Table 2.9: **Results of CSLS.** By default, ADELT computes similarity scores using *cosine similarity* to generate an API keyword dictionary. In this experiment, we replace cosine similarity with *inner product* or *cosine-CSLS-5* to compare different similarity measures. There are two numbers in each table cell: the first one is for transpiling PyTorch to PyTorch, and the second one is for transpiling Keras to PyTorch.

	Keyword						Source-to-Source			
	P@1		P@5		MRR		BLEU		F1	
ADELT (Small)	82.92	90.00	91.67	**97.73**	86.97	94.04	93.83	**92.13**	80.67	80.90
ADELT (Base)	**87.08**	90.00	91.67	**97.73**	89.67	93.96	**95.32**	91.29	**85.72**	**82.01**
Inner Product (Small)	81.25	79.55	91.67	90.00	86.34	85.38	93.24	88.49	78.67	77.08
Inner Product (Base)	85.42	**93.18**	91.67	**97.73**	88.84	**95.71**	94.38	91.75	82.17	81.46
cos-CSLS-5 (Small)	84.17	83.18	**97.92**	93.64	89.89	89.12	94.24	90.43	83.17	76.60
cos-CSLS-5 (Base)	**87.08**	89.55	97.50	**97.73**	**90.63**	93.75	95.20	90.27	85.39	76.18

GPT-3 and Codex 10 examples as demonstrations. Details about prompt designs and hyperparameter setup are shown in Section 2.6.5. We do not calculate precision@5 and mean reciprocal rank for GPT-3 and Codex because the API provided by OpenAI does not support ranking a large number of generations cost-efficiently.

2.6.8 ADELT+

We created a new model, ADELT+, which is based on ADELT but trained on a synthetic dataset. Our goal is to evaluate whether our method can generalize to one-to-many mappings of APIs given enough data.

As we discussed in Section 2.2.4, we allow parameter names to be translated to callable names when the weight passes a threshold τ. In this case, this parameter will be dropped and a new API call will be generated. This mechanism allows ADELT to transpile `layers.Dense(..., activation="relu")` into two layers: `nn.Linear(...)` and `nn.ReLU()`, and similarly `layers.Conv2D(..., activation="relu")` into `nn.Conv2D(...)` and `nn.ReLU()`. However, such cases are rare in transpiling between deep learning frameworks, making it difficult to evaluate our model's ability to transpile one-to-many mappings in practice. Therefore, we create a synthetic dataset, where we replace all consecutive calls of `layers.Dense` and `layers.ReLU` in our dataset with `layers.Dense(..., activation="relu")`, and we

Table 2.10: **Full results with 95% confidence intervals**. For each experiment, we run five experiments with different random seeds. Each cell has two intervals: the first one is for transpiling PyTorch to Keras, and the second one is for transpiling Keras to PyTorch. Each interval is the 95% confidence interval according to the Student's t-Test, where we assume that the result of the five experiments follows a normal distribution.

	Keyword		Source-to-Source	
	P@1		F1	
LM few shot				
GPT-3 [17]	35.4 ± 6.1	39.1 ± 4.2	26.6 ± 5.1	32.1 ± 6.7
Codex [26]	67.5 ± 8.3	79.1 ± 7.8	59.9 ± 2.7	67.1 ± 2.2
GPT-4	74.5 ± 5.8	83.2 ± 3.5	67.7 ± 2.6	74.9 ± 1.7
ADELT				
ADELT (Small)	82.9 ± 1.2	90.0 ± 1.6	79.0 ± 2.2	76.7 ± 1.5
ADELT (Base)	87.1 ± 1.2	90.0 ± 2.5	83.4 ± 0.8	82.0 ± 2.2
w/o PyBERT (Small)	52.1 ± 2.6	63.6 ± 4.5	37.2 ± 8.2	43.0 ± 4.5
w/o PyBERT (Base)	45.0 ± 3.9	54.6 ± 5.3	33.0 ± 6.5	36.3 ± 3.2
w/o Adv Loss (Small)	80.4 ± 1.4	88.6 ± 2.0	65.8 ± 2.0	73.6 ± 1.8
w/o Adv Loss (Base)	86.3 ± 1.4	90.5 ± 2.4	78.2 ± 2.1	72.3 ± 3.6
Dot product (Small)	82.9 ± 4.5	90.0 ± 7.2	74.6 ± 2.7	73.2 ± 3.4
Dot product (Base)	87.1 ± 1.2	90.0 ± 2.0	80.2 ± 0.7	78.8 ± 0.8

replace all consecutive calls of `layers.Conv2D` and `layers.ReLU` with `layers.Conv2D(..., activation="relu")`. Then we train a new model, ADELT+, using our synthetic dataset.

We then evaluated ADELT+ using our evaluation dataset. The value of the threshold τ is set heuristically to 5 (95% of values in the score matrix lies in -7 to 7). Table 2.2 in Section 2.3.6 and Table 2.11 in the appendix show that ADELT+ can model one-to-many mappings of APIs. For instance, Table 2.11 shows that ADELT can transpile `layers.Conv2D` with `activation='relu'` into two API calls: `nn.Conv2d` and `nn.ReLU`.

Table 2.11: **A synthetic example of convolution layer from the evaluation dataset of the Keras-PyTorch transpilation task.** We show the Keras code, ground truth PyTorch code, and the outputs from Codex, ADELT, and ADELT +. ✓: the output is the same or equivalent to the ground truth. ✓: the output contains an equivalent of the ground truth, but it also contains incorrect extra code. ✗: the output is incorrect.

Source	`in_dim = 64`
	`out_dim = 128`
	`layers.Conv2D(filters=out_dim,`
	`  kernel_size=3,`
	`  activation="relu")`

Truth	`in_dim = 64`
	`out_dim = 128`
	`nn.Conv2d(in_dim, out_dim, 3)`
	`nn.ReLU()`

Codex ✗	`in_dim = 64`
	`out_dim = 128`
	`nn.Conv2d(in_dim, out_dim, 3)`

ADELT ✗	`in_dim = 64`
	`out_dim = 128`
	`nn.Linear(in_features=in_dim,`
	`  out_features=out_dim,`
	`  kernel_size=3)`

ADELT+ ✓	`in_dim = 64`
	`out_dim = 128`
	`nn.Conv2d(in_channels=in_dim,`
	`  out_channels=out_dim,`
	`  kernel_size=3)`
	`nn.ReLU()`

Table 2.12: Addtional case study 1.

```
# Source Program
import torch.nn as nn
class BasicBlock(nn.Module):
    def __init__(self, dim):
        super.__init__()
        self.bn1 = nn.BatchNorm2d(dim)
        self.act1 = nn.LeakyReLU(0.2)
        self.conv1 = nn.Conv2d(dim, dim, 3)
        self.pool1 = nn.MaxPool2d(3, 2)


# Transpiled by ADELT
import tensorflow.keras.layers as layers
class BasicBlock(layers.Layer):
    def __init__(self, dim):
        super.__init__()
        self.bn1 = layers.BatchNormalization()
        self.act1 = layers.LeakyReLU(alpha=0.2)
        self.conv1 = layers.Conv2D(filters=dim, kernel_size=3)
        self.pool1 = layers.MaxPooling2D(pool_size=3, stride=2)


# Ground Truth
import tensorflow.keras.layers as layers
class BasicBlock(layers.Layer):
    def __init__(self, dim):
        super.__init__()
        self.bn1 = layers.BatchNormalization()
        self.act1 = layers.LeakyReLU(0.2)
        self.conv1 = layers.Conv2D(dim, 3)
        self.pool1 = layers.MaxPooling2D(3, 2)
```

2.6.9 More Case Studies

In Table 2.12 and Table 2.13, we select two PyTorch-Keras cases in our evaluation dataset for illustration. They are examples of the average length of all evaluation examples in the evaluation set.

In each case, ADELT makes the correct transpilation. The only textual difference is that ADELT's transpilation only contains keyword arguments while the ground truth still contains positional arguments. However, because the prediction and the ground truth are the same after canonicalization, we consider each case as an exact match during evaluation.

Table 2.13: Addtional case study 2.

```
# Source Program
import torch.nn as nn
class AttentionBlock(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.attn = nn.MultiheadAttention(
            args.d_model, args.n_heads, dropout=args.att_dropout)
        self.drop1 = nn.Dropout(args.dropout)
        self.norm1 = nn.LayerNorm(args.d_model)


# Transpiled by ADELT
import tensorflow.keras.layers as layers
class AttentionBlock(layers.Layer):
    def __init__(self, args):
        super().__init__()
        self.attn = layers.MultiHeadAttention(
            num_heads=args.n_heads, key_dim=args.d_model, dropout=args.att_dropout)
        self.drop1 = layers.Dropout(rate=args.dropout)
        self.norm1 = layers.LayerNormalization()


# Ground Truth
import tensorflow.keras.layers as layers
class AttentionBlock(layers.Layer):
    def __init__(self, args):
        super().__init__()
        self.attn = layers.MultiHeadAttention(
            args.n_heads, args.d_model, dropout=args.att_dropout)
        self.drop1 = layers.Dropout(args.dropout)
        self.norm1 = layers.LayerNormalization()
```

2.6.10 Deep Learining Transpilation across Different Programming Languages

In the main paper, all experiments are conducted on Python due to the scarcity of deep learning programs written in other programming languages such as Java or C. Despite that, in this section we show that ADELT is not limited to the same source and target languages by transpiling code written against the PyTorch library in Python 2 to Keras in Python 3.

To do so, we first canonicalize all PyTorch programs into Python 2 and all Keras programs into Python 3. Then we run ADELT on this modified training data to learn the API keyword dictionary. During inference, we transpile the code skeleton

Table 2.14: Example of transpiling from PyTorch in Python 2 to Keras in Python 3.

Canonicalized Source Program in Python 2

```
import torch.nn as nn
dense = nn.Linear(in_features=dim_in / 2, out_features=dim_out / 2, bias=False)
```

Code Skeleton

```
import torch.nn as nn
print dim_in, dim_out
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in / 2, PLACEHOLDER_3=dim_out / 2, PLACEHOLDER_4=False)
```

Codex Input

```
# Translate from PyTorch in Python2 to Keras in Python3

# PyTorch in Python2
PLACEHOLDER_1

# Keras in Python3
PLACEHOLDER_1

# PyTorch in Python2
PLACEHOLDER_2

# Keras in Python3
PLACEHOLDER_2

# PyTorch in Python2
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        print "Building Model"
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16 / 2, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def forward(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# Keras in Python3
import tensorflow.keras.layers as layers
class Model(layers.Layer):
    def __init__(self):
        super().__init__()
        print("Building Model")
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16 // 2, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def call(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# PyTorch in Python2
import torch.nn as nn
print dim_in, dim_out
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in / 2, PLACEHOLDER_3=dim_out / 2, PLACEHOLDER_4=False)

# Keras in Python3
```

Codex Output

```
import tensorflow.keras.layers as layers
print(dim_in, dim_out)
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in // 2, PLACEHOLDER_3=dim_out // 2, PLACEHOLDER_4=False)
```

Target Program in Python 3

```
import tensorflow.keras.layers as layers
print(dim_in, dim_out)
dense = layers.Dense(units=dim_out // 2, use_bias=False)
```

with Codex using the prompt shown in Table 2.14. Besides adding hint words such as "Python2" and "Python3" into the natural language prompt, we also find it necessary to add to the prompt some examples showing differences between Python 2 and Python 3, such as different `print` statements and different integer division operators. As is shown in Table 2.14, the skeletal codes were successfully transpiled from Python 2 and Python 3 along with the API keywords.

Unfortunately, because almost all deep learning codes are written in the same programming language, Python 3, it is impossible to give a quantitative analysis of the performance of ADELT across programming languages. This section is an illustration of how ADELT can do such transpilation under the assumption that: (a) there is enough unlabeled training data available in both languages to train a joint LM and an API keyword dictionary; (b) there is an existing general-purpose transpiler available to transpile between two programming languages.

Chapter 3

SAFIM: Evaluation of LLMs on Syntax-Aware Code Fill-in-the-Middle Tasks

We introduce **S**yntax-**A**ware **F**ill-**i**n-the-**M**iddle (SAFIM), a new benchmark for evaluating Large Language Models (LLMs) on the code Fill-in-the-Middle (FIM) task. This benchmark focuses on syntax-aware completions of program structures such as code blocks and conditional expressions, and includes 17,720 examples from multiple programming languages, sourced from recent code submissions after April 2022 to minimize data contamination. SAFIM provides a robust framework with various prompt designs and novel syntax-aware post-processing techniques, facilitating accurate and fair comparisons across LLMs. Our comprehensive evaluation of 15 LLMs shows that FIM pretraining not only enhances FIM proficiency but also improves Left-to-Right (L2R) inference using LLMs. Our findings challenge conventional beliefs and suggest that pretraining methods and data quality have more impact than model size. SAFIM thus serves as a foundational platform for future research in effective pretraining strategies for code LLMs. The evaluation toolkit and dataset are available at `https://github.com/gonglinyuan/safim`, and the leaderboard is available at `https://safimbenchmark.com`.

3.1 Introduction

Recent advances in Large Language Models (LLMs) such as GPT-3.5 [46], GPT-4 [47], and CodeLLaMa [48] have revolutionized coding-related tasks. However, existing benchmarks like HumanEval [26] and MBPP [49] focus on generating standalone

functions or single-file code from natural language descriptions, and do not consider the more common practice of modifying and expanding existing code during development.

Recognizing this gap, we introduce the **S**yntax-**A**ware **F**ill-**i**n-the-**M**iddle (SAFIM) benchmark. SAFIM emphasizes syntax-aware completion within code's Abstract Syntax Tree (AST), targeting algorithmic blocks, control-flow expressions, and API function calls, unlike existing Fill-in-the Middle (FIM) benchmarks such as HumanEval-Infilling [50], which are based on filling randomly masked lines or character spans. SAFIM is sourced from code on Codeforces and GitHub created after April 2022, deliberately aiming to avoid overlap with mainstream open-source pretraining corpora like The Stack [51]. This approach reduces the risks of data contamination caused by memoization of test cases, thereby bolstering the credibility of our results. SAFIM, with its 17,720 examples from 8,590 code files, not only surpasses the scale of HumanEval-Infilling, which draws from 164 short code files, but also expands the scope to include multiple programming languages. SAFIM primarily relies on execution-based evaluation, and uses syntactical match evaluation only when execution is not feasible due to external API calls.

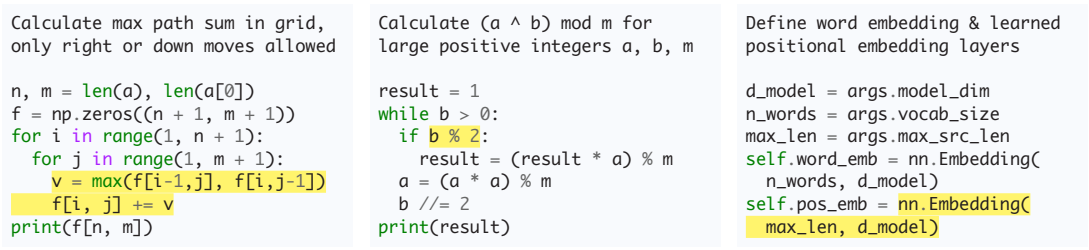
Our comprehensive evaluation of 15 LLMs on SAFIM reveals its effectiveness in providing a fair comparison of models. We implement five distinct prompt designs to accommodate various model types and introduce a syntax-aware truncation algorithm for post-processing the outputs. Our approach unveils the true capabilities of non-FIM-trained models, allowing for a fair comparison with FIM-trained models.

Moreover, SAFIM sheds light on the strengths of various pretraining paradigms and challenges some prevalent beliefs in the field. Specifically, our findings suggest that FIM pretraining not only improves LLMs' performance in FIM inference but also enhances their performance in classical Left-to-Right (L2R) inference scenarios. This supports the growing trend of using FIM as the primary pretraining objective in code LLM development. We also observe that pretraining methods and data quality often outweigh the sheer model size—smaller models with sophisticated pretraining paradigms often outperform larger models. This is particularly evident in task-specific performances on SAFIM, where models pretrained with additional repo-level information excel in API function call completion, while those trained with code execution feedback perform better in control-flow expression generation. However, it is crucial to note that these comparisons across different model families are not controlled experiments and could be influenced by differences in pretraining environments. This suggests future work in pretraining such models under the same environment to validate these observations further. That said, our benchmark, SAFIM, provides a solid foundation for such future research, and opens up new opportunities in designing effective pretraining and fine-tuning paradigms for code LLMs.

3.2 Related Work

Large Language Models for Code. The emergence of Large Language Models (LLMs) like GPT-3 [17] in natural language processing has led to the understanding that merely increasing the number of parameters in pretrained language models will ensure superior performance on unseen tasks. This has led to the application of LLMs to code-related tasks, particularly in code generation. For such tasks, decoder-only models are typically used. Initially, these models, such as Codex [26], PaLM [52], PolyCoder [53], and CodeGen [1], primarily focused on Left-to-Right (L2R) pretraining, a.k.a. "Next Token Prediction." However, the Fill-in-the-Middle (FIM) objective, a.k.a. "Infilling," has become increasingly popular, with models like InCoder [54], StarCoder [55], SantaCoder [56], DeepSeek-Coder [57], and CodeLLaMa [48] showing their effectiveness. Additionally, proprietary models such as GPT-3.5 [46], GPT-4 [47], and Gemini [58], which use undisclosed pretraining methods, also contribute to this domain. While GLM-like models [59] or encoder-decoder models, including CodeGeeX [60], PLBART [61], AlphaCode [62], CodeT5 [14, 63], and AST-T5 [64] exist, they are outside of our paper's scope. Our paper evaluates a select group of these LMs using the SAFIM benchmark. We develop insights into their performance in code FIM tasks, explore the strengths and weaknesses of various pretraining paradigms, and challenge the prevailing belief that a larger number of parameters automatically leads to better performance.

Benchmarking Generative Code LLMs. Existing benchmarks for code generation in LLMs have a gap in effectively evaluating code generation capability for real-world development. Widely-used benchmarks like HumanEval [26] and MBPP [49] are limited to single Python functions and also subject to data contamination [65]. Extensions like HumanEval-X [60], MultiPLe [66], and MBXP [67] expand these benchmarks to other programming languages. Competition-style coding benchmarks like APPS [68] and CodeContests [62], broaden the scope to file-level code generation. However, they still do not reflect typical development, which often involves iterative codebase expansion and invoking external API libraries. On the other hand, contextually richer benchmarks, such as JuICe [27], DS-1000 [69], ARCADE [70], NumpyEval [71], and PandasEval [72], PlotCoder [6], ADELT [73] in data science, and APIBench [74], RepoBench [75], ODEX [76], SWE-Bench [77], GoogleCodeRepo [78], RepoEval [79], and CoCoMIC-Data [80] in software engineering, are often very small, heavily reliant on imperfect match-based evaluation metrics, or lacking in execution-based evaluation. Our SAFIM benchmark, based on Fill-in-the-Middle (FIM) tasks, bridges this gap by providing a comprehensive evaluation framework.



Algorithmic Block Completion | Control-Flow Completion | API Function Call Completion

Figure 3.1: Three splits in the SAFIM benchmark illustrated with code examples. Each example includes a problem description and a code snippet, with a contiguous code segment highlighted in yellow to indicate the part to be masked and completed by LLMs. Contexts in these examples are shortened for clarity.

Fill-in-the-Middle in Training and Evaluating Code LLMs. Fill-in-the-Middle (FIM) originates from masked language modeling (MLM) for training encoder-only models [20] and T5-style span corruption for training encoder-decoder models [81], with span lengths usually limited to 1 to 5 tokens, with the goal of targeting representation learning rather than generation. For coding tasks, InCoder [54] shows the effectiveness of FIM as a pretraining objective for decoder-only models. Fried et al. [54] further establishes the HumanEval-Infilling benchmark, further explored by Bavarian et al. [50] in evaluating GPT-3/Codex variants, showing that a pretraining mix with a 90% FIM ratio does not harm Left-to-Right (L2R) generation performance. CodeLLaMa's evaluations on HumanEval-Infilling support these findings, underscoring the value of FIM in pretraining code-focused LLMs [48]. However, this benchmark, limited to the 164 tiny Python snippets of HumanEval, emphasize the need for a more robust benchmark. SAFIM addresses this need by introducing a comprehensive, syntax-aware FIM benchmark for more detailed evaluations.

3.3 Benchmark Construction

The SAFIM benchmark is designed to evaluate Large Language Models (LLMs) on the Fill-in-the-Middle (FIM) of various code structures. In this section, we describe the collection of the corpora, the generation and filtering of completion tasks, and the evaluation protocols.

3.3.1 Corpora Collection

The SAFIM benchmark is constructed using corpora from two primary sources: *Codeforces* and *GitHub*. Codeforces,[1] a competitive programming platform, offers a wealth of coding problems, unit tests, and solutions. From Codeforces, we scrape problems, unit tests, and their corresponding code solutions. For GitHub, we gather git commits from the GH Archive[2]. From both sources, we gather Python, Java, C++, and C# code files created between April 1, 2022, and January 1, 2023. This selection criteria ensures the inclusion of recent code, avoiding overlap with major pretraining datasets like The Stack [51] (cutoff at March 31, 2022) and the training data for GPT-3.5/GPT-4 (cutoff at September 2021), thus reducing the risk of data contamination.

In processing Codeforces data, we reevaluate each code solution by executing unit tests. We retain only those solutions that consistently pass all unit tests within 50% of the specified time limit, eliminating randomness and noise from external factors. We also filter out excessively lengthy (over twice the size of the shortest accepted solution) or near-duplicate solutions (exceeding a CodeBLEU [3] score threshold of 0.9 against previously added code), resulting in a curated set of 490 coding questions and 8,590 unique code solutions.

For GitHub, we first establish a list of widely-used API libraries for each programming language, detailed in Section 3.8.1. We then extract code files that invoke APIs from such repositories with more than 10 stars to prioritize high-quality code. Files lacking natural language comments or documentation are excluded to avoid unsolvable examples. After thorough filtering and deduplication, our final GitHub corpus consists of 11,936 code files.

3.3.2 Generating and Filtering Completion Tasks

With our corpora ready, we parse each code file into an Abstract Syntax Tree (AST). This enables the creation of structured FIM tasks across three splits: algorithmic block completion, control-flow completion, and API function call completion. The first two are based on the Codeforces corpus, while the latter is based on the GitHub corpus as external API function calls are usually absent in competitive programming. In each split, we mask different code segments and ask the models to reconstruct these segments such that the original program functionality is maintained.

[1] `https://codeforces.com/`

[2] `https://www.gharchive.org/`

Algorithmic Block Completion. Here, we mask a code block critical for solving the coding question, evaluating the LLM's capability in interpreting natural language descriptions and designing algorithms. A "code block" refers to a contiguous list of statements, identified by indentations for Python or curly braces for C-family languages. We target the deepest block in the AST, often the innermost loop layer containing key operations or formulae, like a dynamic programming state transition equation (see Figure 3.1, Left). To avoid masking non-critical blocks (e.g., logging or debugging), we validate each block: if replacing a block with no-op causes unit test failures, it is included; otherwise, it is excluded. Such filtering ensures that only algorithmically significant blocks are included in the benchmark.

Control-Flow Completion. This category focuses on masking critical control expressions in the program, evaluating the LLM's understanding of code control flows. We mask conditional expressions in statements such as `for`, `while`, `do-while`, `for-each`, `if`, and `else-if`. For example, in Figure 3.1 (Middle), we mask `b % 2` in an `if` statement, as it determines when the `result` variable will be updated; we mask `b > 0` of the outer layer `if` in a different example. To ensure the relevance of each masked expression, we only retain cases where substituting the expression with `false`, `true`, or an empty iterable would affect the unit test outcomes. Such filtering guarantees that only expressions critical to the program's control-flow are included in the benchmark.

API Function Call Completion. In this category, we mask calls to functions and object constructors from popular API libraries. This tests the LLM's API knowledge and the ability to integrate such knowledge with code context. Because this split is sourced from the inherently noisy GitHub corpus, we curate the dataset and add necessary hints as comments near each API call, ensuring each example is solvable by humans based on the given context. For example, in Figure 3.1 (Right), the LLM is expected to deduce the correct arguments `max_len` and `d_model` for a positional embedding layer defined by `nn.Embedding`.

The SAFIM benchmark has 17,720 examples across these three categories, with detailed statistics provided in Section 3.8.2.

3.3.3 Evaluation Protocols

We evaluate completions generated by LLMs using *execution-based testing* and *syntactical matching*. The former applies to algorithmic block and control-flow completions, while the latter is used for API function call completion.

Execution-Based Evaluation is applied to examples with unit tests, covering 98.25% of our benchmark. A completion is considered correct if it passes all unit tests. We use the ExecEval framework [82] as our execution environment for this purpose.

Syntactical Match Evaluation is used where unit tests are impractical, which happens in the API function call completion split. This arises due to the potential side effects or dependencies on external environments inherent in external API function calls, which is difficult to check using only unit tests. In such instances, we use syntax matching to evaluate the model's output, comparing it against the ground truth. For instance, outputs like `func(a, b=1, c=2)` are considered equivalent to `func(a, c=2, b=1)`, focusing on syntactical equivalence rather than exact matches.

Our large dataset size of 17,720 examples enables robust evaluations without the need for multiple generations and averaging, as seen in smaller datasets like HumanEval (164 programs). Therefore, we only generate one completion for each LLM on each example and report the percentage of first-attempt passes, i.e., *Pass@1*, as our evaluation metric.

3.4 Prompts and Post-Processing

We now describe our prompt designs and post-processing techniques for the SAFIM benchmark. These aspects make huge impact in model evaluations but are often overlooked. We introduce our approach for creating prompts and our unique syntax-aware post-processing method, which refines model outputs for more accurate and fair benchmarking.

3.4.1 Prompts

LLMs' performance is heavily influenced by the design of the prompts [83, 84]. Using only a limited range of prompt types can skew evaluation results. For instance, Fried et al. [54] use the Prefix-Suffix-Middle (PSM) prompt for FIM-pretrained models and the Instructed Prefix Feeding (IPF) prompt for others, leading to direct comparisons across different prompt types. This method, however, might yield suboptimal performance for different types of LLMs, leading to inaccurate comparisons. We further discuss this in Section 3.6. We address these concerns by introducing a wider range of distinct prompts in our evaluations, as detailed in Figure 3.2:

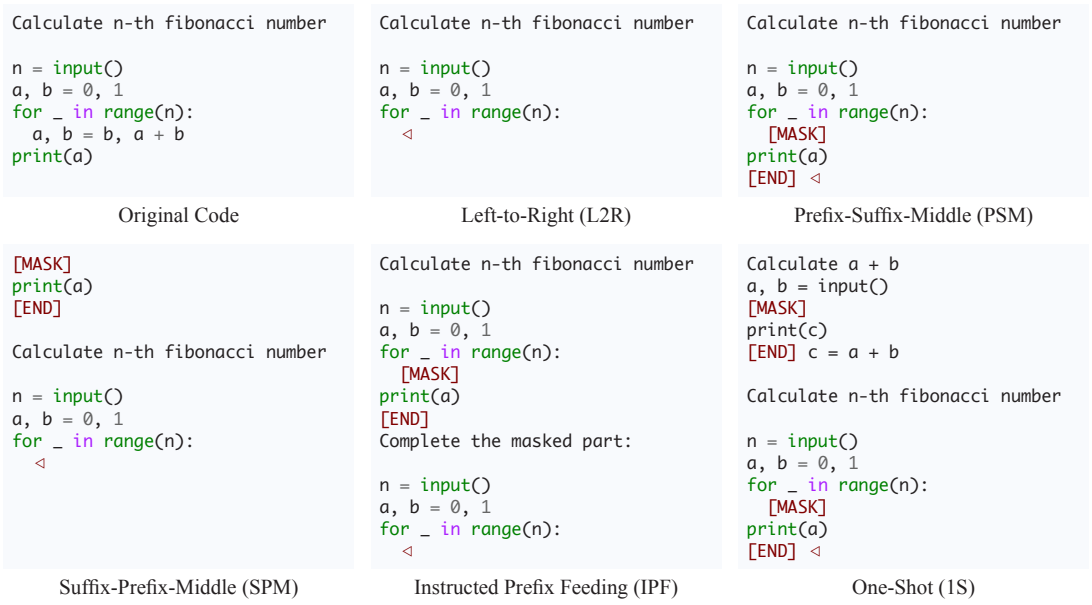


Figure 3.2: The original code is shown in the top-left, with the block `a, b = b, a + b` to be masked. The subsequent cells illustrate five distinct prompt types. The "◁" symbol indicates the end of the prompt, where model generation begins. The tokens `[MASK]` and `[END]` are model-specific, e.g., `<SUF>` and `<MID>` for CodeLLaMa, and `<|mask:0|>` and `<|mask:1|>` for InCoder.

Left-to-Right (L2R). This baseline consists of only the code's prefix and omits the suffix. It provides a foundation to assess the effectiveness of other prompt designs.

Prefix-Suffix-Middle (PSM). PSM uses a placeholder (a.k.a "sentinel token") to indicate the masked code segment, with the model tasked to generate the segment following the prompt. Effective use of this prompt type, however, requires that the model be pretrained with a FIM objective to recognize and appropriately respond to sentinel tokens.

Suffix-Prefix-Middle (SPM). SPM places the suffix at the beginning and the completion segment immediately after the prefix. This structure enables models, even those not pretrained on FIM objectives like CodeGen, to perform the completion task in a left-to-right manner. This adaptability to non-FIM pretrained makes SPM

suitable for a wider range of models, although Rozière et al. [48] reports SPM's inferior performance compared to PSM in the HumanEval-Infilling benchmark.

Instructed Prefix Feeding (IPF). IPF replaces the masked code with a placeholder, followed by an instruction, and then repeats the prefix. It allows non-FIM pretrained models to recognize and tackle completion tasks [54]. Our experiments indicate a tendency in some models to erroneously output the placeholder token as part of their output. To address this, we introduce a logits masking technique to inhibit the generation of placeholder tokens, enhancing the effectiveness of IPF.

One-Shot (1S). Tailored for non-FIM chat models, 1S uses a PSM-style prompt, supplemented with a simple input-output example, which provides the model with context about the task type and the expected input-output format.

3.4.2 Post-Processing

Post-processing is vital for automatic evaluation of LLMs in code generation, yet its importance is often underestimated. The raw output from LLMs is not immediately suitable for evaluation due to potential inclusions of irrelevant natural language or extra code beyond the targeted structure. SAFIM includes two stages of post-processing to address these challenges:

Code Extraction for Chat Models. We use regex-based heuristics to extract code from outputs of chat models like GPT-4, which often mix natural language with code in the Markdown-formatted outputs.

Truncation. An important challenge for models not fine-tuned for instruction following is their inability to determine the endpoint of their outputs. Often, such models generate the correct response but continue to produce extraneous content. A notable example is CodeGen [1], which, due to its open-ended design, lacks the capability to signal an end-of-sequence (`<eos>`), resulting in unbounded output. Therefore, truncation is essential for the effective evaluation of code generation tasks.

However, inconsistencies in truncation methods across different models have led to skewed comparisons in prior work. For example, if the expected output is a Python expression and the truncation method retains only the first line of generated code, it may erroneously dismiss correct expressions that span multiple lines, as illustrated in Figure 3.1 (Right).

Syntax-Aware Truncation. In SAFIM, we introduce a syntax-aware truncation algorithm, replacing the conventional regex-based heuristics. This approach ensures the precise extraction of targeted code structures, thereby allowing for accurate and fair evaluations across different models.

For the algorithmic block completion task, which requires a code block as output, we use an iterative truncation process on the model's output. This involves sequentially removing the last line of the output until two key conditions are met: **(a)** the truncated output must fit into the AST as a "code block" subtree; and **(b)**, the AST of the remaining code—excluding the completion segment—must align with the AST of the original code, in terms of indentation level for Python or curly brace level for C-family languages. Once both conditions are satisfied, the truncated output is considered as the model's finalized completion.

For control-flow and API function call completions, our method incrementally adds characters to the output until it satisfies similar syntax matching criteria: the completed segment must form a valid "expression" node in the AST, and the rest of the code aligns precisely with the original code's AST structure.

3.5 Experimental Setup

We evaluate GPT-3.5 [17, 46], GPT-4 [47], CodeGen [1], InCoder [54], CodeLLaMa [48], StarCoder [55], and DeepSeekCoder [57] using SAFIM. As Table 3.1 shows, these models vary in terms of parameters, data cutoff dates, open-source availability, and pretraining objectives. Given the multilingual (Python, Java, C++, and C#) nature of SAFIM, our selection prioritizes models with multilingual capabilities, and exclude Python-only variants like CodeGen-Mono and StarCoder-Python. As we focus on code sources after April 2022, SAFIM guarantees that, with the exception of CodeLLaMa and DeepSeekCoder, all models are evaluated using clean, out-of-sample test cases. In Section 3.8.9, we further discuss the impact of data contamination on our evaluation results.

For GPT-3.5 and GPT-4, we use the OpenAI API for generation. For the remaining models, generation is conducted via the Huggingface `transformers` library, following established practices in Fried et al. [54], where we use top-p random sampling with $p = 0.95$ and a temperature of 0.2. Model details for reproducibility, including the model identifiers used on OpenAI API and the Huggingface model hub, are provided in Section 3.8.3.

Table 3.1: Summary of evaluated models, highlighting data cutoff dates, open-source status (OS), and pretraining objectives. Dates in red indicate overlap between the model's pretraining data and the SAFIM benchmark in date range (post-April 2022). Data cutoff dates for InCoder are estimated based on their initial paper draft publication dates. The OS column denotes open-source availability ($\surd$ for yes, $\times$ for no), and the FIM column indicates models pretrained with FIM objectives and support for sentinel tokens in FIM inference. *For CodeLLaMa, only 7B/13B versions support FIM inference, while the 34B version does not.

	#Params	**Data Cutoff**	**OS**	**FIM**
GPT-3.5	175B	Sept 2021	$\times$	$\times$
GPT-4	-	Sept 2021	$\times$	$\times$
CodeGen	350M/2B/6B/16B	Oct 2021	$\surd$	$\times$
InCoder	1.3B/6.7B	$\leq$ Mar 2022	$\surd$	$\surd$
CodeLLaMa	7B/13B/34B	Jul 2022	$\surd$	$\surd^*$
StarCoder	15.5B	Mar 2022	$\surd$	$\surd$
DeepSeekCoder	1.3B/6.7B/33B	Feb 2023	$\surd$	$\surd$

3.6 Experimental Results

We now present the experimental results on our SAFIM benchmark, focusing on the effects of prompt designs, the efficacy of our syntax-aware truncation algorithm, and a comparative analysis of various LLMs across tasks. Given the inherent differences in model training environments and configurations, direct comparisons across different model families should be interpreted with caution. The primary value of our work is in establishing the SAFIM benchmark as a cornerstone for future experiments in this field.

3.6.1 Impact of Prompt Designs

Table 3.2 compares the effectiveness of different prompt designs by evaluating each model across various prompts with syntax-aware truncation in post-processing. This experiment reveals that:

Table 3.2: Pass@1 of each model on algorithmic block completion, evaluated with various prompts and using syntax-aware truncation for post-processing. GPT-3.5, CodeGen-16B, and CodeLLaMa-34B cannot be evaluated with the Prefix-Suffix-Middle (PSM) prompt due to lack of support for FIM sentinel tokens, as discussed in Section 3.4.1. The most effective prompt type for each model is highlighted in **bold**.

	L2R	**PSM**	**SPM**	**IPF**	**1S**
GPT-3.5 (175B)	23.2	-	30.1	28.6	**31.2**
CodeGen-16B	24.6	-	**25.9**	15.2	0.4
InCoder-6B	18.1	**25.2**	24.1	12.2	23.2
CodeLLaMa-13B	32.3	10.2	**41.4**	30.9	16.1
CodeLLaMa-34B	35.5	-	**38.5**	35.4	19.6
StarCoder (15.5B)	29.3	44.0	**44.1**	20.8	42.4
DeepSeekCoder-33B	41.6	**60.8**	57.4	33.8	59.9

Prompt Selection is Crucial for Fair Evaluation in Code FIM Tasks. A narrow selection of prompt types can lead to skewed evaluation results, as different models respond differently due to differences in their pretraining data and methods. A potentially skewed evaluation by Fried et al. [54] highlights this by comparing FIM models using the PSM prompt against non-FIM models with the IPF prompt. Doing so suggests a misleading superiority of InCoder-6B (25.2%) over CodeGen-16B (15.2%) in Pass@1 on SAFIM. This comparison, however, overlooks that CodeGen-16B achieves a higher Pass@1 of 25.9% with the SPM prompt, a prompt not included in their evaluation setup. This example shows the necessity for a comprehensive prompt range to ensure fairness. Our work addresses this by reporting the best-performing prompt for each model and includes an extensive result table in Section 3.8.4 for thorough comparison.

FIM Pretraining Boosts Both FIM and L2R Performance. Pretraining LLMs with a FIM objective enhances their performance not only in FIM but also in left-to-right (L2R) generation. The advantage in FIM evaluation is highlighted by the results of CodeLLaMa models: the larger CodeLLaMa-34B, without FIM pretraining, is outperformed by the smaller, FIM+L2R pre-trained CodeLLaMa-13B. A more interesting observation emerges in the "L2R" column of Table 3.2: FIM-pretrained models like StarCoder outperform purely L2R-pretrained models like CodeGen-16B

Table 3.3: Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase. This table presents two numbers for each model evaluated on algorithmic block completion tasks: **Pass@1** and **CErr%** (the percentage of unexecutable programs due to compile or syntax errors in the generated completions).

	No Trunc.		**Syntax Trunc.**	
	Pass@1	**CErr%**	**Pass@1**	**CErr%**
GPT-3.5 (175B)	28.7	25.3	31.2	17.0
GPT-4 (> 220B)	41.7	25.4	42.1	22.9
CodeGen-16B	0.0	99.9	25.9	17.9
InCoder-6B	21.8	25.7	25.2	13.2
CodeLLaMa-13B	16.4	64.6	41.4	10.9
CodeLLaMa-34B	1.0	94.5	38.5	14.7
StarCoder (15.5B)	42.7	14.3	44.1	9.5
DeepSeekCoder-33B	59.7	8.0	60.8	4.0

in L2R mode, despite similar sizes. This finding suggests that FIM pretraining does not harm, and actually enhances, a model's L2R performance, possibly by fostering a better understanding of code via contextually rich pretraining inputs. This supports similar improvements observed in FIM-pretrained GPT-3/Codex models in prior studies [50], and offer strong justification for the recent shift from pure L2R pretraining to FIM pretraining among code LLM developers [55, 57, 48].

3.6.2 Impact of Our Syntax-Aware Truncation

We assess the impact of our syntax-aware truncation algorithm through an ablation study, measuring model performance on the algorithmic block completion task with and without syntax-aware truncation. This analysis focuses on two key numbers: Pass@1 and the percentage of unexecutable programs due to compile or syntax errors in the generated completions. We treat empty outputs after truncation, typically indicative of a failure to identify any valid executable, as compilation errors. The results are shown in Table 3.3. These results show that:

Syntax-Aware Truncation Enhances FIM Output Quality. Table 3.3 shows that our syntax-aware truncation algorithm not only enhances the Pass@1 rates but also significantly reduces compilation errors across various models. This indicates a consistent improvement in the quality of FIM outputs, achieved without additional GPU overhead during model inference. We believe syntax-aware truncation holds promise for real-world code completion applications.

Syntax-Aware Truncation Enables Fair Comparison for Non-FIM Models. As shown in Table 3.3, syntax-aware truncation benefits non-FIM models much more than FIM models. For example, CodeLLaMa-13B's Pass@1 rate jumps from 16.4% to 41.4% with truncation, changing its comparative performance against InCoder-6B, whose Pass@1 only increases marginally from 21.8% to 25.2%. This discrepancy stems from their distinct training approaches. InCoder, exclusively trained on FIM, naturally aligns with FIM-style prompts. In contrast, CodeLLaMa-13B, with a primary focus on L2R in its mixed FIM+L2R training, often produces unwanted extra code after completion. The extra code, while removable by syntax-aware truncation, obscures CodeLLaMa-13B's true effectiveness when such truncation is not applied. By precisely eliminating the extra code, syntax-aware truncation unveils the true coding proficiency of non-FIM or hybrid models like CodeLLaMa, ensuring fair comparisons with FIM-focused models. Additionally, syntax-aware truncation allows open-ended models to be evaluated in FIM tasks.

3.6.3 Comparative Performance Analysis of LLMs

After determining the most effective prompt for each model and verifying the benefits of syntax-aware truncation, we conduct comprehensive evaluations across the entire SAFIM benchmark. Table 3.4 shows model performances in each task category, and Figure 3.3 visualizes the average performance of models against their model sizes. These results offers insights into the capabilities and limitations of code LLMs:

Pretraining Method and Data Are More Important Than Sheer Model Size. Smaller models with sophisticated pretraining paradigms can match or even outperform larger counterparts. For example, StarCoder, with 15.5B parameters, achieves an average Pass@1 of 55.5%, comparable to GPT-4's 53.3%, despite GPT-4's vast size. This pattern recurs in models like CodeLLaMa-13B and DeepSeekCoder-1.3B. Notably, the comparison between StarCoder and GPT-4 is not subject to data contamination, as discussed in Table 3.1. This finding challenges the common belief that larger models automatically yield superior performance, even with basic

Table 3.4: Pass@1 of various models on the SAFIM benchmark, showing their performance in algorithmic block completion (Algo.), control-flow completion (Control), and API function call completion (API). The table also reports the average performance, indicating each model's overall effectiveness on SAFIM.

	Algo.	**Control**	**API**	**Avg**
GPT-3.5 (175B)	31.2	37.5	53.9	40.9
GPT-4 (> 220B)	42.1	55.2	62.6	53.3
CodeGen-350M	16.3	26.1	26.5	22.9
CodeGen-2B	23.5	32.9	32.3	29.5
CodeGen-6B	23.6	34.8	27.7	28.7
CodeGen-16B	25.9	35.7	31.3	31.0
InCoder-1B	21.1	22.9	43.9	29.3
InCoder-6B	25.2	28.2	48.1	33.8
CodeLLaMa-7B	34.7	53.6	46.8	45.0
CodeLLaMa-13B	41.4	57.2	59.7	52.8
CodeLLaMa-34B	38.5	54.0	56.5	49.7
StarCoder (15.5B)	44.1	54.5	68.1	55.5
DeepSeekCoder-1.3B	41.2	54.1	62.6	52.6
DeepSeekCoder-6.7B	54.7	65.8	69.7	63.4
DeepSeekCoder-33B	**60.8**	**71.1**	**75.2**	**69.0**

pretraining methods [17]. Our study suggests that this may not hold true for coding tasks: within the same model family, performance gains from increased size are only modest, while models from different families exhibit substantial performance variations. For example, the weakest CodeLLaMa model surpasses the strongest CodeGen model by 14 points, a far more significant margin than the 7.8-point spread within CodeLLaMa models.

Pretraining Method and Data Influence Task-Specific Performance. We have discussed in Section 3.6.1 that FIM pretraining enhances performance on both FIM evaluation and L2R completion. Dissecting model performance across SAFIM's three splits sheds further light on this impact:

- For API function call completion, repository-level information is key. StarCoder

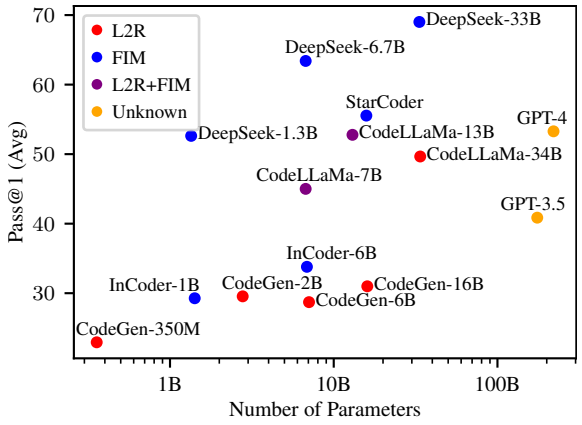


Figure 3.3: Average performance of different models relative to their sizes on the SAFIM benchmark. Each model is represented by a dot, with the x-axis showing model size (number of parameters) and the y-axis showing average performance across three task categories. Dot colors signify pretraining paradigms: red for Left-to-Right (L2R), blue for FIM, purple for a combination of L2R and FIM, and orange for proprietary models with undisclosed pretraining methods.

and DeepSeekCoder, which excel in this task, both incorporate repository context into their pretraining data. StarCoder enriches its training input with GitHub issues and commit messages, while DeepSeekCoder organize code files according to their topological ordering based on API dependencies. These techniques significantly enhance their ability to understand API contexts.

- For control-flow completion, CodeLLaMa's relatively strong performance is attributed to its use of execution-based feedback in its self-instruct training method. By executing generated code and applying the results as rewards or penalties, CodeLLaMa learns to avoid generating unexecutable code or infinite loops, thereby gaining a more refined understanding of control flows.

These findings highlight the pivotal role of the pretraining paradigm in the performance of LLMs on coding tasks.

3.7 Conclusion and Future Work

We introduced the Syntax-Aware Fill-in-the-Middle (SAFIM) benchmark, the first large-scale, multilingual Fill-in-the-Middle (FIM) benchmark equipped with executable unit tests for evaluating code-centric Large Language Models (LLMs). To mitigate data contamination, SAFIM adopts a strict cutoff date for code sources. Moreover, SAFIM uniquely categorizes tasks into three syntax-driven splits: algorithmic block completion, control-flow expression completion, and API function call completion. These splits provide a comprehensive assessment of LLMs' coding capabilities across multiple dimensions. SAFIM's suite of prompts and its novel syntax-aware truncation algorithm for post-processing enable fair comparisons among various types of models, including those not explicitly pretrained on FIM tasks.

The results of our large-scale evaluation highlight the significant impact of pretraining paradigms on LLMs' performance, emphasizing the importance of training method and data quality over sheer model size. We found that FIM pretraining can enhance, rather than harm, Left-to-Right (L2R) inference capabilities, supporting a shift towards FIM as a primary pretraining objective for code LLMs. We acknowledge a key limitation in our study: our conclusions are drawn from comparisons across various model families trained with different paradigms, rather than from controlled experiments altering pretraining paradigms within the same model. Yet, SAFIM establishes a foundational framework for future research into pretraining paradigms and the development of better LLMs for coding tasks.

3.8 Appendix

3.8.1 Details about the API Function Call Completion Task

We consider the following API libraries for each programming language when we construct the API function call completion split of SAFIM:

- **Python:** NumPy, Pandas, Statsmodels, Sci-kit Learn, Matplotlib, NLTK, Gensim, XGBoost, PyTorch, Huggingface Transformers
- **Java:** GSON, Caffeine, Apache Commons, Google HTTP Client, Joda-Time, JavaParser,
- **C++:** GMP, Boost, JSON, QT, Eigen, OpenGL, Tree-Sitter
- **C#:** Newtonsoft.Json, SignalR, RestSharp, LiteDB, BCrypt.Net

Table 3.5: Statistics of each task category of the SAFIM benchmark, including number of examples, total uncompressed disk size of code contexts, average length of code contexts in bytes, and average length of ground truth completions in bytes.

	Source	# Examples	Disk Size	Avg Code Len	Avg Completion Len
Algorithmic Block	Codeforces	8,781	29.4M	3346B	67B
Control-Flow	Codeforces	8,629	29.5M	3415B	16B
API Function Call	GitHub	310	713K	2302B	40B
Total	-	17,720	59.6M	3364B	42B

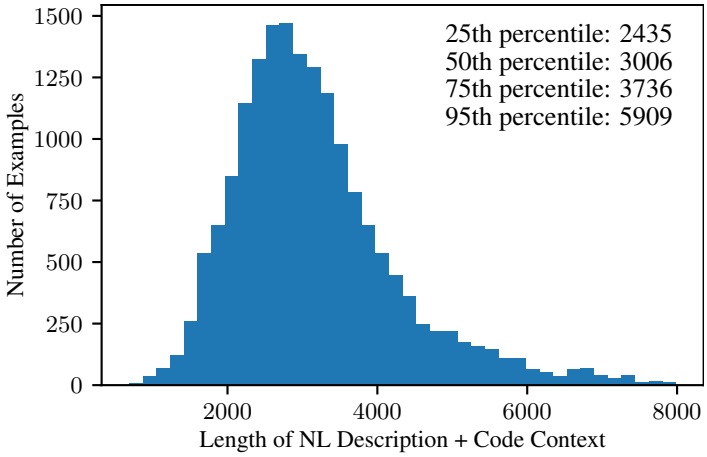


Figure 3.4: Histogram of the total number of **characters** of the natural language problem description and the code context. 424 example longer than 8,000 characters are excluded from this histogram for clarity but counted towards the displayed quantiles.

3.8.2 Statistics of the SAFIM Benchmark

Statistics of each split of the SAFIM benchmark is presented in Table 3.5.

Figure 3.4 shows the distribution the total lengths of problem descriptions plus code contexts of examples measured in characters. A majority of the dataset has less than 6k characters. On average, one BPE token corresponds to 3 to 4 characters in the code domain. This ensures that the evaluated models, all with context windows of at

Table 3.6: Statistics of examples in each programming language of the SAFIM benchmark, including number of examples, total uncompressed disk size of code contexts, average length of code contexts in bytes, average length of ground truth completions in bytes, and average length of identifiers in bytes. The *identifiers* refer to the names of variables, functions, and classes.

	# Examples	Disk Size	Avg Code Len	Avg Completion Len	Avg Identifier Len
Python	9,901	30.0M	3026B	44B	2.73B
Java	4,999	17.3M	3454B	44B	4.14B
C++	1,736	5.14M	2962B	27B	3.65B
C#	1,084	7.24M	6675B	42B	5.79B
Total	17,720	59.6M	3364B	42B	3.59B

least 2,048, accurately reflect performance without bias from input size.

Table 3.6 shows statistics per programming language of examples in SAFIM. This distribution of PLs in SAFIM mirrors the prevalence in our source corpus, especially in Codeforces where most contestants use C++. Table 3.6 also highlights variations in coding style across languages. For instance, C++ and Python programmers favor succinct coding (less code, shorter variable names), while Java and C# users lean towards verbosity. Subsequent sections will discuss how these differences in coding style influence evaluation results.

3.8.3 Details of Model Implementations

Table 3.1 shows the implementations used for evaluating each LLM, including additional models we will discuss in Section 3.8.6. For GPT-3.5 and GPT-4, we use the OpenAI API[3] for generation. For the remaining models, generation is conducted via the Huggingface `transformers` library[4].

3.8.4 Results of All Models on All Prompts

Table 3.8, Table 3.9, and Table 3.10 show experimental results of all models using all types of prompts, where each table shows the results on one task category of SAFIM.

[3] `https://openai.com/blog/openai-api`

[4] `https://github.com/huggingface/transformers`

Table 3.7: The code enviroment for evaluating each LLM and the model identifier on its respective platform.

	Codebase	**Model Identifier**
GPT-3.5	OpenAI API	`gpt-3.5-turbo-0301`
GPT-4	OpenAI API	`gpt-4-1106-preview`
CodeGen-350M	Huggingface Transformers	`Salesforce/codegen-350M-multi`
CodeGen-2B	Huggingface Transformers	`Salesforce/codegen-2B-multi`
CodeGen-6B	Huggingface Transformers	`Salesforce/codegen-6B-multi`
CodeGen-16B	Huggingface Transformers	`Salesforce/codegen-16B-multi`
InCoder-1B	Huggingface Transformers	`facebook/incoder-1B`
InCoder-6B	Huggingface Transformers	`facebook/incoder-6B`
CodeLLaMa-7B	Huggingface Transformers	`codellama/CodeLlama-7b-hf`
CodeLLaMa-13B	Huggingface Transformers	`codellama/CodeLlama-13b-hf`
CodeLLaMa-34B	Huggingface Transformers	`codellama/CodeLlama-34b-hf`
StarCoder (15.5B)	Huggingface Transformers	`bigcode/starcoderbase`
DeepSeekCoder-1.3B	Huggingface Transformers	`deepseek-ai/deepseek-coder-1.3b-base`
DeepSeekCoder-6.7B	Huggingface Transformers	`deepseek-ai/deepseek-coder-6.7b-base`
DeepSeekCoder-33B	Huggingface Transformers	`deepseek-ai/deepseek-coder-33b-base`
Mixtral-8x7B	Huggingface Transformers	`mistralai/Mixtral-8x7B-v0.1`
Phi-1.5 (1.3B)	Huggingface Transformers	`microsoft/phi-1_5`
Phi-2 (2.7B)	Huggingface Transformers	`microsoft/phi-2`
WizardCoder-1B	Huggingface Transformers	`WizardLM/WizardCoder-1B-V1.0`
WizardCoder-3B	Huggingface Transformers	`WizardLM/WizardCoder-3B-V1.0`
WizardCoder-15B	Huggingface Transformers	`WizardLM/WizardCoder-15B-V1.0`
WizardCoder-33B	Huggingface Transformers	`WizardLM/WizardCoder-33B-V1.1`
Magicoder-S-DS-6.7B	Huggingface Transformers	`ise-uiuc/Magicoder-S-DS-6.7B`

3.8.5 Further Results about Syntax-Aware Truncation

Section 3.6.2 explores the benefits of syntax-aware truncation with algorithmic block completion tasks. This section extends the results in Table 3.3 to encompass all tasks in SAFIM. Additionally, we also show the selected prompt for each model, determined by the highest pass@1 rate post-truncation. The results are shown in Table 3.11, Table 3.12, and Table 3.13.

We find that syntax-aware truncation consistently improves the pass@1 rate and reduces compilation errors in both algorithmic block completion and control-flow expression completion, highlighting the effectiveness of syntax-aware truncation.

However, in API function call completion, which involves generation of function

Table 3.8: The performance of each model with each type of prompts on algorithmic block completion. Syntax-aware truncation is used for post-processing. The most effective prompt type for each model is highlighted in **bold**.

	L2R	PSM	SPM	IPF	1S
GPT-3.5 (175B)	23.2	-	30.1	28.6	**31.2**
GPT-4	-	-	-	-	**42.1**
CodeGen-350M	15.4	-	**16.3**	6.8	0.1
CodeGen-2B	22.5	-	**23.5**	13.9	0.0
CodeGen-6B	23.2	-	**23.6**	14.6	0.0
CodeGen-16B	24.6	-	**25.9**	15.2	0.4
InCoder-1B	14.1	**21.1**	19.2	9.0	17.6
InCoder-6B	18.1	**25.2**	24.1	12.2	23.2
CodeLLaMa-7B	30.7	8.8	**34.7**	24.4	7.5
CodeLLaMa-13B	32.3	10.2	**41.4**	30.9	16.1
CodeLLaMa-34B	35.5	-	**38.5**	35.4	19.6
StarCoder (15.5B)	29.3	44.0	**44.1**	20.8	42.4
DeepSeekCoder-1.3B	28.0	**41.2**	38.7	6.5	38.0
DeepSeekCoder-6.7B	36.2	**54.7**	51.3	27.1	52.9
DeepSeekCoder-33B	41.6	**60.8**	57.4	33.8	59.9

invocation expressions or statements, LLMs typically produce error-free code without truncation, as these code segments are typically short and naturally segmented with line breaks. That said, syntax-aware truncation becomes crucial for models and prompts lacking explicit stop signals, such as SPM for CodeGen and IPF for CodeLLaMa-34B. In these scenarios, our truncation method allows fair comparisons across various models by standardizing the completion endpoint.

3.8.6 Evaluation Results of More LLMs

In this section, we expand our evaluation to include additional LLMs: Mixtral [85], Phi [86], WizardCoder [87], and Magicoder [88].

Table 3.14 provides the details of the additional models. Mixtral-8x7B, a sparse mixture-of-experts (MoE) model, uses a router to select two expert 7B models for each

Table 3.9: The performance of each model with each type of prompts on control-flow completion. Syntax-aware truncation is used for post-processing. The most effective prompt type for each model is highlighted in **bold**.

	L2R	PSM	SPM	IPF	1S
GPT-3.5 (175B)	-	-	-	-	**37.5**
GPT-4	-	-	-	-	**55.2**
CodeGen-350M	25.0	-	**26.1**	17.6	-
CodeGen-2B	32.4	-	**32.9**	25.1	-
CodeGen-6B	33.1	-	**34.8**	25.9	-
CodeGen-16B	34.7	-	**35.7**	27.9	-
InCoder-1B	19.6	22.9	**24.4**	11.5	-
InCoder-6B	23.6	28.2	**29.0**	14.9	-
CodeLLaMa-7B	43.1	25.8	**53.6**	40.6	-
CodeLLaMa-13B	45.1	27.3	**57.2**	46.2	-
CodeLLaMa-34B	48.0	-	**54.0**	51.5	-
StarCoder (15.5B)	43.4	**54.5**	53.7	37.4	-
DeepSeekCoder-1.3B	42.6	**54.1**	52.5	35.1	-
DeepSeekCoder-6.7B	50.4	**65.8**	63.8	51.4	-
DeepSeekCoder-33B	55.7	**71.1**	69.8	58.6	-

inference. The Phi models are small LLMs pretrained using distilled data (synthetic data generated by a teacher LLM). WizardCoder and Magicoder are initialized with base models and then finetuned on distilled data. Specifically, WizardCoder variants (15B and 33B) use StarCoder and DeepSeekCoder-33B as their respective base models, while Magicoder-S-DS-6.7B is finetuned from DeepSeekCoder-6.7B. Notably, these models are classified as FIM models because their inherited vocabulary supports FIM special tokens, despite the finetuning process not directly engaging with FIM tasks.

Table 3.15 shows our experimental results, which yield the following insights:

- Mixtral-8x7B achieves performance comparable to CodeLLaMa-7B. Given that Mixtral is not specialized for coding, its comparable performance to 7B code LLMs shows the effectiveness of MoE. Typically, general-purpose LLMs like GPT-3.5, GPT-4, and Mixtral need more parameters to match the performance

Table 3.10: The performance of each model with each type of prompts on API function call completion. Syntax-aware truncation is used for post-processing. The most effective prompt type for each model is highlighted in **bold**.

	L2R	PSM	SPM	IPF	1S
GPT-3.5 (175B)	-	-	-	-	**53.9**
GPT-4	-	-	-	-	**62.6**
CodeGen-350M	23.5	-	**26.5**	9.7	-
CodeGen-2B	30.3	-	**32.3**	10.3	-
CodeGen-6B	25.5	-	**27.7**	13.5	-
CodeGen-16B	**31.3**	-	31.3	16.8	-
InCoder-1B	38.4	**43.9**	43.9	13.5	-
InCoder-6B	41.0	**48.1**	47.1	16.5	-
CodeLLaMa-7B	**48.7**	37.1	46.8	21.6	-
CodeLLaMa-13B	50.3	39.0	**59.7**	39.0	-
CodeLLaMa-34B	50.6	-	47.7	**56.5**	-
StarCoder (15.5B)	50.6	**68.1**	65.2	44.5	-
DeepSeekCoder-1.3B	45.8	**62.6**	51.9	11.9	-
DeepSeekCoder-6.7B	52.3	**69.7**	60.0	52.3	-
DeepSeekCoder-33B	45.5	**75.2**	64.5	50.6	-

of specialized code LLMs.

- Models Pretrained on Distilled Data (Phi-1.5 and Phi-2) exhibit good performance considering their tiny sizes, but they don't reach the high standards set by their HumanEval results. This difference underscores the SAFIM benchmark's diversity and complexity compared to HumanEval.

- Models Finetuned on Distilled Data shows a slight drop in performance compared to their FIM-pretrained base models (e.g., WizardCoder-15B vs. StarCoder, WizardCoder-33B vs. DeepSeekCoder-33B, Magicoder-S-DS-6.7B vs. DeepSeekCoder-6.7B). The performance drop stems from the left-to-right finetuning on distilled data, which lacks FIM objectives, thereby harming the models' proficiency in FIM tasks.

Table 3.11: Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase. This table presents two numbers for each model evaluated: **Pass@1** and **CErr%**, as well as the prompt selected to evaluate each model.

	No Trunc.		Syntax Trunc.		
	Pass@1	**CErr%**	**Pass@1**	**CErr%**	**Prompt**
GPT-3.5 (175B)	28.7	25.3	31.2	17.0	1S
GPT-4 (> 220B)	41.7	25.4	42.1	22.9	1S
CodeGen-16B	0.0	99.9	25.9	17.9	SPM
InCoder-6B	21.8	25.7	25.2	13.2	PSM
CodeLLaMa-13B	16.4	64.6	41.4	10.9	SPM
CodeLLaMa-34B	1.0	94.5	38.5	14.7	SPM
StarCoder (15.5B)	42.7	14.3	44.1	9.5	SPM
DeepSeekCoder-33B	59.7	8.0	60.8	4.0	PSM

Table 3.12: Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase on control-flow expression completion. This table presents two numbers for each model evaluated: **Pass@1** and **CErr%**, as well as the prompt selected to evaluate each model.

	No Trunc.		Syntax Trunc.		
	Pass@1	**CErr%**	**Pass@1**	**CErr%**	**Prompt**
GPT-3.5 (175B)	37.4	19.7	37.5	19.5	1S
GPT-4 (> 220B)	55.2	21.8	55.2	21.9	1S
CodeGen-16B	0.0	99.9	35.7	14.6	SPM
InCoder-6B	10.4	62.1	28.2	11.0	PSM
CodeLLaMa-13B	27.8	54.8	57.2	2.3	SPM
CodeLLaMa-34B	0.3	98.6	54.0	2.7	SPM
StarCoder (15.5B)	51.8	9.8	54.5	6.0	PSM
DeepSeekCoder-33B	70.3	2.8	71.1	1.1	PSM

Table 3.13: Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase on API function call completion. This table presents two numbers for each model evaluated: **Pass@1** and **CErr%**, as well as the prompt selected to evaluate each model.

	No Trunc.		**Syntax Trunc.**		
	Pass@1	**CErr%**	**Pass@1**	**CErr%**	**Prompt**
GPT-3.5 (175B)	44.2	0.0	53.9	0.0	1S
GPT-4 (> 220B)	57.4	0.0	62.6	0.0	1S
CodeGen-16B	0.0	99.9	31.3	1.9	SPM
InCoder-6B	23.9	0.0	48.1	0.0	PSM
CodeLLaMa-13B	33.5	0.0	59.7	0.0	SPM
CodeLLaMa-34B	11.9	0.0	56.5	0.6	IPF
StarCoder (15.5B)	65.8	0.3	68.1	0.3	PSM
DeepSeekCoder-33B	72.3	0.0	75.2	0.0	PSM

These additional findings further reinforce our original conclusion: the pretraining methodology significantly influences the performance of code LLMs.

3.8.7 Result Analysis by Programming Languages

Table 3.16 shows the average pass@1 rate for each LLM in our SAFIM benchmark, broken down by programming language and averaged on three completion tasks. Our analysis reveals that:

LLMs exhibit higher success rates in Java and C#, likely due to the verbosity of these languages, which leads to more predictable coding patterns. Conversely, completion in C++ and Python is more challenging, due to the more concise and less predictable coding styles prevalent among developers. As we discussed in Section 3.8.2, the SAFIM benchmark consist of different programming languages written by different developers, so the results are affected by intrinsic variability in coding styles across PLs.

Table 3.14: Summary of evaluated models, highlighting data cutoff dates, open-source status (OS), and pretraining objectives. Dates in red indicate overlap between the model's pretraining data and the SAFIM benchmark in date range (post-April 2022). Data cutoff dates for InCoder are estimated based on their initial paper draft publication dates. The OS column denotes open-source availability ($\surd$ for yes, $\times$ for no), and the FIM column indicates models pretrained with FIM objectives and support for sentinel tokens in FIM inference.

	#Params	**Data Cutoff**	**OS**	**FIM**
GPT-4	-	Sept 2021	$\times$	$\times$
CodeGen	350M/2B/6B/16B	Oct 2021	$\surd$	$\times$
InCoder	1.3B/6.7B	$\leq$ Mar 2022	$\surd$	$\surd$
CodeLLaMa	7B/13B	Jul 2022	$\surd$	$\surd$
CodeLLaMa	34B	Jul 2022	$\surd$	$\times$
StarCoder	15.5B	Mar 2022	$\surd$	$\surd$
DeepSeekCoder	1.3B/6.7B/33B	Feb 2023	$\surd$	$\surd$
Mixtral	46.7B (8x7B)	$\leq$ Sep 2023	$\surd$	$\times$
Phi	1.3B/2.7B	Mar 2022	$\surd$	$\times$
WizardCoder	1B/3B/15B	Mar 2022	$\surd$	$\surd$
WizardCoder	33B	Feb 2023	$\surd$	$\surd$
Magicoder	6.7B	Feb 2023	$\surd$	$\surd$

Despite the language-dependent variability, the relative rankings of LLMs stay mostly consistent. This underscores the robustness of the SAFIM benchmark and supports our decision to report *micro-averaged* performance metrics in our study.

3.8.8 Case Study

This section presents a case study of the algorithmic block completion task from SAFIM (`task_id: block_completion_008121`). Two similar-performing models, InCoder-6B and CodeCen-16B (achieving pass rates at 25.2% and 25.9% respectively), are compared.

The case originates from problem 1678B1 on Codeforces[5]; given the problem de-

[5] `https://codeforces.com/problemset/problem/1678/B1`

Table 3.15: Pass@1 of various models on the SAFIM benchmark, showing their performance in algorithmic block completion (Algo.), control-flow completion (Control), and API function call completion (API).

	Algo.	**Control**	**API**	**Avg**
GPT-3.5 (175B)	31.2	37.5	53.9	40.9
GPT-4 (> 220B)	42.1	55.2	62.6	53.3
CodeGen-350M	16.3	26.1	26.5	22.9
CodeGen-2B	23.5	32.9	32.3	29.5
CodeGen-6B	23.6	34.8	27.7	28.7
CodeGen-16B	25.9	35.7	31.3	31.0
InCoder-1B	21.1	22.9	43.9	29.3
InCoder-6B	25.2	28.2	48.1	33.8
CodeLLaMa-7B	34.7	53.6	46.8	45.0
CodeLLaMa-13B	41.4	57.2	59.7	52.8
CodeLLaMa-34B	38.5	54.0	56.5	49.7
StarCoder (15.5B)	44.1	54.5	68.1	55.5
DeepSeekCoder-1.3B	41.2	54.1	62.6	52.6
DeepSeekCoder-6.7B	54.7	65.8	69.7	63.4
DeepSeekCoder-33B	**60.8**	**71.1**	**75.2**	**69.0**
Mixtral-8x7B	33.7	50.3	58.4	47.5
Phi-1.5 (1.3B)	19.0	29.9	27.7	25.5
Phi-2 (2.7B)	23.8	34.8	22.3	26.9
WizardCoder-1B	28.1	40.0	57.4	41.8
WizardCoder-3B	34.4	46.3	65.2	48.6
WizardCoder-15B	41.0	52.6	71.0	54.8
WizardCoder-33B	49.5	66.3	74.5	63.4
Magicoder-S-DS-6.7B	41.5	62.3	65.5	56.4

Table 3.16: Average pass@1 of various models on the three tasks in SAFIM, showing their results in different programming languages.

	C++	**Java**	**Python**	**C#**	**Avg**
GPT-3.5 (175B)	39.3	54.2	29.5	40.5	40.9
GPT-4 (> 220B)	49.4	63.3	42.7	54.6	53.3
CodeGen-350M	23.1	33.6	18.7	19.9	22.9
CodeGen-2B	27.9	43.4	24.1	28.9	29.5
CodeGen-6B	30.3	44.6	21.2	26.4	28.7
CodeGen-16B	35.5	46.5	20.7	30.2	31.0
InCoder-1B	21.3	35.9	35.3	32.2	29.3
InCoder-6B	26.2	41.4	40.5	32.4	33.8
CodeLLaMa-7B	33.6	56.1	40.6	47.9	45.0
CodeLLaMa-13B	45.8	60.2	52.5	64.7	52.8
CodeLLaMa-34B	43.3	59.9	49.0	62.3	49.7
StarCoder (15.5B)	52.0	63.9	59.5	54.7	55.5
DeepSeekCoder-1.3B	44.7	61.3	57.7	55.5	52.6
DeepSeekCoder-6.7B	57.6	70.3	67.5	70.4	63.4
DeepSeekCoder-33B	65.8	75.1	72.5	74.7	69.0
Mixtral-8x7B	42.0	58.0	43.0	56.9	47.5
Phi-1.5 (1.3B)	18.8	30.1	26.7	18.9	25.5
Phi-2 (2.7B)	22.6	36.3	23.8	23.3	26.9
WizardCoder-1B	34.1	48.7	44.5	39.3	41.8
WizardCoder-3B	43.3	57.0	53.8	49.8	48.6
WizardCoder-15B	51.7	61.7	59.2	52.1	54.8
WizardCoder-33B	61.1	70.8	67.2	55.0	63.4
Magicoder-S-DS-6.7B	50.8	65.4	57.3	61.4	56.4

scription, the task is to fill in the `# TODO: Your code here` part of the provided code:

```
t=int(input(""))
for z in range(t):
    n=int(input(""))
    a=input("")
    s=[]
    for i in range(0,len(a)-1,2):
        # TODO: Your code here
    b=s.count('10')
    c=s.count('01')
    print(b+c)
```

The ground truth involves appending two characters from the string `a` to the list `s` in each loop iteration (we added 8 spaces at the beginning for clarity):

```
        ab=a[i]+a[i+1]
        s.append(ab)
```

Using the PSM prompt, InCoder-6B successfully generates a valid Python completion using Python string slicing:

```
        s.append(a[i:i+2])
```

CodeGen-16B uses the SPM prompt. Note that CodeGen-16B lacks EOS token support, leading to generation of extra code followed by infinite output generation unless truncated:

```
        s.append(a[i])
    s.append(a[len(a)-1])
    s=s[::-1]
    b=s.count('10')
    c=s.count('01')
    print(b+c)

# [infinite empty lines]
```

Table 3.17: Pass@1 of each model on two versions of algorithmic block completion, including the **original** version (Apr 2022 - Jan 2023) and the **new** version (Apr 2023 - Jan 2024). Numbers in red indicate overlap between the model's pretraining data and the test dataset in date range. The Δ column shows the pass@1 change between the original and the new test datasets.

	Data Cutoff	**Original**	**New**	**Δ**
StarCoder	Mar 2022	44.1	46.7	+2.56
CodeLLaMa-7B	Jul 2022	34.7	32.7	-1.95
CodeLLaMa-13B	Jul 2022	41.4	45.8	+4.40
CodeLLaMa-34B	Jul 2022	38.5	43.8	+5.29
DeepSeekCoder-1.3B	Feb 2023	41.2	46.1	+4.87
DeepSeekCoder-6.7B	Feb 2023	54.7	58.4	+3.65
DeepSeekCoder-33B	Feb 2023	60.8	61.7	+0.91

Applying syntax-aware truncation, we keep only the relevant block completion `s.append(a[i])`. Unfortunately, this still yields an incorrect solution, leading to an outcome of "wrong answer" in evaluation.

This case study underscores the significance of syntax-aware truncation and highlights the behavior of different models.

3.8.9 Further Analysis on Data Contamination

SAFIM is sourced from Codeforces contests and Github code commits created between April 1, 2022 and January 1, 2023. This period, unfortunately, overlaps with the pretraining data of CodeLLaMa and DeepSeekCoder. To analyze the potential influence of data contamination on our evaluation results, we create an new dataset for the algorithmic block completion task based on Codeforces contests from April 1, 2023, to January 31, 2024, without any overlap with the training data of these models. Then we evaluate each of these models and StarCoder, on both datasets. The findings are shown in Table 3.17. We also visualize each model's performance across various months in the new test dataset in Figure 3.5.

Based on Table 3.17, for the new test data, without any overlap with the models' training date ranges, no significant performance decrease is noticed compared to the original dataset, which had a date range overlap. Figure 3.5 also shows stable

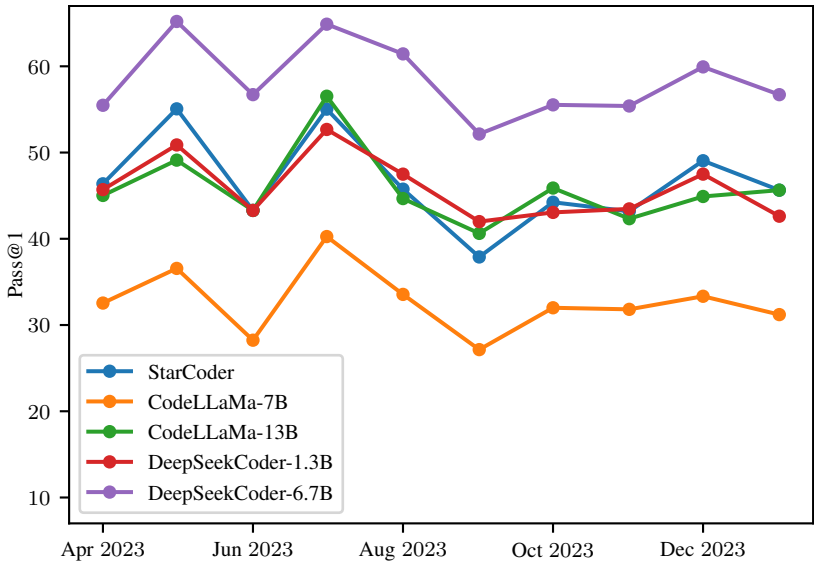


Figure 3.5: Pass@1 scores for each model on algorithmic block completion across various months in the **new** test dataset.

performance across the timeline for all models, without a noticeable decline on newer questions for CodeLLaMa or DeepSeekCoder. These findings suggest that while vigilance against data contamination is prudent, the difference in cutoff dates has a negligible impact on our current evaluation results.

Chapter 4

AST-T5: Structure-Aware Pretraining for Code Generation and Understanding

Large language models (LLMs) have made significant advancements in code-related tasks, yet many LLMs treat code as simple sequences, neglecting its structured nature. We introduce AST-T5, a novel pretraining paradigm that leverages the Abstract Syntax Tree (AST) for enhanced code generation, transpilation, and understanding. Using dynamic programming, our AST-Aware Segmentation retains code structure, while our AST-Aware Span Corruption objective equips the model to reconstruct various code structures. Unlike other models, AST-T5 avoids complex program analyses or architectural changes, so it integrates seamlessly with any encoder-decoder Transformer. Evaluations show that AST-T5 consistently outperforms similar-sized LMs across various code-related tasks including HumanEval and MBPP. Structure-awareness makes AST-T5 particularly powerful in code-to-code tasks, surpassing CodeT5 by 2 points in exact match score for the Bugs2Fix task and by 3 points in exact match score for Java-C# Transpilation in CodeXGLUE. Our code and model are publicly available at `https://github.com/gonglinyuan/ast_t5`

4.1 Introduction

We have witnessed the transformative impact of large language models (LLMs) on various aspects of artificial intelligence in recent years [17, 46, 89], especially in code generation and understanding [13, 14, 48]. By pretraining on massive code corpora such as the GitHub corpus, LLMs learns rich representations, thereby becoming

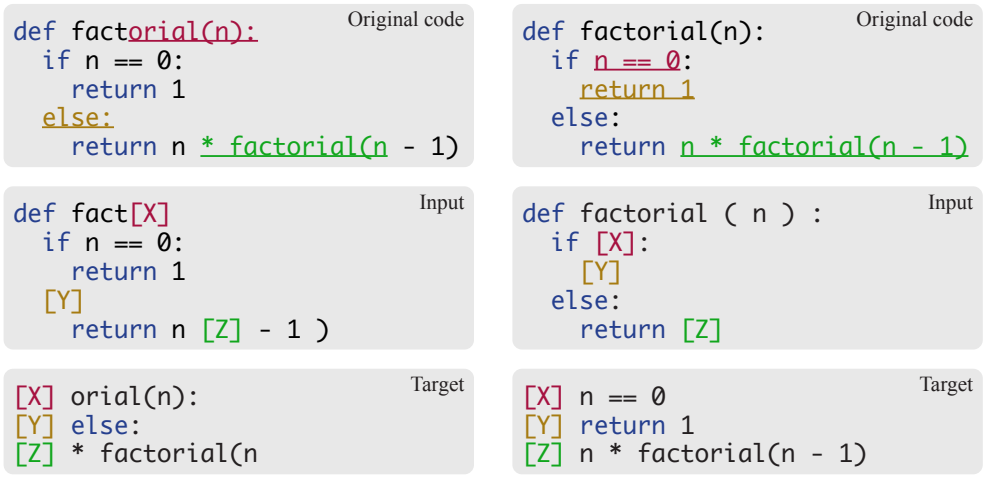


Figure 4.1: Comparison of AST-Aware Subtree Corruption and Vanilla T5 using a Python factorial function. Both methods replace masked spans with sentinel tokens (special tokens added to the vocabulary, shown as [X], [Y], and [Z] in the figure), with output sequences containing the original masked tokens. Inputs and targets are shown in byte-pair encoding (BPE); for instance, "factorial" is encoded into "fact" and "orial". Unlike Vanilla T5, which masks random spans without considering code structure, our approach specifically targets spans aligned with AST subtrees, like expressions and statements.

powerful tools for various downstream applications such as text-to-code generation [26, 49, 90], code-to-code transpilation [2, 33, 91], and code understanding (mapping code to classification labels) [92, 93].

Despite these impressive advances, most existing models interpret code as mere sequences of subword tokens, overlooking its intrinsic structured nature. Prior research has shown that leveraging the Abstract Syntax Tree (AST) of code can significantly improve performance on code-related tasks [9, 10]. Some studies also use code obfuscation during pretraining to teach models about abstract code structures [22, 14]. However, these models often rely on computationally expensive processes like Control-Flow Analysis (CFA), obfuscation, or even actual code execution. Such dependency limits their scalability and imposes stringent conditions like code executability. Consequently, these methods may struggle with real-world code, especially in intricate

languages like C/C++, where comprehensive analysis remains elusive.

In this study, we propose AST-T5, a pretraining paradigm that leverages the Abstract Syntax Tree (AST) structure of code. The key contribution in AST-T5 is a simple yet effective way to exploit code semantics, without the need to run expensive program analysis or execution. Using a lightweight, multi-language parser called Tree-sitter[1], our approach has broad applicability across all syntactically well-defined programming languages. After we parse code into ASTs, we use a dynamic programming-based segmentation algorithm for AST-aware code segmentation to maintain the structural integrity of the input code. Using our novel AST-Aware Span Corruption technique, the model is pretrained to reconstruct various code structures, ranging from individual tokens to entire function bodies. Together, our approach offers three key advantages: (1) enriched bidirectional encoding for improved code understanding, (2) the ability to coherently generate code structures, and (3) a unified, structure-aware pretraining framework that boosts performance across a variety of code-related tasks, particularly in code transpilation.

In addition, other than our specialized AST-aware masking approach, AST-T5 introduces no architecture changes or additional heads, and our pretraining objective remains the same as Vanilla T5. This compatibility enables seamless integration of our model as a drop-in replacement for any T5 variant.

In our experiments, AST-T5 consistently outperforms baselines in code generation, transpilation, and understanding tasks. Through controlled experiments, we empirically demonstrate that these advancements are attributed to our AST-aware pretraining techniques. Notably, AST-T5 not only outperforms similar-sized models like CodeT5 and CodeT5+ across various benchmarks but also remains competitive with, or occasionally even exceeds, the performance of much larger models using the HumanEval [26] and the MBPP [49] benchmarks. Furthermore, the inherent AST-awareness of AST-T5 offers unique advantages in structure-sensitive tasks, such as code-to-code transpilation and Clone Detection, highlighting its effectiveness at capturing the structural nuances of code.

4.2 Related Work

Language Models for Code. Language models (LMs) extended their use from NLP to code understanding and generation. Encoder-only models generally excel in code understanding when finetuned with classifiers [13], while decoder-only models are optimized for code generation through their autoregressive nature [26, 54, 1]. However,

[1] `https://tree-sitter.github.io/tree-sitter/`

these models can falter outside their primary domains of expertise or require increased resources for comparable outcomes. Our work focuses on encoder-decoder models, aiming to efficiently balance performance in both understanding and generation tasks without excessive computational demands.

Efforts Toward Unified Models. Extending NLP models like BART [94] and T5 [81], several studies have developed encoder-decoder architectures, such as PLBART [61] and CodeT5 [14], to perform well in diverse code-related tasks. Although these models show broader utility, they struggle with generating coherent, executable code in complex scenarios like HumanEval [26]. CodeT5+ [63] seeks to address this limitation through an intricate multi-task pretraining strategy across five objectives. In contrast, our proposed model, AST-T5, uses a novel AST-Aware pretraining paradigm to become a unified model capable of generating fluent code and maintaining superior performance in code understanding tasks. Moreover, AST-T5 is more streamlined, because it only uses a single pretraining objective.

Leveraging Code Structure in Pretraining. Code differs from natural language in two key aspects: its executability and strict structural syntax. Previous research leveraged execution traces for improving model performance [95, 96, 97], but this approach faces scalability challenges when applied to large, web-crawled code datasets used in pretraining. Regarding code's structured nature, various studies have integrated syntactic elements into neural network models. Li et al. [98], Kim et al. [99] and Zügner et al. [100] add AST-Aware attention mechanisms in their models, while Alon et al. [101] and Rabinovich et al. [102] focus on modeling AST node expansion operations rather than traditional code tokens. In parallel, Guo et al. [9] and Allamanis et al. [103] explore DFG-Aware attention mechanisms and Graph Neural Networks (GNNs), to interpret code based on its Data Flow Graph (DFG). StructCoder [10] enriches the code input by appending AST and DFG as additional features. These methods, however, necessitate parsing or static analysis for downstream tasks, which is less feasible for incomplete or incorrect code scenarios like bug fixing.

Our work, AST-T5, aligns with methods that utilize code structure only in pretraining, like DOBF [22] and CodeT5 [14], which obfuscate inputs to force the model to grasp abstract structures. Our approach uniquely diverges by using AST-driven segmentation and masking in T5 span corruption during pretraining. This novel approach offers a more refined pretraining signal compared to structure-agnostic T5, equipping our model to proficiently encode and generate semantically coherent code structures.

4.3 Method

In this section, we present AST-T5, a novel pretraining framework for code-based language models that harnesses the power of Abstract Syntax Trees (ASTs). First, AST-T5 parses code into ASTs to enable a deeper understanding of code structure. Leveraging this structure, we introduce AST-Aware Segmentation, an algorithm designed to address Transformer token limits while retaining the semantic coherence of the code. Second, we introduce AST-Aware Span Corruption, a masking technique that pretrains AST-T5 to reconstruct code structures ranging from individual tokens to entire function bodies, enhancing both its flexibility and structure-awareness.

4.3.1 Parsing Code Into ASTs

Unlike traditional language models on code that handle code as simple sequences of subword tokens, AST-T5 leverages the Abstract Syntax Tree (AST) of code to gain semantic insights. For parsing purposes, we assume the provided code is syntactically valid—a reasonable assumption for tasks like code transpilation and understanding. Instead of the often computationally-intensive or infeasible methods of Control-Flow Analysis (CFA) or code execution [9, 10], our method only requires the code to be parsable. We use Tree-sitter, a multi-language parser, to construct the ASTs, where each subtree represents a consecutive span of subword tokens, and every leaf node represents an individual token.

4.3.2 AST-Aware Segmentation

In this subsection, we describe our AST-Aware Segmentation method, which splits lengthy code files into chunks in a structure-perserving manner.

Segmentation in language model pretraining is a critical yet often overlooked aspect. Transformer LMs impose token limits on input sequences, making segmentation essential for fitting these inputs within the `max_len` constraint. A naive approach is Greedy Segmentation, where each chunk, except the last, contains exactly `max_len` tokens Figure 4.2 (Left). This strategy has been widely adopted in previous works, such as CodeT5 [14].

Research in NLP by Liu et al. [41] underscores that segmentation respecting sentence and document boundaries outperforms the greedy strategy. Given programming language's inherently structured nature, which is arguably more complex than natural

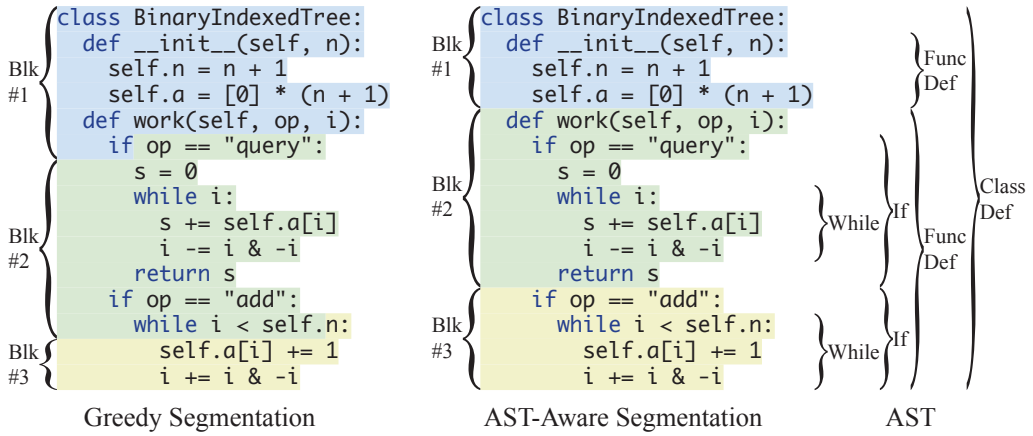


Figure 4.2: Comparison between Greedy Segmentation and AST-Aware Segmentation: For a 112-token code example with `max_len` set at 48, Greedy Segmentation places the first 48 tokens in Block 1, the next 48 tokens in Block 2, and the remaining in Block 3, disrupting the structural integrity of the code. In contrast, AST-Aware Segmentation uses a dynamic programming algorithm to smartly partition the code, aligning with boundaries of member functions or major function branches, thereby preserving the code's structure. The accompanying AST, with some levels pruned for clarity, corroborates that these segmentations indeed coincide with key subtree demarcations.

language, a more sophisticated segmentation approach is even more important. However, this area remains largely unexplored.

AST-Aware Segmentation is our novel approach designed to preserve the AST structure of code during segmentation. Unlike Greedy Segmentation, which can indiscriminately fragment AST structures, our method strategically minimizes such disruptions. As illustrated in the example in Figure 4.2, Greedy Segmentation leads to nine instances of AST breaks—between Block 1 and Block 2, it breaks `If`, `FuncDef`, and `ClassDef`; between Block 2 and Block 3, it breaks `Attr`, `BinaryExpr`, `While`, `If`, `FuncDef`, and `ClassDef`. In contrast, our AST-Aware approach results in only three breaks: between Block 1 and Block 2, it breaks `ClassDef`, and between Block 2 and Block 3, it breaks `FuncDef` and `ClassDef`.

To identify optimal partition boundaries, we developed the following dynamic programming (DP)-based algorithm:

Algorithm 2 Dynamic Programming in AST-Aware Segmentation

```
# n: the length of the code file
#    (number of tokens)
# m: the max number of segments;
#    approximately n / max_len
for k in range(1, m + 1):
  q = Queue()  # double ended queue
  for i in range(1, n + 1):
    while (q.nonempty() and
           q.left() < i - max_len):
      # pop indices before i - max_len
      q.pop_left()
    while (q.nonempty() and
           dp[k-1, q.right()] > dp[k-1, i-1]):
      # maintain monotonicity of values
      q.pop_right()
    q.push_right(i - 1)  # push i - 1
    best_j = q.left()
    # guaranteed to have the smallest value
    prev[k, i] = best_j
    dp[k, i] = cost[i] + dp[k - 1, best_j]
```

1. We construct an array `cost`, where `cost[i]` denotes the number of AST-structure breaks that would occur if partitioning happened right after token i. This array is populated by traversing the AST and incrementing `cost[l..r - 1]` by 1 for each span $[l, r]$ associated with an AST subtree.
2. We define a 2-D array `dp`, where `dp[k, i]` represents the the minimum total number of AST-structure breaks when k partitions are made for the first i tokens, ending the last partition right after the i-th token. The state transition equation is:

$$\texttt{dp}[k, i] = \texttt{cost}[i] + \min_{i-\texttt{max\_len} \le j < i} \texttt{dp}[k-1, j] \tag{4.1}$$

3. While the naive DP algorithm has a quadratic time complexity $O(n^2)$ relative to the code file length n, it can be optimized to $O(n^2/\texttt{max\_len})$ by employing a monotonic queue for sliding-window minimum calculations. This allows for efficient computation across most code files. The pseudocode of the optimized dynamic programming algorithm is shown in Algorithm 2. See Section 4.7.2 for details about complexity calculations.
4. The algorithm outputs the partition associated with `dp[k_min, n]`, where `k_min` $= \arg\min_k(\texttt{dp}[k, n])$, as the most optimal partition.

In comparing AST-Aware Segmentation with Greedy Segmentation—using the exam-

ple in Figure 4.2—we find that the former presents more coherent code segments to the model during pretraining. Conversely, the latter introduces noisy partial expressions near partition boundaries. Consequently, AST-Aware Segmentation not only optimizes the pretraining process but also reduces the mismatch between pretraining and downstream tasks, which often involve complete function definitions as inputs.

4.3.3 Pretraining with Span Corruption

AST-T5's pretraining is based on *span corruption*, a well-established method for pretraining transformer encoder-decoder models [81]. In this approach, 15% of the input tokens are randomly masked and replaced by unique "sentinel" tokens, distinct within each example. Each unique sentinel token is associated with a specific ID and added to the model's vocabulary.

During pretraining, the encoder processes the corrupted input sequence. The decoder's objective is to reconstruct the dropped-out tokens based on the encoder's output representations. Specifically, the target sequence consists of the masked spans of tokens, demarcated by their corresponding sentinel tokens. This framework effectively trains the model to recover the original text from a corrupted input. Figure 4.1 (Left) illustrates an example of the input-output pair for span corruption.

4.3.4 AST-Aware Subtree Corruption

AST-T5 augments the traditional span corruption paradigm by incorporating AST-awareness. Rather than arbitrarily masking consecutive token spans, AST-T5 masks code spans corresponding to AST subtrees, ranging from individual expressions to entire function bodies.

Subtree Masking. We use a recursive algorithm, outlined in Algorithm 3, to traverse the AST and select subtrees for masking. The algorithm aims to fulfill two goals:

1. Introduce sufficient randomness across training epochs to enhance generalization.
2. Control the masking granularity via a tunable hyperparameter θ (named `theta` in Algorithm 3, Line 9).

The "mask quota" m denotes the number of tokens to be masked in a subtree rooted at node t. The size of a subtree corresponds to the number of tokens it encompasses,

Algorithm 3 Subtree Selection in AST-Aware Subtree Corruption

```
1  def mask_subtree(t: ASTNode, m: int):
2    """mask m tokens in subtree t"""
3    ordered_children = []
4    m_remaining = m
5    # distribute m tokens among children of t
6    for child in t.children:
7      # theta: a hyperparameter to control
8      #        masking granularity
9      if child.size > theta:
10       # same mask ratio as the current subtree
11       m_child = m * (child.size / t.size)
12       mask_subtree(child, m_child)  # recurse
13       m_remaining -= m_child
14     else:
15       ordered_children.append(child)
16   weighted_shuffle(ordered_children)
17   # greedy allocation of remaining mask quota
18   for child in ordered_children:
19     m_child = min(m_remaining, child.size)
20     mask_subtree(child, m_child)
21     m_remaining -= m_child
```

derived from the cumulative sizes of its children. For larger subtrees that exceed the size threshold θ, masking is applied recursively (Lines 9-13). Meanwhile, smaller subtrees undergo a weighted shuffle, and the quota m is then apportioned among t's children in a greedy fashion according to the shuffled order (Lines 17-21). The weights for shuffling are determined by a heuristic function on the size of each child, such that masking probabilities are distributed uniformly across leaf nodes. To create a subtree mask for an AST rooted at t with a mask ratio r (e.g., 15% or 25%), one can use `mask_subtree`$(t, \lfloor |t| \cdot r \rfloor)$.

The parameter θ controls the granularity of masking. For example, with $\theta = 5$, the algorithm has a high probability to mask individual tokens and short expressions. As θ increases to 20, the algorithm is more likely to mask larger constructs such as statements. When $\theta = 100$, the probability increases for masking structures like loops or entire function bodies. To foster diverse training scenarios, θ is randomly sampled within a predefined range (e.g., 5 to 100) for each training example. This allows the pretraining framework to inherently accommodate tasks as varied as single-token completion to full function body generation from a given signature.

The subtree masking strategy is the primary distinction between our AST-Aware Subtree Corruption and the Vanilla T5 Span Corruption, as illustrated in Figure 4.1. While conventional T5 variants mask random token spans, with an average span

length of 3 [81] and neglecting code structures, our method targets the masking of AST subtrees, potentially encompassing up to 100 tokens. This equips AST-T5 for generation of various code structures coherently.

Pretraining Objective. Except for the strategy used to select masked tokens and the segmentation strategy described in Section 4.3.2 , our approach adheres to the workflow described in Section 4.3.3. Once subtrees are selected for masking and replaced with sentinel tokens, the encoder processes this modified input. Subsequently, the decoder is tasked with reconstructing the original tokens within the masked subtrees. A side-by-side comparison between our approach and the Vanilla Span Corruption in T5 is presented in Figure 4.1.

4.4 Experimental Setup

Model Architecture. AST-T5 has an architecture similar to $T5_{BASE}$ [81], comprising a 12-layer encoder and a 12-layer decoder, where each layer has 768 dimensions and 12 attention heads. In total, the model has 277M parameters.

Pretraining. AST-T5 is pretrained on a subset of The Stack Dedup corpus [51], a near-deduplicated version of The Stack—a 3.1TB collection of permissively licensed source code from GitHub cutoff at April 2022, spanning 358 programming languages. For our experiments, AST-T5's training involves Python, Java, C, C++, C#, Markdown, and reStructuredText subsets, comprising a 588GB dataset with 93M code and natural language files.

Each file is first parsed into its AST using the Tree-Sitter multi-language parser, and then tokenized with byte-level Byte-Pair Encoding (BPE) using a byte-level BPE token vocabulary. Following AST-Aware Segmentation, these files are partitioned into chunks of 1,024 tokens. Our model is pretrained using the AST-Aware Subtree Corruption objective for 524 billion tokens (1,024 tokens per sequence, 1,024 sequences per batch, and 500k steps). For each training example, we apply AST-Aware Subtree Corruption of it is code, or apply Vanilla T5 Span Corruption of it is natural language. For code, the threshold, θ, is uniformly sampled from 5 to 100. For text, the length of each masked span is uniformly sampled from 1 to 10. Pretraining uses PyTorch, Fairseq[2] and FlashAttention [104] and is conducted on 8 nodes, each with 8x NVIDIA A100 40GB GPUs. Further pretraining hyperparameters are detailed in Section 4.7.3.

[2] `https://github.com/facebookresearch/fairseq`

Table 4.1: Overview of our evaluation benchmarks about test set size, task type, and evaluation metric for each task. "Generation" tasks involve mapping natural language to code, "Transpilation" tasks involve translating code from one programming language to another, and "Understanding" tasks involve classifying code into categorical labels. For MBPP, we follow Nijkamp et al. [1] and evaluate our model on the entire "sanitized" subset without few-shot prompts. For evaluation metrics, "Pass@1" indicates code execution on unit-tests provided in the benchmark using a single generated code per example, with reported pass rates. "EM" (Exact Match) evaluates textual equivalence without execution by comparing two canonicalized code pieces. "Acc" means accuracy in classification tasks. We omit "BLEU scores" because high BLEU values (> 50) can still correspond to unexecutable or significantly flawed code [2], which is not useful in real-world applications. We also discuss evaluation results using the CodeBLEU [3] metric in Section 4.7.6.

	Size	**Type**	**Metric**
HumanEval	164	Generation	Pass@1
MBPP	427	Generation	Pass@1
Concode	2,000	Generation	EM
Bugs2Fix	12,379	Transpilation	EM
Java-C#	1,000	Transpilation	EM
BigCloneBench	415,416	Understanding	F1
Defect Detect	27,318	Understanding	Acc

Evaluation. We evaluate AST-T5 across three types of tasks: text-to-code generation, code-to-code transpilation, and code understanding (classification). Our evaluation encompasses tasks from the CodeXGLUE meta-benchmark [2] and also includes HumanEval [26] and MBPP [49]. Specifically, for text-to-code generation, we assess performance using HumanEval, MBPP, and Concode [90]; for transpilation, we use CodeXGLUE Java-C# and Bugs2Fix [91] for evaluation; and for understanding, we use BigCloneBench [93] and the Defect Detection task proposed by Zhou et al. [92]. Detailed metrics and statistics of these datasets are provided in Table 4.1.

We finetune AST-T5 on the training datasets of all downstream tasks, adhering to the methodology by Raffel et al. [81]. For the HumanEval task, which lacks its own training dataset, we use CodeSearchNet [105], aligning with the approach of Wang et al. [63]. The prompt templates for finetuning are constructed using the

PromptSource framework [106]. The finetuning takes 50k steps, with the peak learning rate set at 10% of the pretraining learning rate. All other hyperparameters from pretraining are retained without further adjustments, and we train only one finetuned model. During inference, rank classification is employed for code understanding tasks and beam search is used for generative tasks, following Sanh et al. [107]. For CodeXGLUE, we evaluate our model on the test set using five prompt templates for each task and report the average performance; for HumanEval and MBPP, we evaluate the top-1 generated output from beam search.

Baselines. We first benchmark AST-T5 against our own T5 baselines to ensure a controlled comparison. All models share identical Transformer architectures, pretraining data, and computational settings, differing only in the use of AST-Aware Segmentation and Subtree Corruption techniques by AST-T5. This setup directly evaluates the efficacy of our proposed methods.

We further benchmark AST-T5 against other language models for code-related tasks. These include decoder-only models such as the GPT variants [17, 26, 108, 109], PaLM [52], InCoder [54], and LLaMa [89]. We also compare with encoder-decoder models, including PLBART [61], CodeT5 [14], StructCoder [10], and CodeT5+ [63]. Notably, $\text{CodeT5}_{\text{BASE}}$ and CodeT5+ (220M) closely resemble our model in terms of architecture and size, but AST-T5 distinguishes itself with its AST-Aware pretraining techniques.

4.5 Evaluation Results

In this section, we evaluate AST-T5 across multiple benchmarks. First, we analyze the contributions of each component within our AST-aware pretraining framework through controlled experiments. Next, we benchmark AST-T5 against existing models in prior work.

4.5.1 Pretraining Procedure Analysis

In this subsection, we analyze the key components that contribute to the pretraining of AST-T5 models. Holding the model architecture, pretraining datasets, and computational environment constant, we sequentially add one component at a time to a T5 baseline trained on code, culminating in our finalized AST-T5 model. Table 4.2 presents the experimental results. These results show that:

Table 4.2: Performance comparison of various pretraining configurations for downstream tasks. Each row represents a sequential modification applied to the model in the previous row. Metrics include "Pass@1" rate for HumanEval, "Exact Match" rate for CONCODE, Bugs2Fix (for "Small" and "Medium" code lengths splits), and Java-C# transpilation (both Java-to-C# and C#-to-Java). F1 score is used for Clone Detection, and Accuracy for Defect Detection, consistent with prior studies.

	Generation		Transpilation		Understanding		
Pretraining Config	**HumanEval**	**Concode**	**Bugs2Fix**	**Java-C#**	**Clone**	**Defect**	**Avg**
T5	5.2	18.3	21.2/13.8	65.5/68.4	96.9	64.1	44.2
+ AST. Segmentation	7.2	20.2	22.5/15.1	66.3/69.3	98.3	65.9	45.7
+ AST. Subtree Corrupt	9.6	22.1	23.3/**16.5**	67.3/72.2	**98.6**	**66.0**	47.0
+ Mask 25% (AST-T5)	14.0	**22.9**	**23.8**/16.1	**68.9/72.3**	**98.6**	65.8	**47.9**
+ Mask 50%	**14.3**	22.0	21.9/15.0	66.5/70.1	97.1	64.2	46.4

AST-Aware Segmentation enhances code language models. A comparison between the first two rows of Table 4.2 shows that the model trained with AST-Aware Segmentation consistently outperforms the T5 baseline that uses Greedy Segmentation across all tasks. The advantage stems from the fact that AST-Aware Segmentation produces less fragmented and thus less noisy training inputs during pretraining. Given that most downstream tasks present coherent code structures, such as entire function definitions, the consistency upheld by AST-Aware pretraining aligns better with these structures, leading to improved generalization.

AST-Aware Span Corruption further boosts generation performance. A comparison between the second and third rows of Table 4.2 reveals an improvement when shifting from Vanilla T5 Span Corruption to our AST-Aware Subtree Corruption. This performance gain is especially notable in generation and transpilation tasks. Such enhancements stem from the ability of AST-Aware Subtree Corruption to guide the model in generating code with better coherence and structural integrity.

Increasing masking ratio improves generation performance. The typical span corruption mask ratio in T5 is set at 15%. Increasing this ratio could potentially enhance the model's generation capabilities, albeit potentially at the expense of understanding tasks. Essentially, a mask ratio of 100% would emulate a GPT-like, decoder-only Transformer. However, in our experiments (last two rows of

Table 4.3: Results of AST-T5 on downstream tasks compared with reported results of established language models. Evaluation metrics align with those in Table 1. Our focus is primarily on models with similar sizes as AST-T5, specifically the "Base" models (100M to 300M parameters), while comparisons against larger models are depicted in Figure 3. Some models are either encoder-only or decoder-only and are thus not suited for certain tasks. These results are labeled with "N/A" in this table because they are not available in the literature.

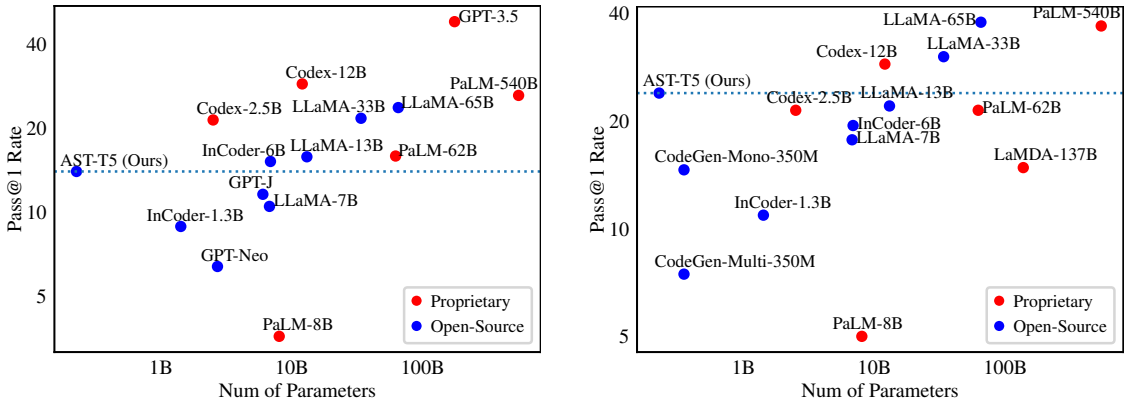
	Generation		**Transpilation**		**Understanding**	
Model	**HumanEval**	**Concode**	**Bugs2Fix**	**Java-C#**	**Clone**	**Defect**
CodeBERT	N/A	N/A	16.4 / 5.2	59.0/58.8	96.5	62.1
GraphCodeBERT	N/A	N/A	17.3 / 9.1	59.4/58.8	97.1	N/A
PLBART	N/A	18.8	19.2 / 9.0	64.6/65.0	97.2	63.2
CodeT5	N/A	22.3	21.6/14.0	65.9/66.9	97.2	65.8
CodeT5+$_{\text{Base}}$	12.0	N/A	N/A	N/A	95.2	**66.1**
StructCoder	N/A	22.4	N/A	66.9/68.7	N/A	N/A
AST-T5 (Ours)	**14.0**	**22.9**	**23.8/16.1**	**68.9/72.3**	**98.6**	65.8

Table 4.2), we observed that raising the mask ratio from 15% to 25% significantly improved generation capabilities without noticeably compromising performance in understanding tasks. Further analysis shows that increasing the masking ratio to 50% yields only a marginal improvement on HumanEval (from 14.0 to 14.3), while adversely impacting transpilation and understanding tasks. Thus, we settled on a 25% mask ratio for our AST-T5 model.

4.5.2 Main Results

Table 4.3 shows AST-T5's performance on downstream tasks compared with previously published results of similarly sized models, specifically those within the "Base" scale (100M to 300M parameters). Figure 4.3a and Figure 4.3b extends this comparison, comparing AST-T5 with larger models using the HumanEval benchmark and the MBPP benchmark, respectively. Additional results on EvalPlus are shown in Section 4.7.4. These results show that:

AST-T5 excels as a unified and parameter-efficient LM for various code-related tasks. While comparable in size, AST-T5 consistently outperforms similar-



(a) HumanEval (b) MBPP

Figure 4.3: Visualizations of AST-T5's performance on HumanEval and MBPP compared to other models compared to models exceeding 300M parameters. Each point on each scatter plot represents a model. The x-axis shows the parameter count in log-scale, while the y-axis shows the Pass@1 rate on HumanEval or MBPP in log-scale. Model open-source status is color-coded: **blue** for open-source and **red** for proprietary.

sized models such as CodeT5 [14] and CodeT5+ [63] in code generation, transpilation, and understanding. Notably, while CodeT5 and CodeT5+ are models at the Base scale, they were evaluated across different tasks. Our model, AST-T5, outperforms the best results of these two models across multiple benchmarks at the same time. Moreover, Figure 4.3a highlights AST-T5's competitiveness against significantly larger models like GPT-J [108] and LLaMa-7B [89] on the HumanEval benchmark, underscoring our model's parameter efficiency. Similarly, Figure 4.3b demonstrates AST-T5's advantages over LLaMa-7B and Codex-2.5B [26] on the MBPP benchmark, showing the effectiveness of AST-T5.

AST-T5 exhibits unique strengths in transpilation through AST-awareness. Table 4.3 highlights AST-T5's superior performance in code-to-code transpilation tasks, showcasing gains a substantial gain of 2 to 5 points on Bugs2Fix and Java-C# transpilation. In transpilation, while surface-level code can exhibit significant variability, the intrinsic AST structures of the source and target often maintain a notable similarity. The capability of AST-T5 to exploit this structural similarity is crucial to its effectiveness. The benefits of being structure-aware are further

exemplified by AST-T5's leading results in Clone Detection, where it surpasses CodeT5 by 3 points, because AST comparisons yield more precise insights than direct code comparisons.

4.6 Conclusion and Future Work

In this chapter, we present AST-T5, a novel pretraining paradigm that harnesses the power of Abstract Syntax Trees (ASTs) to boost the performance of code-centric language models. Using two structure-aware techniques, AST-T5 not only outperforms models of comparable size but also competes favorably against some larger counterparts. The simplicity of AST-T5 lies in its singular pretraining objective and its adaptability as a drop-in replacement for any encoder-decoder LM, highlighting its potential for real-world deployments. Moving forward, we aim to explore the scalability of AST-T5 by training larger models on more expansive datasets.

4.7 Appendix

4.7.1 Limitations

AST-T5 is specifically designed to enhance code generation performance by exclusively masking code within AST subtrees during pretraining. While this specialized approach is advantageous for code generation tasks, it may result in suboptimal performance in natural language generation. Acknowledging this limitation, future versions of AST-T5 could investigate strategies such as masking docstrings and comments to broaden its applicability. This would potentially improve performance across various tasks, including code summarization.

4.7.2 More about AST-Aware Segmentation

In Section 4.3.2, we use a dynamic programming algorithm to calculate the segmentation that results in the least number of AST structure breaks. A naive implementation of the DP algorithm is shown in Algorithm 4.

Denote the length of the code file (in tokens) by n. In the algorithm, m denotes the maximum number of chunks that the file can be split into, which is approximately $n/\text{max\_len}$. So this implementation has time complexity $O(mn \cdot \text{max\_len}) = O(n^2)$, which is not feasible for longer code files. To optimize this algorithm, we use a monotonic queue to compute the sliding-window minimum, as described in Algorithm 2.

Algorithm 4 Dynamic Programming in AST-Aware Segmentation (Before Optimization)

```
for k in range(1, m + 1):
    for i in range(1, n + 1):
        best_j = i - max_len
        for j in range(i - max_len + 1, i):
            if dp[k - 1, j] < dp[k - 1, best_j]:
                best_j = j
        prev[k, i] = best_j
        dp[k, i] = cost[i] + min_value
```

Each element is only pushed into and popped out of the monotonic queue once, so the time complexity of the optimized algorithm is $O(nm) = O(n^2/\text{max\_len})$, making the algorithm $\sim$ 1000x faster when max_len = 1024. This allows the algorithm to segment each code file with 100k tokens in milliseconds.

4.7.3 Pretraining Hyperparameters

Table 4.4 shows the pretraining hyperparameters for our proposed AST-T5 model.

4.7.4 Evaluation Results on EvalPlus

We extend our evaluation to include EvalPlus [4], a more rigorous benchmark that enhances the original HumanEval and MBPP datasets with a substantial number of additional test cases. EvalPlus is designed to provide a more accurate evaluation of the correctness of programs produced by LLMs.

For our tests on HumanEval+ and MBPP+, we use the same hyperparameters used in our evaluations of HumanEval and MBPP. It is important to note that the hyperparameter configurations used in our study are not directly comparable to those used for the models listed on the EvalPlus leaderboard[3]. Our results are compared against established models including GPT-Neo, GPT-J, InCoder, and CodeGen-2 [110].

As shown in Table 4.5, our 277M-parameter AST-T5 outperforms larger models like InCoder-6.7B and CodeGen2-1B, showing the effectiveness and parameter efficiency

[3] https://evalplus.github.io/leaderboard.html

Table 4.4: Pretraining hyperparameters for our AST-T5 model.

Encoder Layers	12
Decoder Layers	12
Hidden Dimension	768
Peak Learning Rate	2e-4
Batch Size	1,024
Warm-Up Steps	10,000
Total Steps	500,000
Sequence Length	1,024
Mask Ratio	25%
Min Subtree Corruption Threshold θ	5
Max Subtree Corruption Threshold θ	100
Relative Position Encoding Buckets	32
Relative Position Encoding Max Distance	128
Adam ϵ	1e-6
Adam (β_1, β_2)	(0.9, 0.98)
Clip Norm	2.0
Dropout	0.1
Weight Decay	0.01

of AST-T5.

4.7.5 Evaluation Results on Multi-Lingual Code Generation

Table 4.6 presents a comparative analysis of our AST-T5 model on Python and Java subsets of the multi-lingual HumanEval and MBXP benchmarks [67]. This analysis includes models such as BLOOM [111], OPT [112], and various configurations of CodeGen [1], as reported in Athiwaratkun et al. [67]. Our results show AST-T5's superior performance across all benchmarks compared to the CodeGen-multi-350M. Furthermore, AST-T5, having 277M parameters, outperforms larger counterparts like BLOOM-7.1B and OPT-13B.

Table 4.5: Performance of AST-T5 on HumanEval+ and MBPP+ benchmarks, compared with reported numbers of language models listed on the EvalPlus leaderboard. The evaluation metric used is Pass@1.

	#Params	**HumanEval+**	**MBPP+**
GPT-Neo	2.7B	6.7	7.9
GPT-J	6B	11.0	12.2
InCoder-1.3B	1.3B	11.0	12.2
InCoder-6.7B	6.7B	12.2	15.9
CodeGen2-1B	1B	9.1	11.0
CodeGen2-3B	3B	12.8	15.9
CodeGen2-7B	7B	17.7	18.3
CodeGen2-16B	16B	16.5	19.5
AST-T5 (Ours)	277M	12.8	19.3

Table 4.6: Results of AST-T5 on multi-lingual HumanEval and MBXP compared with reported results of established language models. The evaluation metric is Pass@1.

	#Params	**HumanEval**		**MBXP**	
		Python	Java	Python	Java
CodeGen-multi	350M	7.3	5.0	7.5	8.2
CodeGen-mono	350M	10.3	3.1	**14.6**	1.9
AST-T5 (Ours)	277M	14.0	**10.6**	**23.9**	**9.8**
BLOOM	7.1B	7.9	8.1	7.0	7.8
OPT	13B	0.6	0.6	1.4	1.4
CodeGen-multi	2B	11.0	11.2	18.8	19.5
CodeGen-mono	2B	20.7	5.0	31.7	16.7
CodeGen-multi	6B	15.2	10.6	22.5	21.7
CodeGen-mono	6B	19.5	8.7	37.2	19.8
CodeGen-multi	16B	17.1	16.2	24.2	28.0
CodeGen-mono	16B	22.6	22.4	40.6	26.8

Table 4.7: Results of AST-T5 on CONCODE with reported results of established language models. The evaluation metric is exact match score and CodeBLEU.

	EM	**CodeBLEU**
GPT-2	17.4	29.7
CodeGPT-2	18.3	32.7
CodeGPT-adapted	20.1	36.0
PLBART	18.8	38.5
CodeT5-Small	21.6	41.4
CodeT5-Base	22.3	43.2
AST-T5 (Ours)	**22.9**	**45.0**

4.7.6 Evaluation Results in CodeBLEU

Table 4.7 presents the performance of various models on the Concode dataset using the CodeBLEU metric, as reported in [14]. CodeBLEU, specifically designed for evaluating code synthesis, computes a weighted average of three scores: textual match (BLEU), AST match, and Data Flow Graph (DFG) match. Our findings show a clear correlation between CodeBLEU and exact match scores.

Chapter 5

AST-FIM: Structure-Aware Fill-in-the-Middle Pretraining for Code

We propose and evaluate AST-FIM, a pretraining strategy that leverages Abstract Syntax Trees (ASTs) to mask complete syntactic structures at scale, ensuring coherent training examples better aligned with real-world code edits. To evaluate real-world fill-in-the-middle (FIM) programming tasks, we introduce Real-FIM-Eval, a benchmark derived from 30,000+ GitHub commits across 12 languages. On infilling tasks, experiments on 1B and 8B parameter models show that AST-FIM outperforms standard random-character FIM by significant margins. On left-to-right tasks, AST-FIM offers same performance whereas standard random character FIM harmed left-to-right generation as FIM rates increase.

5.1 Introduction

Large Language Models (LLMs) trained on diverse, internet-scale datasets have shown remarkable success across various domains, especially code-related applications. Code LLMs now power code generation, code completion/editing, test generation and much more. Following early works [50, 54], *fill-in-the-middle (FIM)* pretraining has emerged as a defining feature of recent code LLMs such as CodeLlama [48], CodeGemma [113], StarCoder2 [114], DeepSeek-Coder [57], Qwen2.5-Coder [115], and Codestral 2501 [116]. Specifically, FIM trains an autoregressive decoder-only model for *infilling*: reconstructing a masked span (the "middle") using its surrounding prefix and suffix.

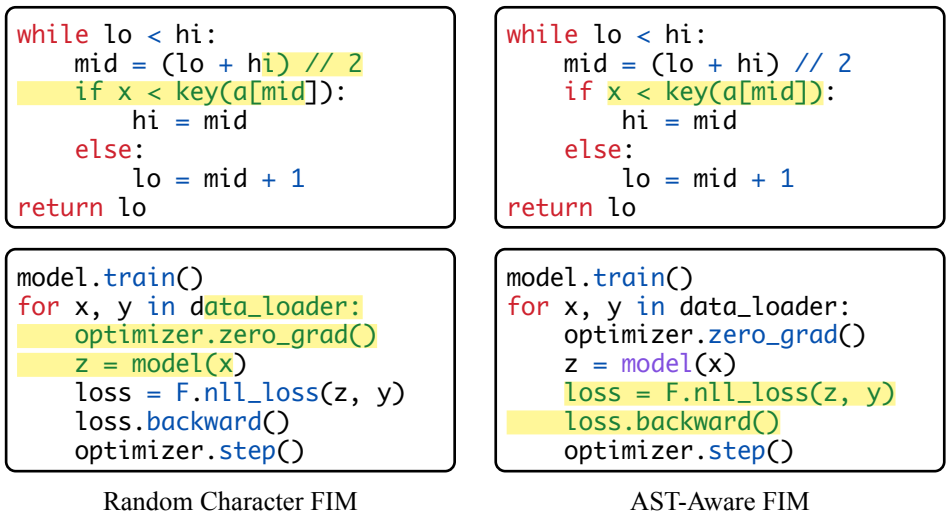


Figure 5.1: **Comparison of masking strategies in Random-Character FIM (Rand-FIM) and our proposed AST-Aware FIM (AST-FIM)** in two examples. The highlighted code is the masked part for FIM training. **Left:** Rand-FIM treats code as a character sequence, masking a random span. **Right:** AST-FIM respects code structure by masking complete subtrees. This syntax-aware masking aligns more closely with typical developer-code interactions.

However, existing FIM pretraining approaches has a limitation: they treat code as text sequences, disregrading its inherent structure. Such *random-character FIM (Rand-FIM)* approach can split code at arbitrary character positions. This creates a mismatch with real-world code editing patterns where developers typically insert complete syntactic elements—for example, adding new statements to a code block or adding entire function definitions to a class. We hypothesize that this misalignment between pretraining inputs and application scenarios leads to suboptimal performance on code infilling tasks. Additionally, Rand-FIM generates noisy and fragmented training inputs, which may degrade Left-to-Right (L2R) generation capability of LLMs. As shown in Guo et al. [57], increasing FIM rates with Rand-FIM harms code generation performance of LLMs.

To address this limitation, we propose **AST-Aware Fill-in-the-Middle (AST-FIM)**, which aligns pretraining with real-world code edits through syntax-aware masking. Real-world code insertions—such as adding statements to a code block or

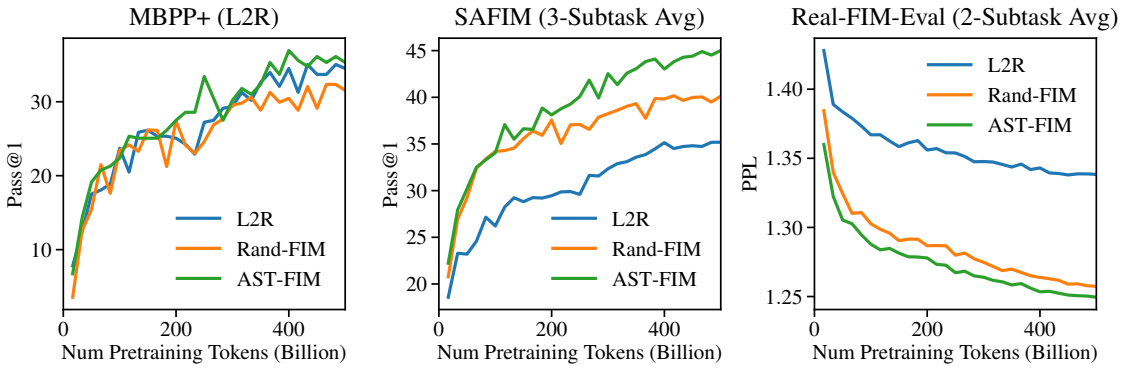


Figure 5.2: **Performance of each model during pretraining, checkpointed every 4000 steps (16.7B tokens). Left:** Pass@1 of MBPP+, a left-to-right task (higher is better). **Middle:** Average pass@1 of SAFIM-Algorithm, SAFIM-Control, and SAFIM-API (higher is better). **Right:** Average perplexity of Real-FIM-Eval-Add and Real-FIM-Eval-Edit (lower is better).

adding functions in a class declaration—typically involve complete syntactic units. AST-FIM leverages Abstract Syntax Trees (ASTs) to mask entire subtrees (Figure 5.1), preserving structural coherence while mimicking developer actions. We implement AST-FIM through a *language-agnostic* masking algorithm. It supports 100+ programming languages without language-specific engineering, using a universal code parser called TreeSitter. This approach maintains the simplicity of FIM pretraining while aligning the pretraining objective with developer behavior.

To verify that AST-FIM's syntax-aware masking translates into better performance in real-world code completion tasks, we need an evaluation benchmark that mirrors everyday developer activity. However, existing benchmarks are difficult to run [77, 117] or unrepresentative of real code changes [50, 118]. We therefore propose **Real-FIM-Eval**, sourced from more than 30,000 recent commits from highly active GitHub projects spanning 12 programming languages. The model is tasked to complete code segments inserted or modified in a git commit, conditioned on their surrounding context. Because every example is a real commit diff, Real-FIM-Eval provides an unbiased, faithful view of real-world code completion and editing abilities of LLMs.

We evaluate AST-FIM and baselines on both standard L2R generation and code

infilling benchmarks, including Real-FIM-Eval. Our experiments in Figure 5.2 show that AST-FIM not only achieves strong performance on code infilling tasks but also retains L2R generation capability, outperforming traditional random-character FIM (Rand-FIM) by significant margins. These trends hold consistently across both 1B and 8B parameter scales, highlighting the scalability of our approach. Notably, AST-FIM-8B rivals leading models of comparable size in L2R code generation benchmarks while surpassing all the counterparts in real-world infilling scenarios. By integrating syntax-aware pretraining into decoder-only LLMs, AST-FIM significantly advances the infilling capability of code LLMs without compromising their core capabilities.

5.2 Related Work

LLMs for Code. LLMs have been successful across various domains. While encoder-only and encoder-decoder models exist [20, 13, 81, 14], decoder-only LLMs have overwhelming popularity given their superior scalability and generation performance. Coding is one of the most impactful domains for LLMs. LLMs can handle coding tasks such as code generation [26, 49], real-time code suggestions [117], and editing [77], with applications ranging from synthesizing executable functions to automating repetitive development workflows [119].

Fill-in-the-Middle (FIM) Pretraining. Code LLMs like CodeLlama [48], StarCoder [55], and DeepSeek-Coder [57] use FIM pretraining to enable the ability to condition on both prefix and suffix. Initial work by Bavarian et al. [50] claimed the *FIM-for-free* property: FIM rates up to 0.9 did not harm left-to-right (L2R) generation. But subsequent models like StarCoder [55] and DeepSeek-Coder [57] cap FIM rates at 0.5. The DeepSeek-Coder technical report describes how higher FIM rate causes significant performance drop. Our experiments confirm that traditional FIM with a 0.7 rate indeed degrades L2R performance, highlighting the need for methods that reconcile this tradeoff. To address this, we propose syntax-aware FIM pretraining by leveraging code structure to preserve generation quality while enhancing infilling capability.

FIM Span Selection. While existing FIM methods uniformly sample spans at random, prior work in both natural language processing [120, 121] and programming languages shows that selecting semantically or syntactically meaningful spans improves downstream task performance of trained models. For code, spans often align with subtrees in Abstract Syntax Trees (ASTs): AST-T5 [64] enhances T5's span corruption with syntax-aware pretraining, and Qwen2.5-Coder [115] uses AST-guided FIM during

post-training. Similar to these efforts, our approach leverages AST subtrees for span selection but has two key differences. First, we propose language-agnostic strategies to sample diverse AST structures, eliminating the need for manual, language-specific rules. Second, we are the first to integrate AST-aware FIM into large-scale pretraining of decoder-only LLMs.

Evaluation of Code FIM. Current code infilling benchmarks limitations. Existing FIM benchmarks, including HumanEval Single-Line infilling [50], SAFIM [118], and CruxEval [122], suffer from artificial edit patterns that can misalign with real-world distributions. Benchmarks based on user feedback, like Copilot Arena [117], while reflecting real-world coding scenarios, are prohibitively difficult for monitoring a model during pretraining. To address these gaps, we propose Real-FIM-Eval, a benchmark that samples masks from real git commit patches to mirror real code edits, scales extensively for high signal-to-noise ratio evaluation, and supports direct evaluation of base models without post-training.

5.3 Method

This section describes the proposed pretraining methods. We first review autoregressive language model (LM) pretraining and traditional random-character Fill-in-the-Middle (FIM) pretraining. Then, we introduce our core contribution, AST-Aware FIM. This approach enhances FIM by using the code's Abstract Syntax Tree (AST) structure to generate training inputs from code data.

5.3.1 Language Model Pretraining

LM pretraining starts by processing input text. The text is first tokenized into a sequence of subword tokens using Byte Pair Encoding (BPE). These tokens are then grouped into sequences of a fixed length, for example, 8,192 tokens in our models. The model is pretrained by predicting the next token in the sequence based only on the tokens that came before it:

$$\Pr(x_i|x_1, \ldots, x_{i-1}).$$

This process is called autoregressive or left-to-right (L2R) training.

5.3.2 Fill-in-the-Middle (FIM) Pretraining

FIM pretraining helps autoregressive decoder-only language models perform infilling tasks—generating text at a specific point within a given prompt [50, 54, 121]. The LM uses both the text before the point (*prefix*) and the text after the point (*suffix*) to generate the missing *middle* part. FIM pretraining optimizes the probability

$$\Pr(\text{middle}|\text{prefix}, \text{suffix}).$$

This pretraining method has become popular for recent code LLMs [48, 57, 55, 115].

FIM Implementation. The FIM pretraining works as follows. First, a document is randomly split into a *prefix*, a *middle* part, and a *suffix*. Second, these parts are rearranged so the middle part comes last. Special sentinel tokens are used to combine these parts into a single sequence for the model. Common formats include:

- *Prefix-Suffix-Middle (PSM):* `[PRE] prefix [SUF] suffix [MID] middle [EOT]`
- *Suffix-Prefix-Middle (SPM):* `[PRE] [SUF] suffix [MID] prefix middle [EOT]`

Third, the language model is trained on this rearranged sequence using negative log-likelihood (NLL) loss, just like in ordinary left-to-right training.

Joint Training with L2R. FIM pretraining is usually done jointly with ordinary left-to-right (L2R) pretraining. During training, for each step, the FIM transformation is applied with a certain probability, called the **FIM rate** p. With probability $1 - p$, ordinary L2R training is used for this step. When p is not too large, LMs trained this way can gain the infilling capability while keeping their L2R capability [50].

5.3.3 AST-Aware Fill-in-the-Middle

In FIM pretraining, each input document is divided into three parts: a prefix, a middle section, and a suffix. Our approach introduces a key difference in how this middle section is selected, i.e., the **masking** algorithm. The traditional method, **Rand-FIM**, treats code as a sequence of characters. It selects the masked part by choosing character positions [48, 50], or token positions [54], uniformly at random. In contrast, our proposed method, **AST-Aware FIM (AST-FIM)**, selects the masked part based on the code's syntactical structure. Specifically, the masked part always corresponds to one or more complete subtrees within the code's Abstract Syntax Tree

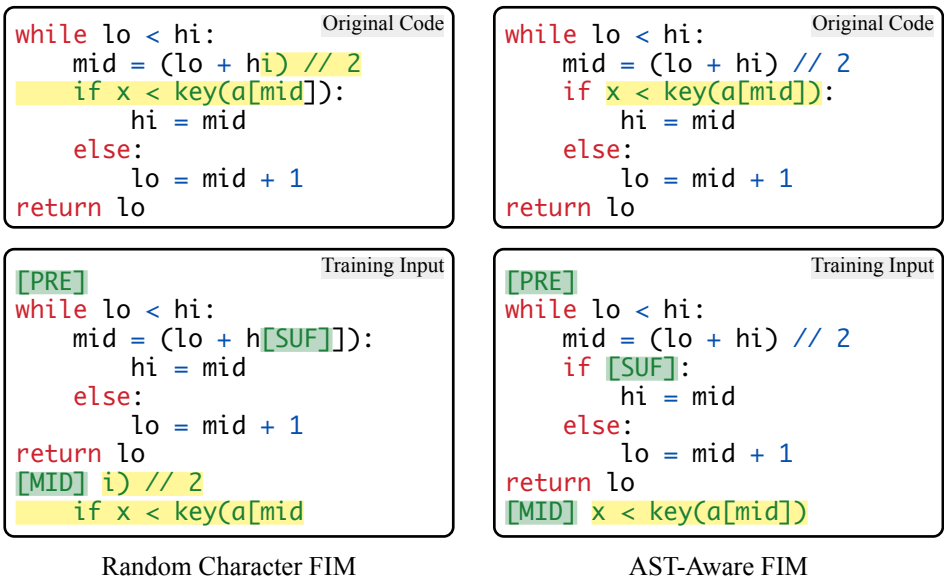


Figure 5.3: **Comparison of training inputs processed by Rand-FIM and AST-FIM using the PSM format.** Given the same code, Rand-FIM selects a random character span as the "middle" part, while AST-FIM selects a span corresponding to entire AST subtrees. AST-FIM generates cleaner training examples that better reflect practical code completion scenarios.

(AST). This way, AST-FIM respects the syntactical structure of the code, which gives the model a strong prior that matches how developers edit and complete code.

Parsing code into ASTs. To implement AST-FIM, we first parse the code into ASTs. We assume that the code files in our training data are syntactically correct. This assumption is generally safe, as most code pushed to GitHub (our training dataset), is syntactically valid. Our approach is lightweight: it only requires parsing; we do not need complex Control-Flow Analysis (CFA) or code execution, which can be computationally expensive or sometimes impossible. For parsing, we use Tree-sitter, a multi-language parser, to construct the ASTs, where each subtree represents a consecutive span of characters in the code.

Masking Algorithm. The FIM masking algorithm determines which span of code to designate as the "middle" part. The input to this algorithm is a code file. The

algorithm then samples a span, ensuring that in AST-FIM, this sampled span aligns with the boundaries of AST subtrees. We designed the masking algorithm with two main goals:

1. The generate masked spans should have good diversity to train the model effectively.
2. The algorithm must be language-agnostic. It should work on any AST without language-specific rules, which is important given that there are 100+ programming languages.

To achieve these goals, we use a mix of two masking methods: **Single-Node Masking** and **Aligned Span Masking**.

Single-Node Masking. This algorithm selects a single AST subtree to mask. We sample an AST node with probability proportional to its size (number of bytes in its corresponding text). We ignore nodes that represent keywords or punctuation, as these are not typically considered part of the *abstract* syntax tree, even though Tree-sitter often parse them as leaf nodes.

Aligned Span Masking. Single-node masking selects only a single AST subtree. But code edits often involve multiple adjacent elements, like multiple statements or multiple methods within a class. Aligned Span Masking allows selecting multiple adjacent AST nodes. Steps:

1. Sample a character span with uniformly random endpoints `[start, end]`.
2. Find the *lowest* AST subtree T that contains this character span. Let $T_1, T_2, \ldots, T_n$ be the direct children of T.
3. Select a continuous sequence of these children, $T_i, T_{i+1}, \ldots, T_j$, to form the middle part. We choose the sequence that has the largest character-level *Intersection over Union (IoU)* with the original random span `[start, end]`.

Training Process. Single-Node Masking and Aligned Span Masking determine how we select the middle part for our AST-FIM pretraining. Then, we format the input using either the PSM or SPM with special tokens. The model is then trained to predict the middle part using NLL loss. Similar to the Rand-FIM, our AST-FIM is typically performed jointly with ordinary L2R objective to ensure the model retains L2R generation capabilities while learning FIM.

Table 5.1: **Distribution of examples across programming languages** in the proposed Real-FIM-Eval benchmark.

Python	Rust	Java	C++	TypeScript	Go	Ruby	C#	JavaScript	Kotlin	PHP	Scala
6,271	4,727	3,716	3,265	3,182	2,587	1,686	1,563	1,502	1,440	1,396	466

5.4 The Real-FIM-Eval Benchmark

We introduce Real-FIM-Eval, a new benchmark developed for this chapter. The motivation behind Real-FIM-Eval is to evaluate FIM capabilities in scenarios that reflect real-world code completion, similar to using an IDE assistant.

5.4.1 Benchmark Construction

Data Source. The Real-FIM-Eval benchmark is built using data from recent GitHub commits between Jan 2025 and Feb 2025. These commits originate from 228 permissively licensed GitHub repositories with 10,000+ stars, spanning top 12 widely-used programming languages. Table 5.1 shows the distribution of examples across these languages. The data collection period is entirely separate from the data used for pretraining our models, minimizing the potential impact of data contamination.

Splits. We process git commits using `diff_match_patch`[1] to identify line-level changes. The commits are then categorized into two splits for Real-FIM-Eval, as visualized in Figure 5.4:

- **Add** (17,879 examples): This split uses git commits where a developer added a new segment of code into an existing file. To create the FIM prompt, we treat the added code segment as the "middle" part that the language model needs to predict. The code surrounding the addition forms the prefix (code before) and the suffix (code after).
- **Edit** (13,922 examples): This split uses git commits where a developer modified existing code by removing a segment and replacing it with a new one. We present this task to LLMs in a *conflict-merge* format. The prompt includes the

[1] `https://github.com/google/diff-match-patch`

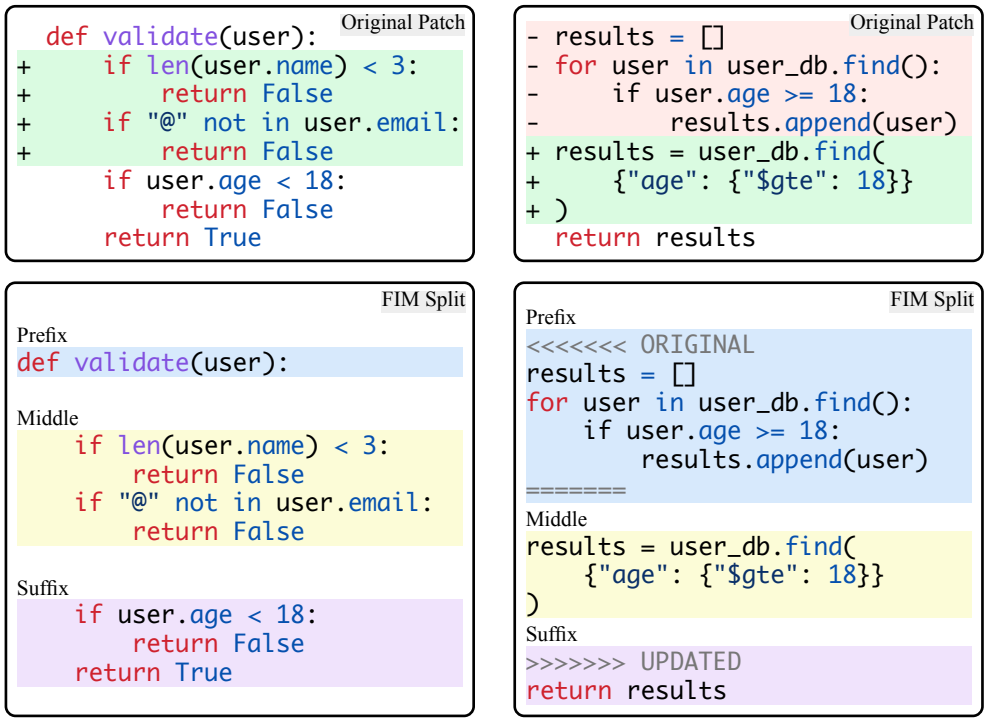


Figure 5.4: **Construction of Fill-in-the-Middle (FIM) examples for the proposed Real-FIM-Eval benchmark splits, derived from real-world git commits. Add:** Uses code insertions; the added code becomes the "middle" to predict. **Edit:** Uses code modifications, presented via a conflict-merge format contrasting the `ORIGINAL` and `UPDATED` code within the surrounding context. The content of the added code is the "middle" part to predict.

code context (prefix and suffix) and marks the original code segment (to be removed). The model is asked to infill the updated code segment.

5.4.2 Evaluation Metric

We evaluate model performance on Real-FIM-Eval using character-level perplexity. This metric measures how well the model predicts the sequence of characters in the ground truth "middle" part. The perplexity is calculated as:

$$\exp\left(-\frac{1}{\text{n\_chars}(\mathbf{y})}\sum_{i\in\text{mid}}\log p_{i,y_i}\right)$$

In this formula, p_{i,y_i} represents the probability the model assigns to the i-th token of the ground truth $\mathbf{y}$ (only the "middle" part that the model is tasked to infill), and $\text{n\_chars}(\mathbf{y})$ is the total number of characters in that ground truth code segment.

We perplexity for evaluation because it provides a scalable and low-noise signal, which helps in obtaining stable comparisons between different models [123]. We do not use execution-based evaluation, because gathering and executing unit tests from web-scraped repos at such a large scale is impractical. Perplexity is good enough to simulate the objective of code completion in an IDE. It reflects the likelihood of generating the code the user intends to write.

5.5 Experimental Setup

This section outlines our experimental setup. We train 1B and 8B models using Rand-FIM and AST-FIM. We evaluate these models on standard code generation tasks and FIM tasks, including Real-FIM-Eval. The goal is to test if our AST-FIM models outperform other FIM methods on code completion while performing comparably to standard L2R models on standard generation tasks.

5.5.1 Training

Model. We conduct experiments on Llama-3-1B and Llama-3-8B model architectures.

Data. Following recent code models, our training data contains 90% programming language data from GitHub and 10% natural language data. The natural language

data includes 8% from Apple DCLM and 2% from Wikipedia. We use Tree-sitter to parse the code data. For tokenization, we use the `tiktoken cl100k_base`[2] tokenizer. The vocabulary size is 100k.

FIM. we use context-level FIM instead of document-level FIM [50]. We train the models jointly with 70% FIM objectives and 30% L2R objectives. Within the FIM portion, 90% use our proposed AST-FIM, and 10% use Rand-FIM. We also use Rand-FIM for programming languages not supported by Tree-sitter and for any code files that fail to parse correctly. For the natural language data, we only apply the standard L2R objective without any FIM.

Hyperparameters. We train the models using 256 H100 GPUs. We train 1B models for 500B tokens. This corresponded to 120,000 steps, using a context window of 8192 tokens. The batch size was 2 contexts per GPU across the 256 GPUs. For 8B models, we train 2x or 4x more steps, corresponding to 1T or 2T tokens. We use a learning rate warmup of 2,000 steps and cosine learning rate decay to zero. The peak learning rate is 5e-4 and 1e-3 for 1B and 8B models, respectively.

5.5.2 Evaluation

We evaluate models on two types of tasks: ordinary L2R generation tasks and FIM tasks. Our goal is to check two things: **(a)** if models trained with our proposed AST-FIM method perform almost as well as standard L2R models on L2R tasks; **(b)** if AST-FIM models perform better than models trained with Rand-FIM method on FIM tasks in real-world scenarios.

L2R Tasks. We use the HumanEval+ and MBPP+ text-to-code generation benchmarks [4]. Following the original MBPP paper [49], we use a standard 3-shot setup for evaluating on MBPP+. We measure performance using the Pass@1 metric for both benchmarks. Pass@1 indicates the percentage of examples for which the model generates functionally correct code in a single attempt, evaluated using test cases.

FIM tasks. We use the SAFIM [118] and Real-FIM-Eval for FIM evaluation. SAFIM tests how well models can infill different types of code structures given both text and code context. SAFIM uses Pass@1 as the metric. Details of Real-FIM-Eval can be found in Section 5.4.

[2] `https://github.com/openai/tiktoken`

Table 5.2: **Comparison of 1B-parameter code LLMs**. All the models are trained under identical conditions. We evaluate FIM models using PSM prompt, and L2R models using a SPM prompt without special tokens (See Section 5.9.1).

	HumanEval+	**MBPP+**	**SAFIM Algorithm**	**SAFIM Control**	**SAFIM API**	**RealFIM Add**	**RealFIM Edit**
	Pass@1	Pass@1	Pass@1	Pass@1	Pass@1	PPL	PPL
L2R	15.9	34.5	26.5	31.0	48.1	1.390	1.286
Rand-FIM	11.6	31.5	28.2	36.1	56.1	1.283	1.232
AST-FIM	15.9	35.3	**33.5**	**41.2**	**60.3**	**1.269**	**1.230**

When evaluating all FIM models, we use the PSM prompt format. We also evaluate L2R models on FIM tasks, even though standard L2R models typically perform poorly on FIM tasks. We try our best to get reasonable results by using the prompt format detailed in Section 5.9.1.

We evaluate the pretrained base models directly without any further training or fine-tuning. For all generation tasks, both L2R and FIM, we use greedy decoding to generate the code.

5.6 Evaluation Results

This section presents evaluation results comparing AST-FIM against the Rand-FIM baseline and other off-the-shelf code LLMs.

5.6.1 Comparison of Pretraining Methods

We compare our proposed AST-FIM pretraining method against two baselines: random character FIM (Rand-FIM) and ordinary L2R pretraining. To ensure a fair comparison, all models use the same Llama-3 1B architecture, same datasets, and the same computational environments. The only difference lies in the pretraining objective applied. The evaluation results comparing these three methods are presented in Table 5.2. Figure 5.2 tracks the performance of each model on FIM tasks across checkpoints during pretraining.

AST-FIM Improves LLM's Capability in Practical FIM tasks. In Table 5.2, AST-FIM outperforms the Rand-FIM baseline across all subtasks within the SAFIM

benchmark. This result is expected, as AST-FIM directly trains the model to complete masked AST structures, which aligns well with the SAFIM benchmark's objective of evaluating AST structure completion. Moreover, AST-FIM achieves better performance than Rand-FIM on the Real-FIM-Eval benchmark. This is important because Real-FIM-Eval reflects practical programming activities, which often involve adding or modifying entire AST structures like statements or function calls, rather than arbitrary character spans. The training signal provided by AST-FIM is better aligned with these common coding patterns compared to Rand-FIM, leading to superior performance on realistic FIM tasks. Furthermore, Figure 5.2 shows this performance advantage is consistent throughout training. It also highlights AST-FIM's data efficiency: AST-FIM reaches performance similar to Rand-FIM after seeing only 50-70% of the training tokens.

AST-FIM Retains L2R Capability. Training models with a high FIM rate can negatively impact their performance on standard L2R code generation tasks. While the optimal FIM rate is debated (with suggestions ranging from 0.5 to 0.9 in different contexts [55, 57, 48, 50]), we used a FIM rate of 0.7 in our experiments. Table 5.2 shows that the Rand-FIM model, trained at this rate, experiences a significant performance decrease on the L2R benchmarks HumanEval+ and MBPP+ compared to the ordinary L2R model. One possible reason is that character-level random FIM can break coherent code structures and introduce noisy boundaries between the prefix, suffix, and middle parts, potentially harming the model's understanding of standard code generation. In contrast, the AST-FIM model achieves L2R performance nearly identical to the baseline L2R model on both HumanEval+ and MBPP+. This suggests that by operating on meaningful code structures, AST-FIM preserves the structural coherence of code, similar to standard L2R training. As a result, AST-FIM enables the integration of strong FIM capabilities into a language model with minimal impact on its L2R performance.

5.6.2 Evaluating AST-FIM at Scale

We now evaluate AST-FIM at a larger scale. We trained an 8B model using AST-FIM for 1 trillion tokens. We also trained a Rand-FIM model with the same architecture and data. Moreover, we trained another AST-FIM model for 2T tokens. We first compare it with a L2R baseline trained for 8T tokens where about 2T tokens are code data. Then, we compare this larger AST-FIM model with other publicly available *base* models (pretrain-only model, without any post-training) of similar size. To ensure a fair comparison of pretraining methods, we focus on *base* models, excluding all

Table 5.3: **Comparison of 6B-8B parameter code LLMs. Top:** AST-FIM (8B/1T) vs. Rand-FIM (8B/1T), trained under identical conditions. **Bottom:** AST-FIM (8B/2T) vs. L2R (8B/8T) and publicly available base models (models without post-training). The pretraining token count of each model is given if it is publicly known. L2R (8B/8T) and Llama-3.1 is evaluated using the same codebase as our models; for other models, we use Huggingface Transformers to evaluate them on FIM tasks. For HumanEval+ and MBPP+, we use their reported numbers or those reported on the EvalPlus website [4]

	HumanEval+	**MBPP+**	**SAFIM**	**SAFIM**	**SAFIM**	**RealFIM**	**RealFIM**
Subset			**Algorithm**	**Control**	**API**	**Add**	**Edit**
Metric	Pass@1	Pass@1	Pass@1	Pass@1	Pass@1	PPL	PPL
Num Examples	164	371	8,731	8,629	310	17,879	13,922
Rand-FIM (8B/1T)	32.3	59.8	50.0	56.5	62.9	1.225	1.172
AST-FIM (8B/1T)	**37.8**	**63.6**	**55.0**	**61.7**	**70.3**	**1.215**	**1.164**
L2R (8B/8T)	34.1	51.2	42.5	45.3	53.5	1.340	1.253
StarCoderBase-7B (1T)	21.3	24.4	42.2	53.4	67.8	1.226	1.185
StarCoder2-7B (4.3T)	29.9	35.4	46.2	58.4	70.6	1.227	1.184
CodeLlama-7B (1T)	35.4	37.8	34.7	53.6	46.8	1.222	1.175
Llama-3.1-8B (15T)	29.9	51.5	39.9	43.0	48.1	1.337	1.264
CodeGemma-7B (1T)	41.5	44.5	50.8	65.4	73.5	1.217	1.187
DeepSeek-Coder-6.7B (2T)	39.6	47.6	54.7	**65.8**	69.7	1.254	1.206
Qwen2.5-Coder-7B (5.5T)	**53.0**	62.9	53.0	59.2	73.9	1.212	1.167
AST-FIM (8B/2T)	42.1	**65.2**	**57.0**	63.2	**74.5**	**1.210**	**1.160**

instruction-finetuned models and distilled models. The results of these experiments are shown in Table 5.3.

AST-FIM Method Can Scale. The findings from our 1B scale experiments remain consistent at the 8B scale. The AST-FIM model with 1T tokens outperforms the Rand-FIM model trained under the same conditions. This improvement is observed in both standard L2R tasks (HumanEval+ and MBPP+) and FIM tasks (SAFIM and Real-FIM-Eval). Furthermore, training for 1T/2T tokens involves multiple epochs over the GitHub dataset. While Rand-FIM's random sampling might benefit from this multi-epoch training like data augmentation, AST-FIM's strong performance highlights its comparable or even superior data efficiency.

Performance of AST-FIM-8B is Competitive Against Similar-Sized Models Our AST-FIM model trained on 2T tokens shows competitive performance against

other base models with 6B to 8B parameters. Notably, AST-FIM achieves significantly better results than most of the compared models on both FIM benchmarks, SAFIM and Real-FIM-Eval.

5.7 Limitations

Our work has several limitations. First, we intentionally do not evaluate AST-FIM on HumanEval Single-Line Infilling, which is originally proposed in Allal et al. [56]. Single-line masking can lead to artificial tasks like completing a single closing brace. Such patterns deviate from realistic code editing scenarios and from our AST-aware training. As a result, AST-FIM might not show a performance advantage over Rand-FIM models on this benchmark. This underscores the need for realistic infilling benchmarks, where our Real-FIM-Eval is an attempt.

However, Real-FIM-Eval uses perplexity-based evaluation rather than generative metrics. While this approach provides a high signal-to-noise ratio for comparing base models (as discussed in Section 5.4), its results are not directly indicative of the models' generative performance. Future work could explore generative metrics to complement the current evaluation framework.

Finally, the AST-FIM approach itself has limitations observed in our experiments. While AST-FIM improves infilling performance, it does not show significant improvements on left-to-right (L2R) generation tasks when compared to models trained solely for L2R. This contrasts with prior work like AST-T5 [64], where AST-aware pretraining significantly improves L2R performance of an encoder-decoder model. A potential reason for this difference is that FIM pretraining for decoder-only LLMs merely permutes training inputs rather than fundamentally changing where loss function is computed.

5.8 Conclusion

Traditional random-character FIM pretraining is suboptimal for real-world applications due to a mismatch with how developers edit code. AST-FIM addresses this limitation by aligning the pretraining objective with code's syntactic structure through AST-aware subtree masking, enabling models to learn more realistic infilling patterns. The Real-FIM-Eval benchmark, based on real-world GitHub commits, validates AST-FIM's strengths in practical code completion and editing tasks. Across model scales, AST-FIM consistently outperforms Rand-FIM, showing scalability and generalizability across programming languages. As the first syntax-aware pretraining

framework for decoder-only code LLMs, AST-FIM offers a foundation for LLMs that excel in both code generation and infilling.

5.9 Appendix

5.9.1 Prompting L2R Models for FIM Evaluation

Evaluating L2R models on FIM tasks is a challenge. Although L2R models typically perform poorly on FIM tasks, we try to obtain the most reasonable results possible.

The challenge is that L2R models do not recognize the special tokens, such as `[PRE]`, `[SUF]`, and `[MID]`, which are essential for the PSM or SPM prompt formats described in Section 5.3.2. So we could not use those prompts directly for L2R models.

To address this, we experimented with two alternative prompt formats for L2R models on FIM tasks:

1. **Prefix-only:** In this format, we only provide the prefix to the model. The model's task is to continue generating code, effectively completing the missing part after the prefix.
2. **SPM without special tokens:** This format mimics the structure of SPM but avoids special tokens. The input prompt is structured as: `suffix` ↵↵ `prefix middle`

   We use two line breaks to separate the suffix and the prefix. The model is expected to generate the missing middle part immediately following the prefix in the input.

We find that the "SPM without special tokens" prompt consistently produced better results. This is because providing the suffix offers valuable context that the L2R model can leverage, even though it is not explicitly trained on FIM tasks with special tokens. Therefore, for all FIM task evaluations involving L2R models presented in this chapter, we use the "SPM without special tokens" prompt format.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation has explored the role of code structure in advancing the capabilities of Large Language Models (LLMs) for program synthesis. While the rise of LLMs has revolutionized the field, we identify a critical gap: they treat code as sequences of plain text, thereby ignoring its inherent, rich structural properties. We aim at equipping LLMs with ability to understand and leverage code structure. We believe that this is important for overcoming their limitations, particularly in complex, low-resource, or highly specialized coding scenarios.

We begin by defining code structure, primarily focusing on the syntactic structure represented by Abstract Syntax Trees (ASTs). We then establish the need for structure awareness, highlighting how current LLMs falter in tasks requiring deep syntactic understanding or dealing with less common programming languages and APIs. The ADELT case study (Chapter 2) serves as an early illustration, showing how an AST-aware decoupling strategy can significantly enhance the performance of code transpilation between deep learning frameworks, a task where standard LLMs struggle with API-level accuracy.

To rigorously measure progress in this domain, we introduce SAFIM (Chapter 3), a syntax-aware fill-in-the-middle benchmark. SAFIM provides a targeted evaluation methodology for evaluating the structural understanding of generative models, showing that pretraining strategies and data quality can be more important than model size alone. This benchmark offers a valuable tool for the community to guide and evaluate future code LLMs.

Building on these insights, we develop novel pretraining methodologies. For encoder-

decoder architectures, AST-T5 (Chapter 4) shows that leveraging ASTs through AST-Aware Segmentation and AST-Aware Subtree Corruption leads to consistent performance improvements across diverse code-related tasks, including generation, transpilation, and classification. Importantly, AST-T5 achieves this without requiring explicit structural information at inference time, allowing for seamless integration into existing workflows.

Subsequently, we address the challenge of scaling these benefits to larger, decoder-only models with AST-FIM (Chapter 5). By designing a pretraining strategy that masks complete syntactic structures during Fill-in-the-Middle (FIM) pretraining, AST-FIM significantly enhances infilling performance on both SAFIM and real-world benchmarks. This work confirms that AST-aware masking is a beneficial strategy for autoregressive models, offering high-quality training signals and improving performance on various coding tasks.

In collectively addressing the research question of how to equip LLMs to understand and leverage code structure, this dissertation has provided not only compelling evidence for the necessity of structure awareness but also practical and effective methods for its evaluation and integration into LLM pretraining. The success of ADELT, SAFIM, AST-T5, and AST-FIM highlights the potential of structure-aware approaches to build more robust, reliable, and versatile LLMs for program synthesis.

6.2 Future Work

While we have shown the benefits of AST-based structural awareness, the field of code understanding and generation remains rich with opportunities. This section outlines some potential directions:

6.2.1 Other Forms of Code Structure

This work has primarily focused on syntactic structure via ASTs. Future research could investigate the incorporation of other structural and semantic information into LLM pretraining and inference.

- **Semantic Structures:** Beyond syntax, semantic information derived from data flow graphs (DFGs), control flow graphs (CFGs), program dependence graphs (PDGs), or type systems could provide deeper understanding. Developing methods to efficiently represent and integrate these more complex structures into LLM architectures, without prohibitive computational overhead during training or inference, remains an open challenge.

- **Inter-File and Project-Level Structures:** Real-world software development often involves understanding relationships across multiple files and modules. Future work could explore techniques for LLMs to explicitly model and leverage project-level structures, such as dependency graphs, directory structures, and build configurations, to improve their performance on tasks requiring broader context. DeepSeek-Coder [57] has shown promising results of using file-dependencies in code repositories to organize training inputs. However, more controlled experiments and direct comparisons are needed before we can reach a conclusion.

- **Dynamic Structures:** Information from program execution, such as execution traces or runtime values, offers another dimension of structural insight. Investigating how such dynamic information could be captured and used by LLMs, perhaps through novel pretraining tasks or multi-modal approaches, could be a promising direction.

6.2.2 Better Evaluation Methodologies for Structural Understanding

The SAFIM benchmark and the Real-FIM-Eval perplexity-based evaluation represent progress in evaluating structural awareness. However, the community would benefit from even more comprehensive and realistic evaluation frameworks.

- **Generative Metrics for Realistic Infilling:** As noted in the limitations of AST-FIM, perplexity-based metrics on benchmarks like Real-FIM-Eval, while providing a good signal-to-noise ratio for base model comparison, do not directly measure generative performance in realistic scenarios. Future work should focus on developing robust generative metrics and benchmarks that better reflect how developers actually use infilling and code completion tools, moving beyond single-line or artificially constrained tasks.

- **Task-Specific Structural Evaluation:** Beyond general code completion or generation, evaluating structural understanding in the context of more complex downstream tasks like semantics-preserving refactoring, bug detection based on structural anomalies, or generating code that adheres to specific architectural patterns would provide deeper insights.

Bibliography

[1] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. arXiv:2203.13474 [cs], Feb 2023. doi: 10.48550/arXiv.2203.13474. URL http://arxiv.org/abs/2203.13474.

[2] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664 [cs], Mar 2021. doi: 10.48550/arXiv.2102.04664. URL http://arxiv.org/abs/2102.04664.

[3] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. arXiv:2009.10297 [cs], September 2020. doi: 10.48550/arXiv.2009.10297. URL http://arxiv.org/abs/2009.10297.

[4] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In Thirty-seventh Conference on Neural Information Processing Systems, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

[5] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. arXiv:1611.01989, March 2017. doi: 10.48550/arXiv.1611.01989. URL http://arxiv.org/abs/1611.01989.

[6] Xinyun Chen, Linyuan Gong, Alvin Cheung, and Dawn Song. PlotCoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 2169–2181, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.169. URL https://aclanthology.org/2021.acl-long.169.

[7] Gabriel Gordon-Hall. Evaluating llms on cobol, March 2024. URL https://bloop.ai/blog/evaluating-llms-on-cobol.

[8] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. VerilogEval: Evaluating large language models for Verilog code generation. arXiv:2309.07544, December 2023. doi: 10.48550/arXiv.2309.07544. URL http://arxiv.org/abs/2309.07544.

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366 [cs], Sep 2021. doi: 10.48550/arXiv.2009.08366. URL http://arxiv.org/abs/2009.08366.

[10] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. StructCoder: Structure-aware transformer for code generation. arXiv:2206.05239 [cs], May 2023. doi: 10.48550/arXiv.2206.05239. URL http://arxiv.org/abs/2206.05239.

[11] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. VerilogCoder: Autonomous Verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. arXiv:2408.08927, March 2025. doi: 10.48550/arXiv.2408.08927. URL http://arxiv.org/abs/2408.08927.

[12] Da Shen, Xinyun Chen, Chenguang Wang, Koushik Sen, and Dawn Song. Benchmarking language models for code syntax understanding. arXiv:2210.14473, October 2022. doi: 10.48550/arXiv.2210.14473. URL http://arxiv.org/abs/2210.14473.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155 [cs], Sep 2020. URL http://arxiv.org/abs/2002.08155.

[14] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859 [cs], Sep 2021. doi: 10.48550/arXiv.2109.00859. URL http://arxiv.org/abs/2109.00859.

[15] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised neural machine translation. arXiv:1710.11041 [cs], Feb 2018. URL http://arxiv.org/abs/1710.11041.

[16] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. arXiv:1409.3215 [cs], Dec 2014. URL http://arxiv.org/abs/1409.3215.

[17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. arXiv:2005.14165 [cs], July 2020. doi: 10.48550/arXiv.2005.14165. URL http://arxiv.org/abs/2005.14165.

[18] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. arXiv:1508.07909 [cs], Jun 2016. URL http://arxiv.org/abs/1508.07909.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. arXiv:1706.03762, Jun 2017. URL https://arxiv.org/abs/1706.03762v5.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805 [cs], May 2019. doi: 10.48550/arXiv.1810.04805. URL http://arxiv.org/abs/1810.04805.

[21] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. arXiv:2001.00059 [cs], Aug 2020. URL http://arxiv.org/abs/2001.00059.

[22] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. arXiv:2102.07492 [cs], Oct 2021. URL http://arxiv.org/abs/2102.07492.

[23] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. arXiv:1505.07818 [cs, stat], May 2016. URL http://arxiv.org/abs/1505.07818.

[24] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. arXiv:1406.2661 [cs, stat], Jun 2014. URL http://arxiv.org/abs/1406.2661.

[25] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. arXiv:1710.04087 [cs], Jan 2018. URL http://arxiv.org/abs/1710.04087.

[26] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. arXiv:2107.03374 [cs], Jul 2021. doi: 10.48550/arXiv.2107.03374. URL http://arxiv.org/abs/2107.03374.

[27] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. arXiv:1910.02216 [cs], October 2019. doi: 10.48550/arXiv.1910.02216. URL http://arxiv.org/abs/1910.02216.

[28] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. Kgtorrent: A dataset of python jupyter notebooks from kaggle. 2021 IEEE/ACM 18th International

Conference on Mining Software Repositories (MSR), page 550–554, May 2021. doi: 10.1109/MSR52588.2021.00072. URL http://arxiv.org/abs/2103.10558.

[29] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady, 10:707, Feb 1966. URL https://ui.adsabs.harvard.edu/abs/1966SPhD...10..707L. ADS Bibcode: 1966SPhD...10..707L.

[30] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. arXiv: 1301.3781, Jan 2013. URL https://arxiv.org/abs/1301.3781v3.

[31] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, page 651–654. Association for Computing Machinery, Aug 2013. ISBN 9781450322379. doi: 10.1145/2491411.2494584. URL https://doi.org/10.1145/2491411.2494584.

[32] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2014, page 173–184. Association for Computing Machinery, Oct 2014. ISBN 9781450332101. doi: 10.1145/2661136.2661148. URL https://doi.org/10.1145/2661136.2661148.

[33] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. arXiv:2006.03511 [cs], Sep 2020. doi: 10.48550/arXiv.2006.03511. URL http://arxiv.org/abs/2006.03511.

[34] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. arXiv:2110.06773 [cs], Feb 2022. URL http://arxiv.org/abs/2110.06773.

[35] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building code transpilers for domain-specific languages using program synthesis. European Conference on Object-Oriented Programming (ECOOP), pages 30 pages, 1247897 bytes, 2023. ISSN 1868-8969. doi: 10.4230/LIPICS.ECOOP.

2023.38. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.38.

[36] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In Hans-Juergen Boehm and Cormac Flanagan, editors, ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pages 3–14. ACM, 2013.

[37] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. ACM Trans. Graph., 38(6):204:1–204:13, 2019.

[38] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022, pages 1004–1016. ACM, 2022.

[39] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, pages 1205–1220. ACM, 2018. doi: 10.1145/3183713.3196891. URL https://doi.org/10.1145/3183713.3196891.

[40] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Tenspiler: A verified lifting-based compiler for tensor operations. arXiv:2404.18249 [cs], April 2024. doi: 10.48550/arXiv.2404.18249. URL http://arxiv.org/abs/2404.18249.

[41] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692 [cs], Jul 2019. doi: 10.48550/arXiv.1907.11692. URL http://arxiv.org/abs/1907.11692.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv:1412.6980 [cs], Jan 2017. URL http://arxiv.org/abs/1412.6980.

[43] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Hubs in space: Popular nearest neighbors in high-dimensional data. Journal of Machine Learning Research, 11(86):2487–2531, 2010. ISSN 1533-7928. URL http://jmlr.org/papers/v11/radovanovic10a.html.

[44] Herve Jegou, Cordelia Schmid, Hedi Harzallah, and Jakob Verbeek. Accurate image search using the contextual dissimilarity measure. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(1):2–11, Jan 2010. ISSN 1939-3539. doi: 10.1109/TPAMI.2008.285.

[45] Georgiana Dinu, Angeliki Lazaridou, and Marco Baroni. Improving zero-shot learning by mitigating the hubness problem. arXiv:1412.6568 [cs], Apr 2015. URL http://arxiv.org/abs/1412.6568.

[46] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs], March 2022. doi: 10.48550/arXiv.2203.02155. URL http://arxiv.org/abs/2203.02155.

[47] OpenAI. GPT-4 technical report. arXiv:2303.08774 [cs], December 2023. doi: 10.48550/arXiv.2303.08774. URL http://arxiv.org/abs/2303.08774.

[48] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. arXiv:2308.12950 [cs], August 2023. doi: 10.48550/arXiv.2308.12950. URL http://arxiv.org/abs/2308.12950.

[49] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. arXiv:2108.07732 [cs], Aug 2021. doi: 10.48550/arXiv.2108.07732. URL http://arxiv.org/abs/2108.07732.

[50] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models

to fill in the middle. arXiv:2207.14255 [cs], July 2022. doi: 10.48550/arXiv.2207.14255. URL http://arxiv.org/abs/2207.14255.

[51] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The Stack: 3 TB of permissively licensed source code. arXiv:2211.15533 [cs], November 2022. doi: 10.48550/arXiv.2211.15533. URL http://arxiv.org/abs/2211.15533.

[52] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways. arXiv:2204.02311 [cs], Oct 2022. doi: 10.48550/arXiv.2204.02311. URL http://arxiv.org/abs/2204.02311.

[53] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code. arXiv:2202.13169 [cs], May 2022. doi: 10.48550/arXiv.2202.13169. URL http://arxiv.org/abs/2202.13169.

[54] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. arXiv:2204.05999 [cs], April 2023. doi: 10.48550/arXiv.2204.05999. URL http://arxiv.org/abs/2204.05999.

[55] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim,

Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you! arXiv:2305.06161 [cs], December 2023. doi: 10.48550/arXiv.2305.06161. URL `http://arxiv.org/abs/2305.06161`.

[56] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. SantaCoder: don't reach for the stars! arXiv:2301.03988 [cs], February 2023. doi: 10.48550/arXiv.2301.03988. URL `http://arxiv.org/abs/2301.03988`.

[57] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. arXiv:2401.14196 [cs], January 2024. doi: 10.48550/arXiv.2401.14196. URL `http://arxiv.org/abs/2401.14196`.

[58] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm

Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, and Others. Gemini: A family of highly capable multimodal models. arXiv:2312.11805 [cs], December 2023. doi: 10.48550/arXiv.2312.11805. URL `http://arxiv.org/abs/2312.11805`.

[59] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. GLM: General language model pretraining with autoregressive blank infilling. arXiv:2103.10360 [cs], March 2022. doi: 10.48550/arXiv.2103.10360. URL `http://arxiv.org/abs/2103.10360`.

[60] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on humaneval-x. arXiv:2303.17568 [cs], March 2023. doi: 10.48550/arXiv.2303.17568. URL `http://arxiv.org/abs/2303.17568`.

[61] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. arXiv:2103.06333 [cs], Apr 2021. doi: 10.48550/arXiv.2103.06333. URL `http://arxiv.org/abs/2103.06333`.

[62] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. Science, 378(6624):1092–1097, December 2022. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.abq1158. URL `http://arxiv.org/abs/2203.07814`. arXiv:2203.07814 [cs].

[63] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. CodeT5+: Open code large language models for code understanding and generation. arXiv:2305.07922 [cs], May 2023. doi: 10.48550/arXiv.2305.07922. URL `http://arxiv.org/abs/2305.07922`.

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

[cs], January 2024. doi: 10.48550/arXiv.2401.03003. URL http://arxiv.org/abs/2401.03003.

[65] Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E. Gonzalez, and Ion Stoica. Rethinking benchmark and contamination for language models with rephrased samples. arXiv:2311.04850 [cs], November 2023. doi: 10.48550/arXiv.2311.04850. URL http://arxiv.org/abs/2311.04850.

[66] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. arXiv:2208.08227 [cs], December 2022. doi: 10.48550/arXiv.2208.08227. URL http://arxiv.org/abs/2208.08227.

[67] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. arXiv:2210.14868 [cs], March 2023. doi: 10.48550/arXiv.2210.14868. URL http://arxiv.org/abs/2210.14868.

[68] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. arXiv:2105.09938 [cs], November 2021. doi: 10.48550/arXiv.2105.09938. URL http://arxiv.org/abs/2105.09938.

[69] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. arXiv:2211.11501 [cs], November 2022. doi: 10.48550/arXiv.2211.11501. URL http://arxiv.org/abs/2211.11501.

[70] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks. arXiv:2212.09248 [cs], December 2022. doi: 10.48550/arXiv.2212.09248. URL http://arxiv.org/abs/2212.09248.

[71] Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. ToolCoder: Teach code generation models to use api search tools. arXiv:2305.04032 [cs], September 2023. doi: 10.48550/arXiv.2305.04032. URL http://arxiv.org/abs/2305.04032.

[72] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. arXiv:2112.02969 [cs], December 2021. doi: 10.48550/arXiv.2112.02969. URL http://arxiv.org/abs/2112.02969.

[73] Linyuan Gong, Jiayi Wang, and Alvin Cheung. ADELT: Transpilation between deep learning frameworks. arXiv:2303.03593 [cs], May 2024. doi: 10.48550/arXiv.2303.03593. URL http://arxiv.org/abs/2303.03593.

[74] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. arXiv:2305.15334 [cs], May 2023. doi: 10.48550/arXiv.2305.15334. URL http://arxiv.org/abs/2305.15334.

[75] Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. arXiv:2306.03091 [cs], October 2023. doi: 10.48550/arXiv.2306.03091. URL http://arxiv.org/abs/2306.03091.

[76] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. arXiv:2212.10481 [cs], May 2023. doi: 10.48550/arXiv.2212.10481. URL http://arxiv.org/abs/2212.10481.

[77] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-Bench: Can language models resolve real-world Github issues? arXiv:2310.06770 [cs], October 2023. doi: 10.48550/arXiv.2310.06770. URL http://arxiv.org/abs/2310.06770.

[78] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. arXiv:2206.12839 [cs], June 2023. doi: 10.48550/arXiv.2206.12839. URL http://arxiv.org/abs/2206.12839.

[79] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. arXiv:2303.12570 [cs], October 2023. doi: 10.48550/arXiv.2303.12570. URL http://arxiv.org/abs/2303.12570.

[80] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. arXiv:2212.10007 [cs], May 2023. doi: 10.48550/arXiv.2212.10007. URL http://arxiv.org/abs/2212.10007.

[81] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv:1910.10683 [cs, stat], Jul 2020. doi: 10.48550/arXiv.1910.10683. URL http://arxiv.org/abs/1910.10683.

[82] Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xCodeEval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. arXiv:2303.03004 [cs], November 2023. doi: 10.48550/arXiv.2303.03004. URL http://arxiv.org/abs/2303.03004.

[83] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv:2302.11382 [cs], February 2023. doi: 10.48550/arXiv.2302.11382. URL http://arxiv.org/abs/2302.11382.

[84] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. arXiv:2310.11324 [cs], October 2023. doi: 10.48550/arXiv.2310.11324. URL http://arxiv.org/abs/2310.11324.

[85] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts. arXiv:2401.04088 [cs], January 2024. doi: 10.48550/arXiv.2401.04088. URL http://arxiv.org/abs/2401.04088.

[86] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo

de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. arXiv:2306.11644 [cs], October 2023. doi: 10.48550/arXiv.2306.11644. URL http://arxiv.org/abs/2306.11644.

[87] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering code large language models with Evol-Instruct. arXiv:2306.08568 [cs], June 2023. doi: 10.48550/arXiv.2306.08568. URL http://arxiv.org/abs/2306.08568.

[88] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is all you need. arXiv:2312.02120 [cs], December 2023. doi: 10.48550/arXiv.2312.02120. URL http://arxiv.org/abs/2312.02120.

[89] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. arXiv:2302.13971 [cs], Feb 2023. doi: 10.48550/arXiv.2302.13971. URL http://arxiv.org/abs/2302.13971.

[90] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. arXiv:1808.09588 [cs], Aug 2018. doi: 10.48550/arXiv.1808.09588. URL http://arxiv.org/abs/1808.09588.

[91] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. arXiv:1812.08693 [cs], May 2019. doi: 10.48550/arXiv.1812.08693. URL http://arxiv.org/abs/1812.08693.

[92] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. arXiv:1909.03496 [cs, stat], Sep 2019. doi: 10.48550/arXiv.1909.03496. URL http://arxiv.org/abs/1909.03496.

[93] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In 2014 IEEE International Conference on Software Maintenance and Evolution, page 476–480, Sep 2014. doi: 10.1109/ICSME.2014.77.

[94] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv:1910.13461 [cs, stat], Oct 2019. doi: 10.48550/arXiv.1910.13461. URL http://arxiv.org/abs/1910.13461.

[95] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In The International Conference on Learning Representations (ICLR) 2019, Sep 2018. URL https://openreview.net/forum?id=H1gfOiAqYm.

[96] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis. arXiv:2107.00101 [cs], Jun 2021. URL https://arxiv.org/abs/2107.00101.

[97] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. arXiv:2301.13816 [cs], Jan 2023. URL https://arxiv.org/abs/2301.13816.

[98] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, page 4159–4165, July 2018. doi: 10.24963/ijcai.2018/578. URL http://arxiv.org/abs/1711.09573.

[99] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. arXiv:2003.13848 [cs], March 2021. doi: 10.48550/arXiv.2003.13848. URL http://arxiv.org/abs/2003.13848.

[100] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. arXiv:2103.11318 [cs], March 2021. doi: 10.48550/arXiv.2103.11318. URL http://arxiv.org/abs/2103.11318.

[101] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. arXiv:1910.00577 [cs, stat], July 2020. doi: 10.48550/arXiv.1910.00577. URL http://arxiv.org/abs/1910.00577.

[102] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. arXiv:1704.07535 [cs, stat], April 2017. doi: 10.48550/arXiv.1704.07535. URL http://arxiv.org/abs/1704.07535.

[103] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. arXiv:1711.00740 [cs], Nov 2017. URL https://arxiv.org/abs/1711.00740.

[104] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. arXiv:2205.14135 [cs], June 2022. doi: 10.48550/arXiv.2205.14135. URL http://arxiv.org/abs/2205.14135.

[105] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436 [cs, stat], Jun 2020. doi: 10.48550/arXiv.1909.09436. URL http://arxiv.org/abs/1909.09436.

[106] Stephen H. Bach, Victor Sanh, Zheng-Xin Yong, Albert Webson, Colin Raffel, Nihal V. Nayak, Abheesht Sharma, Taewoon Kim, M. Saiful Bari, Thibault Fevry, Zaid Alyafeai, Manan Dey, Andrea Santilli, Zhiqing Sun, Srulik Ben-David, Canwen Xu, Gunjan Chhablani, Han Wang, Jason Alan Fries, Maged S. Alshaibani, Shanya Sharma, Urmish Thakker, Khalid Almubarak, Xiangru Tang, Dragomir Radev, Mike Tian-Jian Jiang, and Alexander M. Rush. PromptSource: An integrated development environment and repository for natural language prompts. arXiv:2202.01279 [cs], March 2022. doi: 10.48550/arXiv.2202.01279. URL http://arxiv.org/abs/2202.01279.

[107] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M. Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Tali Bers, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. Multitask prompted training enables zero-shot task generalization. arXiv.org, Oct 2021. URL https://arxiv.org/abs/2110.08207v3.

[108] Ben Wang and Aran Komatsuzaki. GPT-J-6B: 6B JAX-based Transformer, Jun 2021. URL https://arankomatsuzaki.wordpress.com/2021/06/04/gpt-j/.

[109] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5297715.

[110] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. CodeGen2: Lessons for training LLMs on programming and natural

languages. arXiv:2305.02309 [cs], July 2023. doi: 10.48550/arXiv.2305.02309. URL http://arxiv.org/abs/2305.02309.

[111] BigScience. Bigscience Language Open-science Open-access Multilingual (BLOOM), May 2021. URL https://huggingface.co/bigscience/bloom.

[112] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models. arXiv:2205.01068 [cs], June 2022. doi: 10.48550/arXiv.2205.01068. URL http://arxiv.org/abs/2205.01068.

[113] CodeGemma Team. CodeGemma: Open code models based on Gemma. arXiv:2406.11409, June 2024. doi: 10.48550/arXiv.2406.11409. URL http://arxiv.org/abs/2406.11409.

[114] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and the stack v2: The next generation. arXiv:2402.19173, 2024. URL https://arxiv.org/abs/2402.19173.

[115] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. arXiv:2409.12186, November 2024. doi: 10.48550/arXiv.2409.12186. URL http://arxiv.org/abs/2409.12186.

[116] MistralAI, 2025. URL https://mistral.ai/news/codestral-2501.

[117] Wayne Chi, Valerie Chen, Anastasios Nikolas Angelopoulos, Wei-Lin Chiang, Aditya Mittal, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. Copilot arena: A platform for code LLM evaluation in the wild. arXiv:2502.09328, February 2025. doi: 10.48550/arXiv.2502.09328. URL http://arxiv.org/abs/2502.09328.

[118] Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of LLMs on syntax-aware code fill-in-the-middle tasks. arXiv:2403.04814, June 2024. doi: 10.48550/arXiv.2403.04814. URL http://arxiv.org/abs/2403.04814.

[119] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from Github Copilot. arXiv:2302.06590, February 2023. doi: 10.48550/arXiv.2302.06590. URL http://arxiv.org/abs/2302.06590.

[120] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. SpanBERT: Improving pre-training by representing and predicting spans. arXiv:1907.10529, January 2020. doi: 10.48550/arXiv.1907.10529. URL http://arxiv.org/abs/1907.10529.

[121] Chris Donahue, Mina Lee, and Percy Liang. Enabling language models to fill in the blanks. arXiv:2005.05339, September 2020. doi: 10.48550/arXiv.2005.05339. URL http://arxiv.org/abs/2005.05339.

[122] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. arXiv:2401.03065, January 2024. doi: 10.48550/arXiv.2401.03065. URL http://arxiv.org/abs/2401.03065.

[123] Lovish Madaan, Aaditya K. Singh, Rylan Schaeffer, Andrew Poulton, Sanmi Koyejo, Pontus Stenetorp, Sharan Narang, and Dieuwke Hupkes. Quantifying variance in evaluation benchmarks. arXiv:2406.10229, June 2024. doi: 10.48550/arXiv.2406.10229. URL http://arxiv.org/abs/2406.10229.