entropy

MDPI

Review

Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review

Man-Fai Wong [1], Shangxin Guo [2], Ching-Nam Hang [1], Siu-Wai Ho [3] and Chee-Wei Tan [4,*]

[1] Department of Computer Science, City University of Hong Kong, Hong Kong, China; mfwong29-c@my.cityu.edu.hk (M.-F.W.); cnhang3-c@my.cityu.edu.hk (C.-N.H.)
[2] Shenzhen Research Institute, City University of Hong Kong, Shenzhen 518057, China; sxguo2-c@my.cityu.edu.hk
[3] Teletraffic Research Centre, University of Adelaide, Adelaide, SA 5005, Australia; siuwai.ho@adelaide.edu.au
[4] School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore
* Correspondence: cheewei.tan@ntu.edu.sg

Abstract: This paper provides a comprehensive review of the literature concerning the utilization of Natural Language Processing (NLP) techniques, with a particular focus on transformer-based large language models (LLMs) trained using Big Code, within the domain of AI-assisted programming tasks. LLMs, augmented with software naturalness, have played a crucial role in facilitating AI-assisted programming applications, including code generation, code completion, code translation, code refinement, code summarization, defect detection, and clone detection. Notable examples of such applications include the GitHub Copilot powered by OpenAI's Codex and DeepMind AlphaCode. This paper presents an overview of the major LLMs and their applications in downstream tasks related to AI-assisted programming. Furthermore, it explores the challenges and opportunities associated with incorporating NLP techniques with software naturalness in these applications, with a discussion on extending AI-assisted programming capabilities to Apple's Xcode for mobile software development. This paper also presents the challenges of and opportunities for incorporating NLP techniques with software naturalness, empowering developers with advanced coding assistance and streamlining the software development process.

Keywords: software naturalness; large language models; AI-assisted programming

Citation: Wong, M.-F.; Guo, S.; Hang, C.-N.; Ho, S.-W.; Tan, C.-W. Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review. *Entropy* **2023**, *25*, 888. https://doi.org/10.3390/e25060888

Academic Editor: Lei Wang

Received: 26 April 2023
Revised: 25 May 2023
Accepted: 25 May 2023
Published: 1 June 2023

1. Introduction

The advent of Big Code has become increasingly relevant in today's software development landscape as the size and complexity of software systems continue to grow [1]. Big Code refers to the vast collection of online software artifacts such as source code repositories, bug databases, and code snippets. It represents a wealth of knowledge and experience that researchers can draw upon to improve the quality and efficiency of their own projects. The goal of Big Code is to build tools and techniques that can assist software engineers to analyze, understand, and make predictions about large codebases in a scalable and efficient manner. Big Code also has the potential to revolutionize artificial intelligence (AI) development by unitizing Big Code data. The development of statistical programming systems involves the utilization of advanced programming languages, powerful machine learning techniques such as large language models (LLMs), and natural language processing (NLP) techniques based on the software naturalness hypothesis [2]. This hypothesis posits that computer programs written in diverse programming languages can be comprehended and manipulated similarly to NLP's treatment of human natural languages.

By employing this combination of tools, probabilistic models of extensive codebases can be constructed. These systems query a probabilistic model and calculate the most probable predictions to solve a specific challenge [3], which are then presented to the

developer. In other words, the programming language is regarded as the natural language for the NLP techniques in this study. There are several crucial areas of fundamental research focused on advancing probabilistic models of "Big Code" using statistical and machine learning methodologies. By considering source code as a series of tokens and leveraging the inherent patterns and structures within vast code repositories, NLP techniques can be developed to enhance AI-assisted programming tasks, including code generation, code completion, code refinement, code summarization, defect detection, and clone detection.

AI-assisted programming can enable software engineers to work more efficiently and effectively [4], especially in situations where complex algorithms are being used that involve large amounts of code (i.e., Big Code regime). It also strikes a balance between productivity and ensuring safety, security, and reliability within the programming development environment [5]. In fact, this can even lead to the development of AI-based predictive analysis that allows human developers to more easily interact with code using natural language commands and queries as part of the software development process [6]. AI-based predictive analysis [7] can also more accurately anticipate potential issues throughout the software development life cycle and flag critical incidents [8] before they occur [9,10].

Several recent reviews have explored specific topics related to LLMs, such as fairness and bias [11], interpretability [12], explainability [13], and privacy preservation [14]. However, this review focuses primarily on language models with software naturalness. In Table 1, a detailed comparison of other reviews that have examined related topics is provided. This review also delves into the analysis of the publicly available Big Code dataset, which is designed to assist programming with AI. This review addresses the process of using language models for assessing software naturalness and examines the concept of evaluating language models using entropy. Additionally, the latest developments in AI-assisted programming using transformer-based LLMs trained on Big Code are explored, and both the generation and comprehension aspects are discussed. The review concludes with the open challenges and opportunities in AI-assisted programming. This review paper highlights the unique contributions of this review in comparison to existing reviews.

Reviews have emphasized the significance of AI-assisted programming, leading to significant advancements in this critical field of study. However, the essential components of AI-assisted programming have been presented separately, resulting in a fragmented understanding of the topic. Despite this, these independent studies have created an opportunity to view AI-assisted programming from a more comprehensive perspective. In light of this, our survey aims to provide a more structured approach to framing AI-assisted programming that extends beyond the examination of individual research topics. By doing so, this review paper hopes to offer a more comprehensive understanding of this field, highlighting the interdependencies between different areas of research.

Table 1. Comparison of surveys on language models in software naturalness.

Title	Year	Focus Area
A Survey of Machine Learning for Big Code and Naturalness [15]	2019	Big Code and Naturalness
Software Vulnerability Detection Using Deep Neural Networks: A Survey [16]	2020	Security
A Survey on Machine Learning Techniques for Source Code Analysis [17]	2021	Code Analysis
Deep Security Analysis of Program Code: A Systematic Literature Review [18]	2022	Security
A Survey on Pretrained Language Models for Neural Code Intelligence [19]	2022	Code Summarization and Generation, and Translation
Deep Learning Meets Software Engineering: A Survey on Pre-trained Models of Source Code [20]	2022	Software Engineering
Software as Storytelling: A Systematic Literature Review [21]	2023	Storytelling
Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing [22]	2023	Prompt-based Learning

The remainder of this review article is structured as follows. Section 2 provides an overview of the background knowledge in Big Code and software naturalness, covering topics such as the available dataset, tokenization process, existing language models, and the

measurement of language models using entropy. Section 3 explores recent applications of LLMs trained with Big Code in AI-assisted programming tasks. Section 4 discusses the potential challenges and opportunities associated with LLMs in this context. Finally, Section 5 concludes the study and outlines possible directions for future work in this field.

2. Background

2.1. Main Big Code Dataset

Researchers have successively released a large amount of Big Code to train LLMs. Most datasets used to train LLMs can be applied into different tasks such as code generation and code summarization. LLMs use unsupervised learning and require large amounts of high-quality and diverse data to achieve high accuracy and generalization in their predictions. Access to large-scale, high-quality, diverse, and representative datasets is essential for developing high-performing LLMs on software naturalness. The datasets found in the literature are described in Table 2, which were accessed on 18 May 2023.

Table 2. Summary of public datasets used on Big Code. All URLs were accessed on 18 May 2023.

Dataset Name	Year	Sample Size	Language(s)	Supported Task(s)	Online URL
GitHub Java Corpus [23]	2013	14.7K	Java	Code Completion	https://groups.inf.ed.ac.uk/cup/javaGithub/
Description2Code [24]	2016	7.6K	Java, C#	Code Generation, Code Summarization	https://github.com/ethancaballero/description2code
BigCloneBench [25]	2015	5.5K	Java	Defect Detection, Clone Detection	https://github.com/clonebench/BigCloneBench
CodRep [26]	2018	58K	Java	Code Refinement, Defect Detection	https://github.com/ASSERT-KTH/CodRep-competition
CONCODE [27]	2018	104K	Java	Code Generation	https://github.com/sriniiyer/concode
WikiSQL [28]	2018	87K	SQL	Code Summarization	https://github.com/salesforce/WikiSQL
Bugs2Fix [29]	2019	122K	Java	Defect Detection, Code Refinement	https://sites.google.com/view/learning-fixes
Devign [30]	2019	26.4K	C	Code Generation, Defect Detection	https://sites.google.com/view/devign
CodeSearchNet [31]	2019	2M	Python, Javascript, Ruby, Go, Java, PHP	Code Generation, Code Summarization, Code Translation	https://github.com/github/CodeSearchNet
The Pile [32]	2020	211M	Python	Coder Generation	https://pile.eleuther.ai
CodeNet [33]	2021	13M	C++, C, Python, Java	Code Generation, Code Refinement	https://github.com/IBM/Project_CodeNet
CodeXGLUE [34]	2021	176K	Python, Java, PHP, JavaScript, Ruby, Go	Code Generation, Code Completion, Code Summarization, Defect Detection	https://github.com/microsoft/CodeXGLUE
HumanEval [35]	2021	164	Python	Code Generation	https://github.com/openai/human-eval
APPS [36]	2021	10K	Python	Code Generation	https://github.com/hendrycks/apps
Codeparrot [37]	2022	22M	Python	Code Generation	https://hf.co/datasets/transformersbook/codeparrot
CodeContests [38]	2022	13.6K	C++, Java, JavaScript, C# and 8 more	Code Generation	https://github.com/deepmind/code_contests
CERT [39]	2022	5.4M	Python	Code Generation	https://github.com/microsoft/PyCodeGPT
InCoder [40]	2022	670K	Python, JavaScript, HTML and 24 more	Code Generation, Code Summarization	https://github.com/dpfried/incoder
PolyCoder [41]	2022	1K	C, C++, Java, JavaScript, C#, Go and 6 more	Code Generation	https://github.com/VHellendoorn/Code-LMs
ExecEval [42]	2023	58K	Ruby, Javascript, Go, C++, C and 6 more	Code Sumarization, Code Generation, Code Translation	https://github.com/ntunlp/xCodeEval

2.2. Tokenization

Figure 1 illustrates the pipeline of language models on software naturalness. Similar to other neural networks and raw text, language models cannot process source code directly, so the first step of the standard pipeline is to convert the code inputs into numbers of which the model can make sense. To do this, a tokenizer can be used to split the input into code syntax keyword, variables, or symbols (similar to punctuation) that are called tokens. Each token is mapped to an integer in the next step. These tokens typically correspond to words, punctuation marks, or other meaningful elements of the text. Tokenization is an important step in many NLP tasks, as it allows machine learning algorithms to process and analyze text in a more efficient and meaningful way. Some popular tokenizers are available to be used directly such as Byte-Pair Encoding (BPE) [43] and RoBERTa [44].

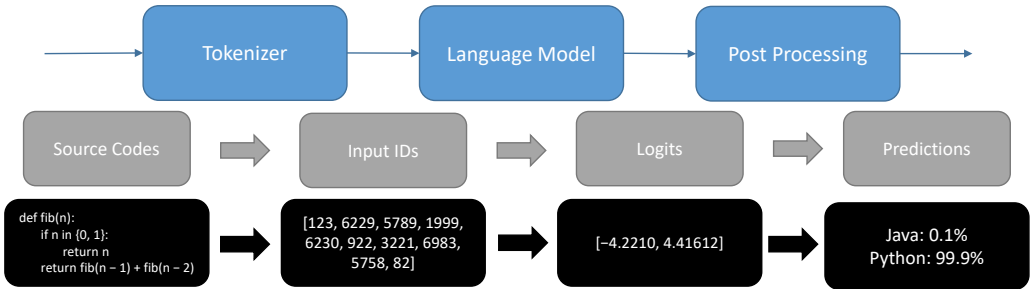


Figure 1. Pipeline of language models on software naturalness.

In the tokenization process, each token is assigned a unique identifier or index which can be used to represent the token in a numerical format that can be understood by machine learning models. Different tokenization strategies may be used depending on the specific task at hand, such as splitting text into words, phrases, or even individual characters. One common challenge in tokenization is dealing with ambiguity or variability in the text. For example, words may have different meanings depending on the context in which they appear, or may be misspelled or abbreviated in unpredictable ways. There are various techniques that can be used to address these challenges, such as using contextual information or statistical models to help disambiguate the text.

2.3. Language Models on Software Naturalness

In this section, some of the leading transformer-based language models are presented. Figure 2 displays the timeline of the evolution of LLMs since 2018.

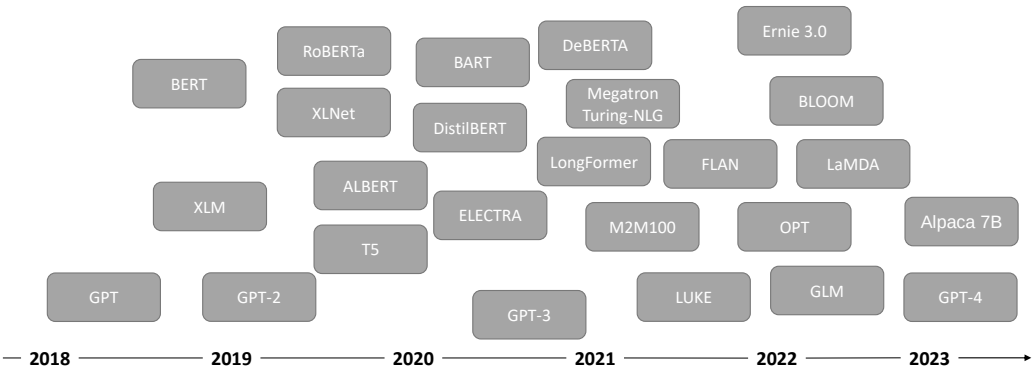


Figure 2. Timeline for the development of transformer-based large language models.

Table 3 provides a summary of transformer-based language models used in AI-assisted programming. Transformer-based models are a type of neural network architecture used in NLP and other machine learning tasks. The transformer maintains a similar architecture as the encoder–decoder architecture shown in Figure 3, but the models use a self-attention mechanism to weigh the importance of different parts of the input sequence, allowing them

to capture dependencies between all parts of the sequence, as shown in Figure 4. They can be parallelized more easily than previous models, resulting in faster training and lower inference times. The transformer model is one of the most well-known transformer-based models and has been used in various NLP tasks. Recently, large transformer-based models such as GPT-4 [45] and LLaMA [46] have achieved state-of-the-art performance in many benchmarks. The transformer's ability to capture long-range dependencies is heavily reliant on dot-product attention with softmax normalization, leading to a quadratic space and time complexity in relation to sequence length, which can be a hindrance for longer inputs. This study focuses on transformer-based models for AI-assisted programming tasks.

Table 3. Summary of language models using transformers for AI-assisted programming.

Model	Type	AI-Assisted Programming Tasks
Encoder-only	Understanding	Code Summarization, Code Translation
Decoder-only	Generation	Code Generation, Code Completion
Encoder–decoder	Generation and Understanding	Code Generation, Code Refinement, Defect Detection, Clone Detection

Encoder–decoder models [47] refer to sequence-to-sequence models, utilizing both components of the transformer architecture [48]. The encoder's attention layers can access all words in the input sentence at each stage, while the decoder's attention layers can only access the words preceding a given word in the input. Sequence-to-sequence models such as BART [49], T5 (Text-to-Text Transfer Transformer) [50], and TreeGen [51] are well-suited for tasks that involve generating new text based on an input, such as code generation, code refinement, defect detection, and clone detection, for AI-assisted programming tasks.

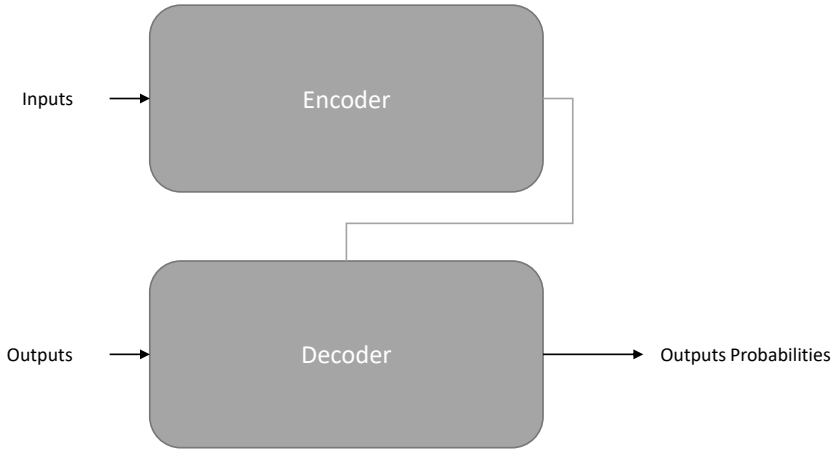


Figure 3. Encoder–decoder architecture. The model is primarily composed of two blocks: The encoder receives an input and builds a representation of its features, while the decoder uses the encoder's representation along with other inputs to generate a target sequence.

Encoder-only models, also known as autoencoders, use only an encoder network to transform input data into a compressed representation. They are commonly used in unsupervised learning tasks such as dimensionality reduction and anomaly detection in NLP tasks. In the past, code embedding approaches could be utilized to obtain the representation from the input data such as Neural Network Language Model [52], Code2Vec [53], ELMo [54], TextRank [55], and GGNN [56]. For AI-assisted programming tasks, they are used for understanding tasks to learn useful representations with the BERT [57] and RoBERTa [44] of data in an unsupervised manner, which can be used as features for downstream tasks such as code translation and code summarization.

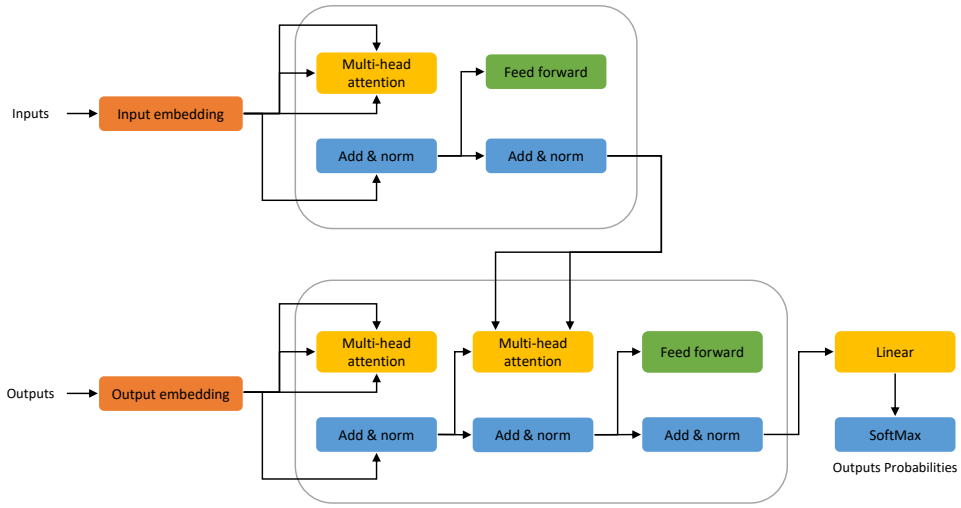


Figure 4. Transformer architecture. The transformer architecture retains a similar structure to that of the encoder–decoder architecture. The encoder considers all words in a sentence, while the decoder works sequentially. Once the initial words are predicted, they are used to generate subsequent words. The attention layers in the encoder consider all the words in a sentence, while the decoder works sequentially and can only focus on the words it has already translated.

Decoder-only models, also known as autoregressive models, are a type of neural network architecture used in natural language processing tasks such as GPT-2 [58], GPT-3 [59], GPT-J [60], Reformer [61], and GPT-Neo [62], which use the decoder to predict the next token output given all previous tokens. They rely solely on a decoder network to generate output text, predicting the probability distribution of the next token given the previously generated tokens. Although they are simpler and more efficient than encoder–decoder models, they may not be as effective in tasks requiring a deeper understanding of the input–output sequence relationship. Nevertheless, they are still widely used in various natural language processing tasks for AI-assisted programming, such as code generation and code completion, and have demonstrated impressive performance in several benchmarks.

2.4. Measurement of Language Models with Entropy

Language models on software naturalness are trained on large code corpora and used to predict the next token in the code given its context. Mathematically, assuming a set of program tokens $\mathbb{T}$ and a set of program sequences $\mathbb{S}$, the set of possible systems is $S \subset \mathbb{S}$. A language model is a probability distribution $p(.)$ over systems $s \in S$:

$$\forall s \in S[0 < p(s) < 1] \wedge \sum_{s \in S} p(s) = 1. \quad (1)$$

An estimated language model known as a pre-trained language model [63] is created by computing a maximum-likelihood estimation (MLE) of the parameter of a suitably chosen parametric distribution $p(\cdot)$ given a corpus C of programs $C \subseteq S$. This process is described in Section 2.2. The tokenization of the code is defined by the programming language to estimate the probability distribution of code tokens given the preceding context. It uses this information to make predictions or decisions in the software engineering tasks. The models are trained to predict the probability distribution of words in a sequence, based on the previous words in that sequence [64]. The language model is typically constructed using N-gram models, which have a long history in statistical language modeling and are widely used for estimating the probability distribution of words or characters in a text sequence [65,66]. This was the standard method before the development of word vectors and distributed representations of language using Recurrent Neural Networks (RNN) [67]. Given a system s with a sequence of tokens $\{W_1, W_2, \ldots W_n\}$, N-gram models can estimate

the likelihood of tokens following other tokens. As a result, the model can estimate the probability of s by multiplying a series of conditional probabilities:

$$p(s) = p(W_1)p(W_2|a_1)p(W_3|W_1W_2)\dots p(W_n|W_1\dots W_{n-1}). \quad (2)$$

An N-gram model captures the co-occurrence patterns of words or characters in the text. Mathematically, an N-gram model can be represented as a set of N-grams, each represented as a tuple of n items and their associated probabilities. The probability of an N-gram can be estimated by the MLE based on the frequency of occurrence of the N-gram in a given training corpus. This also assumes a Markov property, i.e., token occurrences are influenced only by a limited prefix length of n. Thus, for example, in a 3-gram $(n = 3)$ model:

$$p(W_i|W_1\dots W_{i-1}) \cong p(W_i|W_{i-2}W_{i-1}). \quad (3)$$

The probability of a word W_i given its preceding word W_{i-1} can be estimated:

$$p(W_i|W_{i-1}) = count(W_{i-1}, W_i)/count(W_{i-1}), \quad (4)$$

where $count(W_{i-1}, W_i)$ is the number of times the 3-gram (W_{i-1}, W_i) appears in the training corpus, and $count(W_{i-1})$ is the number of times the word W_{i-1} appears in the training corpus. The models have achieved great success in recent years and have been a driving force behind recent advancements in NLP. The performance of the technique depends on the quality of the language model and the ability of the model to accurately reflect the patterns and structures of the target data. Therefore, much research effort has been devoted to improving the quality of language models for these tasks, including developing better training algorithms, larger training corpora, and better evaluation metrics.

A representative corpus of repetitive and highly predictable programs is utilized to capture regularities within the corpus in order to evaluate the naturalness of software language models. By estimating the language model from this representative corpus, it can predict the contents of new programs with high confidence, thereby minimizing the surprise associated with the new program. In NLP, this idea is often measured using perplexity or cross-entropy (log-transformed version). Given a program $p = \{w_1, w_2, \dots, w_n\}$, of length n, and a language model Θ, it assumes that the probability of the programs estimated by the model is p_Θ, and, thus, the cross-entropy $H_\Theta(p)$ can be measured:

$$H_\Theta(p) = -\frac{1}{n}\log p_\Theta(w_1, w_2, \dots, w_n) \quad (5)$$

and a formulation can be derived from Equation (2):

$$H_\Theta(p) = -\frac{1}{n}\sum_{i=1}^{n}\log p_\Theta(w_i|w_1, w_2, \dots, w_{i-1}). \quad (6)$$

The entropy rate of a language model is utilized to assess the naturalness of the generated text [68]. It can be computed by taking the negative logarithm of the probability of each generated token. An effective model should have low entropy for the majority of programs, assigning higher probabilities (i.e., values closer to 1) to most words in the program, thereby resulting in lower absolute log values. In practice, this involves using techniques such as maximum likelihood estimation or neural networks to estimate the parameters. The final model can then be used to make predictions by calculating the probability of a given sequence of words. Estimating entropy from empirical data has been an interesting area in information theory for AI-assisted programming [69]. For example, a method for estimating entropy with a confidence interval was proposed in [70]. Another method for estimating the entropy and redundancy of a language was provided in [68]. A model weighting principle based on the minimum description length principle was applied in [71] to develop a direct

estimator of the entropy rate. The estimator can be used to estimate a Bayesian confidence interval for the entropy rate using Monte Carlo techniques. Techniques for estimating the entropy rate have been reviewed in [72]. Analytical results of estimators for entropy and mutual information can be found in [73].

3. AI-Assisted Programming Tasks

There are two main categories of AI-assisted programming tasks related to software naturalness: generation and understanding. The former includes code generation, code completion, code translation, code refinement, and code summarization. The latter is concerned with understanding code and includes defect detection and clone detection. Researchers have made significant efforts to enhance the quality of language models for these tasks by improving pre-training schemes, increasing the size of training corpora, developing better fine-tuning datasets, and using improved evaluation metrics. The frameworks and tools developed for these specific tasks are discussed in this section, and a summary of all the frameworks reviewed is presented in Table 4.

3.1. Code Generation

Program synthesis, also known as source code generation, is the process of automatically generating source code from a programming language based on user-specified constraints [74,75]. This study focuses on text-to-code generation for code generation, while code-to-code generation is referred to as code translation, which is discussed in Section 3.3. The history of code generation dates back to the use of theorem provers to construct a proof of user-provided specifications and extract corresponding logical programs [76,77]. With the increasing popularity of deep learning methods, neural methods, including Long Short–Term Memory (LSTM) [78] and Recursive–Reverse–Recursive Neural Network [79], have been adopted to generate output programs with specific inductive biases given sufficient program samples. More recently, transformer-based LLMs such as GPT-3 [59] and T5 [50] have shown impressive performance in code generation tasks by leveraging contextual representations learned from large amounts of code, as well as public code sources and natural language data, to improve program synthesis. These approaches incorporate systematic pre-training and fine-tuning tasks to develop a deep understanding of code structure and meaning, making them well-suited for software development tasks. To evaluate the models for code generation tasks, different metrics are available such as *pass@k* [35], which measures the percentage of problems solved using k generated programs per problem, BLEU-4 [80], and exact match accuracy on program synthesis benchmarks such as APPS [36], MBPP [81], and CodeBLEU [50], which consider both syntactic and semantic matches based on code structure in addition to N-gram matches.

3.2. Code Completion

Code completion, also known as autocompletion, is a software development feature that suggests possible code completions as a programmer types [82]. Its goal is to save time and reduce errors by providing suggestions for method names, variable names, and even entire code snippets [83]. Previous research on code completion started with statistical language models [84,85]. Later, LSTM-based deep learning approaches were applied to the task, aiming to learn the semantic information of source code without considering its syntactic structure [86]. To address the limitations of LSTM-based language models, transformer architecture was introduced for code completion. Normally, the language models for code completion are trained using a causal language model that predicts the unknown token after a sequence of known tokens. Recent work on code completion using LLMs [35,87] has shown impressive performance on benchmarks, such as CodeXGLUE [34], compared to existing statistical language models and deep learning approaches.

3.3. Code Translation

Code translation is the process of converting code from one programming language to another, with the goal of migrating legacy software. While theoretically possible, building a code translator is challenging due to differences in syntax and platform APIs between programming languages. Most current translation tools are rule-based, requiring hand-crafted rewrite rules applied to an abstract syntax tree (AST) derived from the input source code. However, creating such tools demands significant expertise in both the source and target languages. Recent studies have explored using statistical machine translation [88,89] as well as deep learning approaches [90,91] for programming language translation. Quality evaluation for generated functions often uses the BLEU score, while the exact match is used to compare generated output with reference ground truth.

3.4. Code Refinement

Code refinement, which can be referred to as automated program repair (APR), is the process of automatically fixing bugs or vulnerabilities by converting a buggy function into a correct one. Deep learning models have a strong learning capability that enables them to learn various patterns for transforming buggy programs into patched ones from large code corpora. Many studies [92,93] have demonstrated the superior performance of deep learning-based techniques over traditional template-based [94,95], heuristic-based [96–98], and constraint-based [99,100] APR techniques. LLM is used to generate plausible patches or modifications to a given incorrect code. The model can be trained on a large corpus of correct code to learn the patterns and structures of correct code. When LLMs are given a faulty code, the model can then generate suggestions for how to correct it as one of the downstream tasks. The LLMs for code refinement can be evaluated by CodeXGLUE [34] or HumanEval [35] as the abstracted codes or the classical APR benchmarks such as Defects4J [101] and QuixBugs [102] as real-world codes, but the understanding and generation of concrete variable and function names is still mandatory and challenging [103].

3.5. Code Summarization

Code summarization is a technique used to generate English descriptions of code snippets at the function level, which can then be used to generate documentation. Typically, this involves taking the source code as input and producing a natural language summary as output. In AI-assisted programming tools, code summarization can be used to analyze code and identify optimization opportunities, such as using a binary Euclid algorithm instead of a traditional modular arithmetic-based algorithm, which can significantly improve software performance. In recent years, there has been promising research into the automatic generation of natural language descriptions of programs, with studies such as [104–106] making notable progress in this area. The rise of deep learning, coupled with the abundance of data from open-source repositories, has made automatic code summarization an area of interest for researchers. Many of the neural approaches [107,108] use a sequence-to-sequence approach to generate source code summaries, with some models converting the source code into various types of representations, such as token-based [109,110], tree-based [111,112], and graph-based [113,114], before passing it through language models.

3.6. Defect Detection

As software systems increase in complexity, it becomes more challenging to identify errors. Defect detection aims to enhance software reliability by predicting whether a piece of code is susceptible to bugs or not, by detecting previously unknown errors. Rule-based approaches have been defined in existing defect detection frameworks by inferring likely programming rules from various sources such as code, version histories, and comments [91,115,116]. Statistical language models based on N-gram language models have also been widely used in this area [117–119]. More recently, many deep learning-based solutions [95,120–125] have been proposed to bridge the gap by suggesting different feature sets from which the detection framework can learn, attempting to imitate how a practitioner

looks for vulnerabilities. However, LLMs, such as CodeBERT [126], have recently emerged as a promising technique in this field due to their ability to understand code structure. These models can be trained on a large corpus of error-free code and used to identify patterns and structures in source code that deviate from those learned from the error-free code as a binary classification task [127,128]. To evaluate the model predictions, accuracy, precision, recall, and F1 scores can be used.

3.7. Clone Detection

Clone detection involves identifying identical or similar code fragments, known as clones, within or across software systems. The goal of clone detection is to measure the similarity between two code snippets and determine if they have the same functionality. Clones can be classified into four types [129,130], with types 1–3 being syntactic clones that differ in minor ways, while type 4 clones, known as semantic clones, are difficult to detect since they have different syntax but the same semantics and, thus, require manual validation. With the increasing amount of source code, large-scale and automatic clone detection has become essential. Several tools have been developed to perform clone detection [131–136], using techniques such as comparison of the AST, tokens, or source code text. Notable clone detection datasets include BigCloneBench [25], which contains Java code snippets.

Table 4. Summary of language models for AI-assisted programming tasks.

Framework	Year	Task(s)	Baseline(s)	Supported Language(s)	Open Sourced
Refactory [137]	2019	Defect Detection	BLEU	Java	✗
CuBERT [138]	2020	Code Refinement, Defect Detection	BERT	Python	✓
CugLM [139]	2020	Code Completion	BERT	Java, TypeScript	✓
Intellicode [140]	2020	Code Generation, Code Completion	GPT-2	Python, C#, JavaScript, and TypeScrip	✗
Great [141]	2020	Defect Detection	Vanilla Transformers	Python	✓
TreeGEN [51]	2020	Code Generation	Vanilla Transformers	Python	✓
C-BERT [127]	2020	Defect Detection	BERT	C	✗
TransCoder [142]	2020	Code Translation	Vanilla Transformers	C++, Java, and Python	✗
GraphCodeBERT [143]	2020	Code Summarization, Code Refinement	BERT	Java	✗
Codex [35]	2021	Code Generation, Code Completion, Code Summarization, Benchmark	GPT-3	JavaScript, Go, Perl, and 6 more	✗
Copilot [144]	2021	Code Generation, Code Completion	Codex	Java, PHP, Python, and 5 more	✗
CodeT5 [145]	2021	Code Summarization, Code Generation, Code Translation, Code Refinement, Defect Detection, Clone Detection	T5	Python, Java	✓
Tfix [146]	2021	Code Refinement, Defect Detection	T5	JavaScript	✓
CodeRL [147]	2021	Code Summarization, Code Generation, Code Translation, Code Refinement, Defect Detection, Clone Detection	T5	Java	✓
TreeBERT [148]	2021	Code Summarization	Vanilla Transformers	Python, Java	✓
BUGLAB [149]	2021	Code Refinement, Defect Detection	GREAT	Python	✓
TBCC [150]	2021	Clone Detection	Vanilla Transformers	C, Java	✓
APPS [36]	2021	Benchmark	N/A	Python	✓
CodeXGLUE [34]	2021	Benchmark	N/A	Python	✓
CoTexT [151]	2021	Code Summarization, Code Generation, Code Refinement, Defect detection	T5	Python, Java, Javascript, PHP, Ruby, Go	✓
SynCoBERT [152]	2021	Code Translation, Defect Detection, Clone Detection	BERT	Ruby, Javascript, Go, Python, Java, PHP	✗
TravTrans [153]	2021	Code Completion	Vanilla Transformers	Python	✗
CCAG [154]	2021	Code Completion	Vanilla Transformers	JavaScript, Python	✗
DeepDebug [155]	2021	Defect Detection	Reformer	Java	✓

Table 4. *Cont.*

Framework	Year	Task(s)	Baseline(s)	Supported Language(s)	Open Sourced
Recoder [93]	2021	Defect Detection	TreeGen	Java	✓
PLBART [156]	2021	Code Summarization, Code Generation, Code Translation, Code Refinement, Clone Detection, Detect Detection	BART	Java, Python	✗
CODEGEN [157]	2022	Code Generation	GPT-NEO & GPT-J	Python	✓
GPT-2 for APR [158]	2022	Code Refinement	GPT-2	JavaScript	✓
CERT [39]	2022	Code Generation	CODEGEN	Python	✓
PyCoder [87]	2022	Code Generation	GPT-2	Python	✓
AlphaCode [38]	2022	Code Generation	GPT	Java	✗
InCoder [40]	2022	Code Generation, Code Completion, Code Summarization	GPT-3	Java, JavaScript, Python	✓
RewardRepair [159]	2022	Code Refinement, Defect Detection	T5	Java	✓
CodeParrot [37]	2022	Code Generation	GPT-2	Python	✓
AlphaRepair [160]	2022	Code Refinement, Defect Detection	CodeBERT	Java	✓
CodeReviewer [128]	2022	Code Summarization, Code Refinement, Defect Detection	CodeT5	Java	✓
TransRepair [161]	2022	Code Refinement, Defect Detection	BLEU	Java	✗
NatGen [162]	2022	Code Generation, Code Translation, Code Refinement	CodeT5	Java, Python, Go, JavaScript, Ruby, PHP	✓
DualSC [163]	2022	Code Generation, Code Summarization	T5	Shellcode	✓
VulRepair [164]	2022	Code Refinement, Defect Detection	T5	C, C++	✓
CoditT5 [165]	2022	Code Summarization, Defect Detection	CodeT5	Java, Python, Ruby, PHP, Go, JavaScript	✓
C4 [166]	2022	Clone Detection	CodeBERT	C++, C#, Java, Python	✓
SPT-Code [167]	2022	Code Summarization, Code Completion, Code Refinement, Code Translation	CodeBERT & GraphCode-BERT	Python, Java, JavaScript, PHP, Go	✓
ExploitGen [168]	2023	Code Generation	CodeBERT	Python, Assembly	✓
Santacoder [169]	2023	Code Summarization, Code Generation	GPT-2	Python, Java, and Javascript	✓
xCodeEval [42]	2023	Benchmark	N/A	Python, Java, C++, PHP, and 8 more	✓
StarCoder [170]	2023	Code Generation, Code Completion, Code Summarization	BERT & SantaCoder	HTML, Python, Java, and 83 more	✓

4. Challenges and Opportunities

4.1. Computational Expense

Training an LLM with millions of parameters can be computationally expensive. This is because training involves processing vast amounts of data in codes and optimizing the model's parameters to generate accurate predictions [171]. Overall, computational expense can be due to lack of training data and computing resources such as memory, GPU, or even electricity. At the same time, the quality of the training data used to train a language model is also crucial, as poor quality data or bias in the data can lead to incorrect predictions. LLMs require massive computational resources to train, fine-tune, and run, which can be a hindrance for organizations with limited hardware resources [172].

To reduce the computational expense of training LLMs, researchers and developers can employ various techniques, such as training on subsets of the data [173,174], optimizing the hyperparameters [175], and leveraging transfer learning to reuse the knowledge learned from previous tasks. These techniques can help to speed up the training process and reduce the amount of required computing resources. Instead of training the LLMs continuously, some works focus on using prompt-learning [176,177] and human feedback [178–182] to improve performance of the LLMs. In prompt-based learning, the prompt serves as a guide or prompt to the language model, providing it with relevant context and guidance to generate an output that is appropriate for a particular task. The prompt can be a simple sentence or a full paragraph, depending on the complexity of the task and the amount of information needed to guide the LLMs. One of the main advantages of prompt-based learning is its flexibility and ease of use. It allows users to quickly fine-tune pre-trained language models for specific tasks without requiring a large amount of task-specific data.

Additionally, prompt-based learning can be used in a semi-supervised or unsupervised manner, where the prompt provides a small amount of supervision to the language model, further reducing the necessary amount of task-specific data.

4.2. Quality Measurement

Leveraging LLMs in AI-assisted programming tasks has enormous potential to improve software development efficiency and reduce the time and effort required to write code manually. However, several challenges need to be addressed to ensure the performance and effectiveness of LLMs. One of the primary concerns is the quality of the generated code or documentation [35], which can be impacted by the accuracy and robustness of the LLMs. While automated code generation can save time, it can also lead to poor-quality code that is difficult to maintain and may contain bugs or security vulnerabilities [183]. Therefore, it is critical to ensure that the generated code meets the desired specifications and adheres to coding standards and best practices [184]. Another significant challenge is integrating the generated code into existing software systems seamlessly [185], ensuring that it can be maintained and updated easily over time.

To address these challenges and improve the reliability and quality of LLMs in AI-assisted programming tasks, researchers and developers are exploring various approaches and techniques. These include incorporating advanced machine learning and optimization algorithms [186,187] and developing new tools and frameworks for integrating generated code into existing software systems. Some researchers have attempted to use Variational Autoencoders [188] or Generative Adversarial Networks [189] to generate synthetic data that can be used for training LLMs, but they must ensure that the performance of these generative models is robust and reliable to ensure the quality of the synthetic data. Meanwhile, it is possible to adopt active learning [190] to improve the performance of LLMs while requiring fewer labeled training instances. This approach works by allowing the model to choose the data from which it learns [191], which enables it to compute the statistically optimal way to select training data while avoiding poor-quality data, such as buggy codes, that can negatively impact model performance. One of the significant benefits of incorporating active learning into the training process is that it can help reduce the time and effort required to label large amounts of data manually, making it a cost-effective solution for many applications [192]. By selecting the most informative data points for labeling, active learning can improve the accuracy and robustness of machine learning models, even when working with limited labeled data. The integration of active learning with LLMs remains an open question in this field of study. While active learning has shown promise in improving the performance of machine learning models, including LLMs, the application of this technique to LLMs has not yet been fully explored.

4.3. Software Security

Software security is a critical concern in the development of the use of LLMs [193]. While LLMs have shown significant promise in a wide range of code-related tasks, they also introduce unique security challenges that must be addressed to ensure safety and security. One of the primary security concerns when using LLMs is the potential for these models to introduce vulnerabilities into the code [194]. For example, poorly designed LLMs may generate code that is prone to buffer overflow or SQL injection attacks. Another critical concern is the possibility of LLMs being manipulated or exploited to generate malicious code that can be used for cyberattacks. For instance, an attacker may use a poisoned dataset to manipulate an LLM, resulting in the generation of malicious code that can be used to exploit vulnerabilities in the software system. Also, users without programming knowledge can generate programs with a Trojan horse phishing attack.

When using LLMs for AI-assisted programming tasks, it is essential to address software security to ensure that the generated codes or documents are secure and free from vulnerabilities, as well as to ensure the integrity of the training data used to train the LLMs. Code validation and testing involve thorough validation and testing of the generated code

before integrating it with real-world systems to identify and fix any security issues. Data sanitization and validation ensure that the training data are free from malicious code or sources of bias.

4.4. Software Piracy

Software piracy refers to the unauthorized copying, distribution, or use of copyrighted software without the permission of the software's owner [195–197]. This can take many forms, including making copies of software for personal or commercial use, distributing software through unauthorized channels, or using software beyond the terms of the licensing agreement. As the field of natural language generation and statistical machine learning for Big Code and AI-assisted programming continues to grow, concerns over software piracy have arisen. The use of open source code repositories for training AI models has led to lawsuits, with companies such as Microsoft and OpenAI accused of software piracy. The issue at hand is whether the use of open source code for training LLMs violates copyright laws. While the legal implications of this issue are still being debated, it is important to consider the ethical implications as well. The use of copyrighted code without permission raises questions about fairness and equity in the development of AI-assisted programming tools [198,199]. Also, the use of user data to train these models raises concerns over privacy and data protection. As the field continues to evolve, it will be important for researchers and developers to consider these issues and work towards finding solutions that balance the benefits of AI-assisted programming with the need for ethical and legal compliance. This may include clarifying rules around secondary uses of copyrighted code, as well as developing more transparent and opt-in data policies for training AI models.

To address software piracy, one approach is to ensure that the training data used for the development of these models are legally obtained and do not violate any copyrights or intellectual property rights according to the U.S. Copyright Office [200]. Organizations can also establish clear policies and guidelines for the ethical and legal use of these technologies. For instance, developers can be required to obtain permission or licenses before using proprietary code or software in their work. Machine learning algorithms can also be trained to identify and prevent the unauthorized distribution of copyrighted material and pirated code or software.

4.5. Integration with Existing Tools

The opportunity to integrate tools and LLMs enhances and streamlines the software development process. By incorporating LLMs into integrated tools as cloud virtual service providers [201,202], developers can leverage the power of NLP to automate repetitive tasks, improve code quality and readability, and increase efficiency in software development. This integration can enable developers to experiment prompt engineering with public LLMs under data compliance, data security, data governance and best practices directly from their own development environment. Copilot for Xcode [203] serves as a real-world example of an application integrated with LLMs, allowing Apple developers to utilize GitHub Copilot [144] for code suggestions and ChatGPT [176] for code explanation and mutation using natural language. The connection between Xcode and Copilot is achieved by establishing communication between the Xcode source editor extension and the Copilot server, presenting suggestions in a user interface not handled by Xcode. To obtain additional information beyond the source code and file type provided by Xcode, the app utilizes the Accessibility API, which represents objects in a user interface and exposes information about each object within the application. Furthermore, for in-place code editing, the app employs the use of Apple Scripts, a scripting language in macOS for task automation, to programmatically execute extension commands and emulate menu bar interactions. The details to integrate the Copilot with Xcode are illustrated in Figure 5.

With these workarounds, Copilot for Xcode successfully enables Xcode to support GitHub Copilot, as shown in Figure 6. In addition, it facilitates the integration of an external chat panel that can access and read the user's code. This chat panel serves as a connection

point to leverage LLMs for functionalities such as code explanation and mutation using natural language. The chat panel can also be extended with plugins to offer additional features, including support for natural language terminal commands. The incorporation of Copilot into Xcode signifies a notable advancement in AI-powered programming for iOS/macOS, expanding the capabilities of language models to widely-used mobile software development tools.

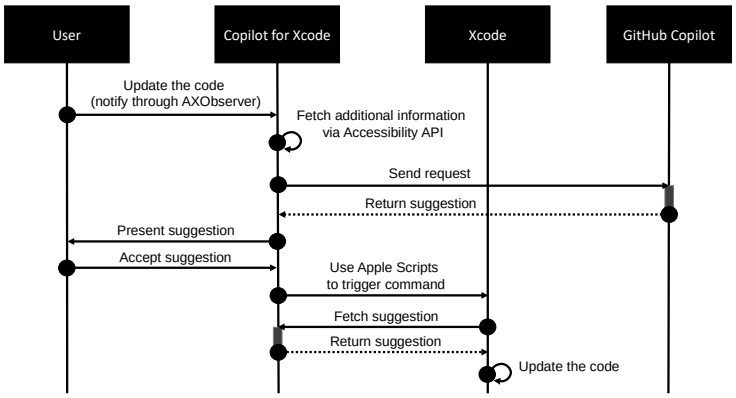
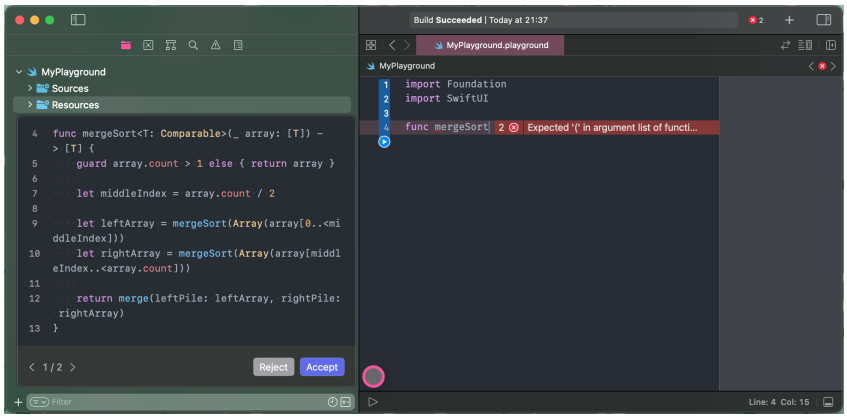
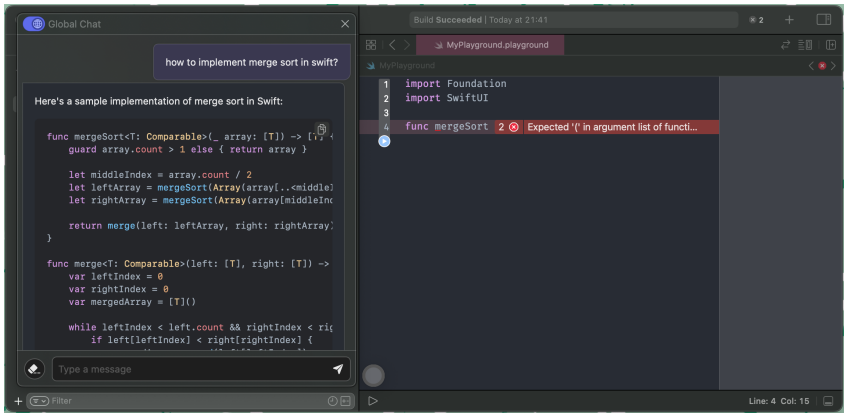


Figure 5. A sequence diagram of Copilot for Xcode to produce real-time suggestions with GitHub Copilot. When a user attempts to update their code, the Copilot for Xcode first receives a notification and sends a request to the GitHub Copilot API. Once the suggestions from GitHub Copilot are returned, the user can choose to adopt the suggestions and apply the changes directly to Xcode.



(a) Copilot for Xcode displaying suggestions from GitHub Copilot.



(b) Copilot for Xcode displaying the chat panel.

Figure 6. Interface of Copilot for Xcode integrated with Apple Xcode. (**a**,**b**) are the actual user interface tool, where a developer can interact with the GitHub Copilot inside the Xcode.

5. Conclusions

This review paper explores the applications of LLMs in software naturalness to gain a better understanding of software development processes and develop applications that cater to the human aspects of software development. Firstly, it provides a background on Big Code and software naturalness, covering topics such as available datasets, tokenization processes, existing language models, and entropy-based measurements. Secondly, it summarizes recent applications of LLMs trained with Big Code in various tasks, including code generation, code completion, code translation, code refinement, code summarization, defect detection, and clone detection. Lastly, it discusses the potential challenges and opportunities associated with LLMs in the context of AI-assisted programming tasks.

Analyzing Big Code repositories and identifying patterns of naturalness can lead to more effective methods for AI-assisted programming. This can ultimately improve the quality and productivity of AI-assisted programming, making it easier for programmers to create high-quality software with fewer errors in less time. In addition to the challenges faced by LLMs for codes mentioned in this review paper, there are significant opportunities for future work in the field. These opportunities include exploring the development of LLMs that prioritize transparency and interpretability, enabling clearer explanations for code suggestions and bug fixing. Emphasizing the design of AI-assisted programming applications that prioritize fairness, transparency, and privacy is crucial, as current research tends to focus primarily on performance and efficiency. By pursuing these avenues, AI-assisted programming applications can be advanced to be more user-centric, ethically responsible, and adaptable, ultimately leading to more efficient and effective programming workflows.

Author Contributions: Conceptualization, M.-F.W. and C.-W.T.; methodology, M.-F.W., S.G., C.-N.H., S.-W.H. and C.-W.T.; software: S.G. and C.-W.T.; validation, M.-F.W., S.-W.H. and C.-W.T.; supervision, M.-F.W., S.-W.H. and C.-W.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported in part by the Ministry of Education, Singapore, under its Academic Research Fund (No. 022307 and AcRF RG91/22) and Google Faculty Award.

Institutional Review Board Statement: Not applicable.

Data Availability Statement: Data sharing not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Vechev, M.; Yahav, E. Programming with "Big Code". *Found. Trends® Program. Lang.* **2016**, *3*, 231–284. [CrossRef]
2. Hindle, A.; Barr, E.T.; Su, Z.; Gabel, M.; Devanbu, P. On The Naturalness of Software. In Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 837–847.
3. Goodman, J.T. A bit of progress in language modeling. In *Computer Speech & Language*; Elsevier: Amsterdam, The Netherlands, 2001; pp. 403–434.
4. Dijkstra, E.W. *A Preliminary Investigation into Computer Assisted Programming*; The University of Texas: Austin, TX, USA, 2007.
5. Rajamani, S. AI Assisted Programming. In Proceedings of the 15th Annual ACM India Compute Conference, Jaipur, India, 9–11 November 2022; p. 5.
6. Dijkstra, E.W. The Humble Programmer. *Commun. ACM* **1972**, *15*, 859–866. [CrossRef]
7. Ji, Y.; Bosselut, A.; Wolf, T.; Celikyilmaz, A. The Amazing World of Neural Language Generation. In Proceedings of the Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts, Virtual, 19–20 November 2020; pp. 37–42.
8. Surameery, N.M.S.; Shakor, M.Y. Use ChatGPT to Solve Programming Bugs. *Int. J. Inf. Technol. Comput. Eng. (IJITC)* **2023**, *3*, 17–22.
9. Talamadupula, K. Applied AI Matters: AI4Code: Applying Artificial Intelligence to Source Code. *AI Matters* **2021**, *7*, 18–20. [CrossRef]
10. Ross, S.I.; Martinez, F.; Houde, S.; Muller, M.; Weisz, J.D. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In Proceedings of the 28th International Conference on Intelligent User Interfaces, Sydney, Australia, 27–31 March 2023; pp. 491–514.

11. Mehrabi, N.; Morstatter, F.; Saxena, N.; Lerman, K.; Galstyan, A. A Survey on Bias and Fairness in Machine Learning. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–35. [CrossRef]
12. Carvalho, D.V.; Pereira, E.M.; Cardoso, J.S. Machine Learning Interpretability: A Survey on Methods and Metrics. *Electronics* **2019**, *8*, 832. [CrossRef]
13. Tjoa, E.; Guan, C. A Survey on Explainable Artificial Intelligence (XAI): Toward Medical XAI. *IEEE Trans. Neural Netw. Learn. Syst.* **2020**, *32*, 4793–4813. [CrossRef] [PubMed]
14. Beigi, G.; Liu, H. A Survey on Privacy in Social Media: Identification, Mitigation, and Applications. *ACM Trans. Data Sci.* **2020**, *1*, 1–38. [CrossRef]
15. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–37. [CrossRef]
16. Lin, G.; Wen, S.; Han, Q.L.; Zhang, J.; Xiang, Y. Software Vulnerability Detection using Deep Neural Networks: A Survey. *Proc. IEEE* **2020**, *108*, 1825–1848. [CrossRef]
17. Sharma, T.; Kechagia, M.; Georgiou, S.; Tiwari, R.; Vats, I.; Moazen, H.; Sarro, F. A Survey on Machine Learning Techniques for Source Code Analysis. *arXiv* **2022**, arXiv:2110.09610.
18. Sonnekalb, T.; Heinze, T.S.; Mäder, P. Deep Security Analysis of Program Code: A Systematic Literature Review. *Empir. Softw. Eng.* **2022**, *27*, 2. [CrossRef]
19. Xu, Y.; Zhu, Y. A Survey on Pretrained Language Models for Neural Code Intelligence. *arXiv* **2022**, arXiv:2212.10079.
20. Niu, C.; Li, C.; Luo, B.; Ng, V. Deep Learning Meets Software Engineering: A Survey on Pre-trained Models of Source Code. In Proceedings of the 31st International Joint Conference on Artificia Intelligence (IJCAI-22), Vienna, Austria, 23–29 July 2022.
21. Ciancarini, P.; Farina, M.; Okonicha, O.; Smirnova, M.; Succi, G. Software as Storytelling: A Systematic Literature Review. *Comput. Sci. Rev.* **2023**, *47*, 100517. [CrossRef]
22. Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; Neubig, G. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv. (CSUR)* **2023**, *55*, 1–35. [CrossRef]
23. Allamanis, M.; Sutton, C. Mining Source Code Repositories at Massive Scale using Language Modeling. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 207–216.
24. Description2Code Dataset. 2016. Available online : https://github.com/ethancaballero/description2code (accessed on 18 May 2023).
25. Svajlenko, J.; Roy, C.K. Description2Code Dataset. 2021. Available online: https://github.com/clonebench/BigCloneBench (accessed on 18 May 2023).
26. Chen, Z.; Monperrus, M. The CodRep Machine Learning on Source Code Competition. *arXiv* **2018**, arXiv:1807.03200.
27. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Mapping Language to Code in Programmatic Context. *arXiv* **2018**, arXiv:1808.09588.
28. Zhong, V.; Xiong, C.; Socher, R. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv* **2017**, arXiv:1709.00103.
29. Tufano, M.; Watson, C.; Bavota, G.; Penta, M.D.; White, M.; Poshyvanyk, D. An Empirical Study on Learning Bug-fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2019**, *28*, 1–29. [CrossRef]
30. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Proceedings of the Advances in Neural Information Processing Systems 32 (NeurIPS 2019), Vancouver, BC, Canada, 8–14 December 2019.
31. Husain, H.; Wu, H.H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv* **2019**, arXiv:1909.09436.
32. Gao, L.; Biderman, S.; Black, S.; Golding, L.; Hoppe, T.; Foster, C.; Phang, J.; He, H.; Thite, A.; Nabeshima, N.; et al. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv* **2020**, arXiv:2101.00027.
33. Puri, R.; Kung, D.S.; Janssen, G.; Zhang, W.; Domeniconi, G.; Zolotov, V.; Dolby, J.; Chen, J.; Choudhury, M.; Decker, L.; et al. CodeNet: A Large-scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *arXiv* **2021**, arXiv:2105.12655.
34. Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.B.; Drain, D.; Jiang, D.; Tang, D.; et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv* **2021**, arXiv:2102.04664.
35. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374.
36. Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; et al. Measuring Coding Challenge Competence With APPS. *arXiv* **2021**, arXiv:2105.09938.
37. Tunstall, L.; Von Werra, L.; Wolf, T. *Natural Language Processing with Transformers*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2022.
38. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-level Code Generation with Alphacode. *Science* **2022**, *378*, 1092–1097. [CrossRef]
39. Zan, D.; Chen, B.; Yang, D.; Lin, Z.; Kim, M.; Guan, B.; Wang, Y.; Chen, W.; Lou, J.G. CERT: Continual Pre-training on Sketches for Library-oriented Code Generation. In Proceedings of the 31st International Joint Conference on Artificia Intelligence (IJCAI-22), Vienna, Austria, 23–29 July 2022.
40. Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.t.; Zettlemoyer, L.; Lewis, M. Incoder: A Generative Model for Code Infilling and Synthesis. *arXiv* **2022**, arXiv:2204.05999.

41. Xu, F.F.; Alon, U.; Neubig, G.; Hellendoorn, V.J. A Systematic Evaluation of Large Language Models of Code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022; pp. 1–10.
42. Khan, M.A.M.; Bari, M.S.; Do, X.L.; Wang, W.; Parvez, M.R.; Joty, S. xCodeEval: A Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval. *arXiv* **2023**, arXiv: 2303.03004.
43. Sennrich, R.; Haddow, B.; Birch, A. Neural Machine Translation of Rare Words with Subword Units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 1715–1725.
44. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. Roberta: A Robustly Optimized BERT Pretraining Approach. *arXiv* **2019**, arXiv:1907.11692.
45. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774.
46. Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. LLaMA: Open and Efficient Foundation Language Models. *arXiv* **2023**, arXiv:2302.13971.
47. Cho, K.; van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1724–1734.
48. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is All You Need. In Proceedings of the Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017.
49. Lewis, M.; Liu, Y.; Goyal, N.; Ghazvininejad, M.; Mohamed, A.; Levy, O.; Stoyanov, V.; Zettlemoyer, L. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Virtual, 5–10 July 2020; pp. 7871–7880.
50. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring The Limits of Transfer Learning with a Unified Text-to-text Transformer. *J. Mach. Learn. Res.* **2020**, *21*, 5485–5551.
51. Sun, Z.; Zhu, Q.; Xiong, Y.; Sun, Y.; Mou, L.; Zhang, L. Treegen: A Tree-based Transformer Architecture for Code Generation. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; pp. 8984–8991.
52. Morin, F.; Bengio, Y. Hierarchical Probabilistic Neural Network Language Model. In Proceedings of the International Workshop on Artificial Intelligence and Statistics, Bridgetown, Barbados, 6–8 January 2005; pp. 246–252.
53. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. *Code2Vec: Learning Distributed Representations of Code*; ACM: New York, NY, USA, 2019; Volume 3, pp. 1–29.
54. Peters, M.; Neumann, M.; Iyyer, M.; Gardner, M.; Clark, C.; Lee, K.; Zettlemoyer, L. Deep Contextualized Word Representations. In Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, New Orleans, LA, USA, 1–6 June 2018; pp. 2227–2237.
55. Mihalcea, R.; Tarau, P. TextRank: Bringing order into text. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, Barcelona, Spain, 25–26 July 2004; pp. 404–411.
56. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs. In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
57. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA, 2–7 June 2019.
58. Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* **2019**, *1*, 9.
59. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
60. Wang, B.; Komatsuzaki, A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. 2021. Available online: https://github.com/kingoflolz/mesh-transformer-jax (accessed on 18 May 2023).
61. Kitaev, N.; Kaiser, L.; Levskaya, A. Reformer: The Efficient Transformer. In Proceedings of the International Conference on Learning Representations, Virtual, 26–30 April 2020.
62. Black, S.; Gao, L.; Wang, P.; Leahy, C.; Biderman, S. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. 2021. Available online: https://github.com/EleutherAI/gpt-neo (accessed on 18 May 2023).
63. Jurafsky, D.; Martin, J.H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed.; Prentice Hall PTR: Hoboken, NJ, USA, 2000.
64. Bengio, Y.; Ducharme, R.; Vincent, P. A Neural Probabilistic Language Model. In Proceedings of the Advances in Neural Information Processing Systems 13 (NIPS 2000), Denver, CO, USA, 27 November–2 December 2000.
65. Katz, S. Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer. *IEEE Trans. Acoust. Speech Signal Process.* **1987**, *35*, 400–401. [CrossRef]
66. Brown, P.F.; Della Pietra, V.J.; Desouza, P.V.; Lai, J.C.; Mercer, R.L. Class-based N-gram Models of Natural Language. *Comput. Linguist.* **1992**, *18*, 467–480.
67. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space. *arXiv* **2013**, arXiv:1301.3781.

68. Shannon, C.E. Prediction and Entropy of Printed English. *Bell Syst. Tech. J.* **1951**, *30*, 50–64. [CrossRef]
69. Mozannar, H.; Bansal, G.; Fourney, A.; Horvitz, E. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *arXiv* **2022**, arXiv:2210.14306.
70. Ho, S.W.; Yeung, R.W. The Interplay between Entropy and Variational Distance. *IEEE Trans. Inf. Theory* **2010**, *56*, 5906–5929. [CrossRef]
71. Kennel, M.B.; Shlens, J.; Abarbanel, H.D.; Chichilnisky, E. Estimating Entropy Rates with Bayesian Confidence Intervals. *Neural Comput.* **2005**, *17*, 1531–1576. [CrossRef]
72. Feutrill, A.; Roughan, M. A Review of Shannon and Differential Entropy Rate Estimation. *Entropy* **2021**, *23*, 1046. [CrossRef] [PubMed]
73. Paninski, L. Estimation of Entropy and Mutual Information. *Neural Comput.* **2003**, *15*, 1191–1253. [CrossRef]
74. Waldinger, R.J.; Lee, R.C. PROW: A Step toward Automatic Program Writing. In Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, 7–9 May 1969; pp. 241–252.
75. Manna, Z.; Waldinger, R.J. Toward Automatic Program Synthesis. *Commun. ACM* **1971**, *14*, 151–165. [CrossRef]
76. Manna, Z.; Waldinger, R. Knowledge and Reasoning in Program Synthesis. *Artif. Intell.* **1975**, *6*, 175–208. [CrossRef]
77. Green, C. Application of Theorem Proving to Problem Solving. In *Readings in Artificial Intelligence*; Elsevier: Amsterdam, The Netherlands, 1981; pp. 202–222.
78. Dong, L.; Lapata, M. Language to Logical Form with Neural Attention. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 33–43.
79. Parisotto, E.; Mohamed, A.r.; Singh, R.; Li, L.; Zhou, D.; Kohli, P. Neuro-Symbolic Program Synthesis. *arXiv* **2016**, arXiv:1611.01855.
80. Lin, C.Y.; Och, F.J. Orange: A Method for Evaluating Automatic Evaluation Metrics for Machine Translation. In Proceedings of the 20th International Conference on Computational Linguistics, Geneva, Switzerland, 23–27 August 2004; pp. 501–507.
81. Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. Program Synthesis with Large Language Models. *arXiv* **2021**, arXiv:2108.07732.
82. Dong, Y.; Gu, T.; Tian, Y.; Sun, C. SnR: Constraint-based Type Inference for Incomplete Java Code Snippets. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 1982–1993.
83. Amazon, C. AI Code Generator—Amazon CodeWhisperer. Available online: https://aws.amazon.com/codewhisperer (accessed on 18 May 2023).
84. Robbes, R.; Lanza, M. How Program History Can Improve Code Completion. In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, L'aquila, Italy, 15–16 September 2008; pp. 317–326.
85. Bruch, M.; Monperrus, M.; Mezini, M. Learning from Examples to Improve Code Completion Systems. In Proceedings of the 7th Joint Meeting of The European Software Engineering Conference and The ACM SIGSOFT Symposium on The Foundations of Software Engineering, Amsterdam, The Netherlands, 24–28 August 2009; pp. 213–222.
86. Svyatkovskiy, A.; Zhao, Y.; Fu, S.; Sundaresan, N. Pythia: Ai-assisted code completion system. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Anchorage, AK, USA, 4–8 August 2019; pp. 2727–2735.
87. Takerngsaksiri, W.; Tantithamthavorn, C.; Li, Y.F. Syntax-Aware On-the-Fly Code Completion. *arXiv* **2022**, arXiv:2211.04673.
88. Koehn, P.; Federico, M.; Shen, W.; Bertoldi, N.; Bojar, O.; Callison-Burch, C.; Cowan, B.; Dyer, C.; Hoang, H.; Zens, R.; et al. Open Source Toolkit for Statistical Machine Translation: Factored Translation Models and Confusion Network Decoding. In Proceedings of the CLSP Summer Workshop Final Report WS-2006, Baltimore, MD, USA, 1 June–1 August 2007 .
89. Artetxe, M.; Labaka, G.; Agirre, E. Unsupervised Statistical Machine Translation. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November 2018.
90. Allamanis, M.; Barr, E.T.; Bird, C.; Sutton, C. Learning Natural Coding Conventions. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 281–293.
91. Acharya, M.; Xie, T.; Pei, J.; Xu, J. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In Proceedings of the 6th Joint Meeting of The European Software Engineering Conference and The ACM SIGSOFT Symposium on The Foundations of Software Engineering, Dubrovnikm, Croatia, 3–7 September 2007; pp. 25–34.
92. Jiang, N.; Lutellier, T.; Tan, L. Cure: Code-aware Neural Machine Translation for Automatic Program Repair. In Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering, Madrid, Spain, 22–30 May 2021; pp. 1161–1173.
93. Zhu, Q.; Sun, Z.; Xiao, Y.a.; Zhang, W.; Yuan, K.; Xiong, Y.; Zhang, L. A Syntax-guided Edit Decoder for Neural Program Repair. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 341–353.
94. Jiang, J.; Xiong, Y.; Zhang, H.; Gao, Q.; Chen, X. Shaping Program Repair Space with Existing Patches and Similar Code. In Proceedings of the 27th ACM SIGSOFT International Symposium On Software Testing And Analysis, Amsterdam, The Netherlands, 16–21 July 2018; pp. 298–309.
95. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T.F. TBar: Revisiting Template-based Automated Program Repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing China, 15–19 July 2019; pp. 31–42.
96. Yuan, Y.; Banzhaf, W. Arja: Automated Repair of Java Programs via Multi-objective Genetic Programming. *IEEE Trans. Softw. Eng.* **2018**, *46*, 1040–1067. [CrossRef]

97. Wen, M.; Chen, J.; Wu, R.; Hao, D.; Cheung, S.C. Context-aware patch generation for better automated program repair. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 1–11.
98. Saha, R.K.; Lyu, Y.; Yoshida, H.; Prasad, M.R. Elixir: Effective Object-oriented Program Repair. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Urbana-Champaign, IL, USA, 30 October–3 November 2017; pp. 648–659.
99. Xiong, Y.; Wang, J.; Yan, R.; Zhang, J.; Han, S.; Huang, G.; Zhang, L. Precise Condition Synthesis for Program Repair. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering, Buenos Aires, Argentina, 20–28 May 2017; pp. 416–426.
100. Xuan, J.; Martinez, M.; Demarco, F.; Clement, M.; Marcote, S.L.; Durieux, T.; Le Berre, D.; Monperrus, M. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* **2016**, *43*, 34–55. [CrossRef]
101. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In Proceedings of the International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 437–440.
102. Lin, D.; Koppel, J.; Chen, A.; Solar-Lezama, A. QuixBugs: A Multi-lingual Program Repair Benchmark Set Based on The Quixey Challenge. In Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, Vancouver, BC, Canada, 22–27 October 2017; pp. 55–56.
103. Jiang, N.; Liu, K.; Lutellier, T.; Tan, L. Impact of Code Language Models on Automated Program Repair. In Proceedings of the IEEE/ACM 45th International Conference on Software Engineering, Melbourne, Australia, 14–20 May 2023.
104. Sridhara, G.; Hill, E.; Muppaneni, D.; Pollock, L.; Vijay-Shanker, K. Towards Automatically Generating Summary Comments for Java Methods. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 20–24 September 2010; pp. 43–52.
105. Moreno, L.; Aponte, J.; Sridhara, G.; Marcus, A.; Pollock, L.; Vijay-Shanker, K. Automatic Generation of Natural Language Summaries for Java Classes. In Proceedings of the 21st International Conference on Program Comprehension, San Francisco, CA, USA, 20–21 May 2013.
106. Sridhara, G.; Pollock, L.; Vijay-Shanker, K. Generating Parameter Comments and Integrating with Method Summaries. In Proceedings of the IEEE 19th International Conference on Program Comprehension, Kingston, ON, Canada, 22–24 June 2011; pp. 71–80.
107. Ahmad, W.; Chakraborty, S.; Ray, B.; Chang, K.W. A Transformer-based Approach for Source Code Summarization. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Virtual, 5–10 July 2020; pp. 4998–5007.
108. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing Source Code Using a Neural Attention Model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 2073–2083.
109. Allamanis, M.; Peng, H.; Sutton, C. A Convolutional Attention Network for Extreme Summarization of Source Code. In Proceedings of the International Conference on Machine Learning, New York, NY, USA, 20–22 June 2016; pp. 2091–2100.
110. Chen, Q.; Zhou, M. A Neural Framework for Retrieval and Summarization of Source Code. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 826–831.
111. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional Neural Networks Over Tree Structures for Programming Language Processing. In Proceedings of the AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; Volume 30.
112. Liang, Y.; Zhu, K. Automatic Generation of Text Descriptive Comments for Code Blocks. In Proceedings of the AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018; Volume 32.
113. Tufano, M.; Watson, C.; Bavota, G.; Di Penta, M.; White, M.; Poshyvanyk, D. Deep Learning Similarities From Different Representations of Source Code. In Proceedings of the 15th International Conference on Mining Software Repositories, Gothenburg, Sweden, 27 May–3 June 2018.
114. Ou, M.; Cui, P.; Pei, J.; Zhang, Z.; Zhu, W. Asymmetric Transitivity Preserving Graph Embedding. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 1105–1114.
115. Livshits, B.; Zimmermann, T. Dynamine: Finding Common Error Patterns by Mining Software Revision Histories. *ACM SIGSOFT Softw. Eng. Notes* **2005**, *30*, 296–305. [CrossRef]
116. Wasylkowski, A.; Zeller, A.; Lindig, C. Detecting Object Usage Anomalies. In Proceedings of the 6th Joint Meeting of The European Software Engineering Conference and The ACM SIGSOFT Symposium on The Foundations of Software Engineering, Dubrovnik, Croatia, 3–7 September 2007; pp. 35–44.
117. Charniak, E. *Statistical Language Learning*; MIT Press: Cambridge, MA, USA, 1996.
118. Nessa, S.; Abedin, M.; Wong, W.E.; Khan, L.; Qi, Y. Software Fault Localization Using N-gram Analysis. In Proceedings of the Wireless Algorithms, Systems, and Applications: 3rd International Conference, Dallas, TX, USA, 26–28 October 2008; pp. 548–559.
119. Wang, S.; Chollak, D.; Movshovitz-Attias, D.; Tan, L. Bugram: Bug Detection with N-gram Language Models. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 708–719.
120. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y.; De Vel, O.; Montague, P. Cross-project Transfer Representation Learning for Vulnerable Function Discovery. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3289–3297. [CrossRef]
121. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. Vuldeepecker: A Deep Learning-based System for Vulnerability Detection. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium, San Diego, CA, USA, 18–21 February 2018.

122. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; McConley, M. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In Proceedings of the 17th IEEE International Conference on Machine Learning and Applications, Orlando, FL, USA, 17–20 December 2018; pp. 757–762.
123. Le, T.; Nguyen, T.; Le, T.; Phung, D.; Montague, P.; De Vel, O.; Qu, L. Maximal Divergence Sequential Autoencoder for Binary Software Vulnerability Detection. In Proceedings of the International Conference on Learning Representations, New Orleans, LA, USA, 6–9 May 2019.
124. Chen, Z.; Kommrusch, S.; Tufano, M.; Pouchet, L.N.; Poshyvanyk, D.; Monperrus, M. Sequencer: Sequence-to-sequence Learning for End-to-end Program Repair. *IEEE Trans. Softw. Eng.* **2019**, *47*, 1943–1959. [CrossRef]
125. Gupta, R.; Pal, S.; Kanade, A.; Shevade, S. Deepfix: Fixing Common C Language Errors by Deep Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4–9 February 2017; Volume 31.
126. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Findings of the Association for Computational Linguistics (EMNLP 2020), Virtual, 16–20 November 2020; pp. 1536–1547.
127. Buratti, L.; Pujar, S.; Bornea, M.; McCarley, S.; Zheng, Y.; Rossiello, G.; Morari, A.; Laredo, J.; Thost, V.; Zhuang, Y.; et al. Exploring Software Naturalness through Neural Language Models. *arXiv* **2020**, arXiv:2006.12641.
128. Li, Z.; Lu, S.; Guo, D.; Duan, N.; Jannu, S.; Jenks, G.; Majumder, D.; Green, J.; Svyatkovskiy, A.; Fu, S.; et al. Automating Code Review Activities by Large-scale Pre-training. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 1035–1047.
129. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [CrossRef]
130. Roy, C.K.; Cordy, J.R. A Survey on Software Clone Detection Research. *Queen's Sch. Comput. TR* **2007**, *541*, 64–68.
131. Kontogiannis, K.A.; DeMori, R.; Merlo, E.; Galler, M.; Bernstein, M. Pattern Matching for Clone and Concept Detection. *Autom. Softw. Eng.* **1996**, *3*, 77–108. [CrossRef]
132. Ducasse, S.; Rieger, M.; Demeyer, S. A Language Independent Approach for Detecting Duplicated Code. In Proceedings of the IEEE International Conference on Software Maintenance, Oxford, UK, 30 August–3 September 1999; pp. 109–118.
133. Baxter, I.D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. Clone Detection using Abstract Syntax Trees. In Proceedings of the International Conference on Software Maintenance, Bethesda, MD, USA, 16–19 November 1998; pp. 368–377.
134. Chen, K.; Liu, P.; Zhang, Y. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 175–186.
135. Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.K.; Lopes, C.V. Sourcerercc: Scaling code clone detection to big-code. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 1157–1168.
136. Yu, H.; Lam, W.; Chen, L.; Li, G.; Xie, T.; Wang, Q. Neural Detection of Semantic Code Clones via Tree-based Convolution. In Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension, Montreal, QC, Canada, 25–26 May 2019; pp. 70–80.
137. Hu, Y.; Ahmed, U.Z.; Mechtaev, S.; Leong, B.; Roychoudhury, A. Re-factoring based Program Repair applied to Programming Assignments. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, CA, USA, 11–15 November 2019; pp. 388–398.
138. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and Evaluating Contextual Embedding of Source Code. In Proceedings of the International Conference on Machine Learning, Virtual, 13–18 July 2020; pp. 5110–5121.
139. Liu, F.; Li, G.; Zhao, Y.; Jin, Z. Multi-task Learning Based Pre-trained Language Model for Code Completion. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual, 21–25 September 2020; pp. 473–485.
140. Svyatkovskiy, A.; Deng, S.K.; Fu, S.; Sundaresan, N. Intellicode Compose: Code Generation Using Transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual, 8–13 November 2020; pp. 1433–1443.
141. Hellendoorn, V.J.; Sutton, C.; Singh, R.; Maniatis, P.; Bieber, D. Global Relational Models of Source Code. In Proceedings of the International Conference on Learning Representations, Virtual, 26–30 April 2020.
142. Roziere, B.; Lachaux, M.A.; Chanussot, L.; Lample, G. Unsupervised Translation of Programming Languages. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 20601–20611.
143. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the International Conference on Learning Representations, Vienna, Austria, 3–7 May 2021.
144. Friedman, N. Introducing GitHub Copilot: Your AI Pair Programmer. 2021. Available online: https://github.com/features/copilot (accessed on 18 May 2023).
145. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, Punta Cana, Dominican Republic, 7–11 November 2021; pp. 8696–8708.

146. Berabi, B.; He, J.; Raychev, V.; Vechev, M. Tfix: Learning to Fix Coding Errors with a Text-to-text Transformer. In Proceedings of the International Conference on Machine Learning. PMLR, Virtual, 18–24 July 2021; pp. 780–791.
147. Le, H.; Wang, Y.; Gotmare, A.D.; Savarese, S.; Hoi, S. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In Proceedings of the Advances in Neural Information Processing Systems 35 (NeurIPS 2022), New Orleans, LA, USA, 28 November 2022.
148. Jiang, X.; Zheng, Z.; Lyu, C.; Li, L.; Lyu, L. TreeBERT: A Tree-based Pre-trained Model for Programming Language. In Proceedings of the Uncertainty in Artificial Intelligence, Virtual, 27–30 July 2021; pp. 54–63.
149. Allamanis, M.; Jackson-Flux, H.; Brockschmidt, M. Self-supervised Bug Detection and Repair. In Proceedings of the Advances in Neural Information Processing Systems 34 (NeurIPS 2021), Virtual, 6–14 December 2021.
150. Hua, W.; Liu, G. Transformer-based Networks Over Tree Structures for Code Classification. *Appl. Intell.* **2022**, *52*, 8895–8909. [CrossRef]
151. Phan, L.; Tran, H.; Le, D.; Nguyen, H.; Annibal, J.; Peltekian, A.; Ye, Y. CoTexT: Multi-task Learning with Code-Text Transformer. In Proceedings of the 1st Workshop on Natural Language Processing for Programming, Virtual, 6 August 2021; pp. 40–47.
152. Wang, X.; Wang, Y.; Mi, F.; Zhou, P.; Wan, Y.; Liu, X.; Li, L.; Wu, H.; Liu, J.; Jiang, X. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. *arXiv* **2021**, arXiv:2108.04556.
153. Kim, S.; Zhao, J.; Tian, Y.; Chandra, S. Code Prediction by Feeding Trees to Transformers. In Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering, Madrid, Spain, 22–30 May 2021; pp. 150–162.
154. Wang, Y.; Li, H. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. In Proceedings of the AAAI Conference on Artificial Intelligence, Virtual, 2–9 February 2021; pp. 14015–14023.
155. Drain, D.; Clement, C.B.; Serrato, G.; Sundaresan, N. Deepdebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons. *arXiv* **2021**, arXiv:2105.09352.
156. Ahmad, W.; Chakraborty, S.; Ray, B.; Chang, K.W. Unified Pre-training for Program Understanding and Generation. In Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Virtual, 6–11 June 2021; pp. 2655–2668.
157. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv* **2022**, arXiv:2203.13474.
158. Lajkó, M.; Csuvik, V.; Vidács, L. Towards Javascript Program Repair with Generative Pre-trained Transformer (GPT-2). In Proceedings of the 3rd International Workshop on Automated Program Repair, Pittsburgh, PA, USA, 19 May 2022; pp. 61–68.
159. Ye, H.; Martinez, M.; Monperrus, M. Neural Program Repair with Execution-based Backpropagation. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 1506–1518.
160. Xia, C.S.; Zhang, L. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 959–971.
161. Li, X.; Liu, S.; Feng, R.; Meng, G.; Xie, X.; Chen, K.; Liu, Y. TransRepair: Context-aware Program Repair for Compilation Errors. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, MI, USA, 10–14 October 2022; pp. 1–13.
162. Chakraborty, S.; Ahmed, T.; Ding, Y.; Devanbu, P.T.; Ray, B. NatGen: Generative Pre-training by "Naturalizing" Source Code. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 18–30.
163. Yang, G.; Chen, X.; Zhou, Y.; Yu, C. Dualsc: Automatic Generation and Summarization of Shellcode via Transformer and Dual Learning. In Proceedings of the International Conference on Software Analysis, Evolution and Reengineering, Honolulu, HI, USA, 15–18 March 2022.
164. Fu, M.; Tantithamthavorn, C.; Le, T.; Nguyen, V.; Phung, D. VulRepair: A T5-based Automated Software Vulnerability Repair. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 935–947.
165. Zhang, J.; Panthaplackel, S.; Nie, P.; Li, J.J.; Gligoric, M. CoditT5: Pretraining for Source Code and Natural Language Editing. In Proceedings of the International Conference on Automated Software Engineering, Rochester, MI, USA, 10–14 October 2022.
166. Tao, C.; Zhan, Q.; Hu, X.; Xia, X. C4: Contrastive Cross-language Code Clone Detection. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Virtual, 16–17 May 2022; pp. 413–424.
167. Niu, C.; Li, C.; Ng, V.; Ge, J.; Huang, L.; Luo, B. SPT-code: Sequence-to-sequence Pre-training for Learning Source Code Representations. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 2006–2018.
168. Yang, G.; Zhou, Y.; Chen, X.; Zhang, X.; Han, T.; Chen, T. ExploitGen: Template-augmented Exploit Code Generation based on CodeBERT. *J. Syst. Softw.* **2023**, *197*, 111577. [CrossRef]
169. Allal, L.B.; Li, R.; Kocetkov, D.; Mou, C.; Akiki, C.; Ferrandis, C.M.; Muennighoff, N.; Mishra, M.; Gu, A.; Dey, M.; et al. SantaCoder: Don't Reach for the Stars! *arXiv* **2023**, arXiv:2301.03988.
170. Li, R.; Allal, L.B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. StarCoder: May the source be with you! *arXiv* **2023**, arXiv:2305.06161.

171. Zhang, M.; He, Y. Accelerating Training of Transformer-based Language Models with Progressive Layer Dropping. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 14011–14023.
172. Han, X.; Zhang, Z.; Ding, N.; Gu, Y.; Liu, X.; Huo, Y.; Qiu, J.; Yao, Y.; Zhang, A.; Zhang, L.; et al. Pre-trained Models: Past, Present and Future. *AI Open* **2021**, *2*, 225–250. [CrossRef]
173. Lin, H.; Bilmes, J. *How to Select a Good Training-Data Subset for Transcription: Submodular Active Selection for Sequences*; Technical report; Washington University: Washington, DC, USA, 2009.
174. Liang, W.; Zou, J. MetaShift: A Dataset of Datasets for Evaluating Contextual Distribution Shifts and Training Conflicts. In Proceedings of the International Conference on Learning Representations, Virtual, 25–29 April 2022.
175. Yin, Y.; Chen, C.; Shang, L.; Jiang, X.; Chen, X.; Liu, Q. AutoTinyBERT: Automatic Hyper-parameter Optimization for Efficient Pre-trained Language Models. In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, Bangkok, Thailand, 1–6 August 2021 ; pp. 5146–5157.
176. OpenAI. CHATGPT: Optimizing Language Models for Dialogue. 2023. Available online: https://online-chatgpt.com/ (accessed on 16 May 2023).
177. Serban, I.V.; Sankar, C.; Germain, M.; Zhang, S.; Lin, Z.; Subramanian, S.; Kim, T.; Pieper, M.; Chandar, S.; Ke, N.R.; et al. A Deep Reinforcement Learning Chatbot. *arXiv* **2017**, arXiv:1709.02349.
178. Christiano, P.F.; Leike, J.; Brown, T.; Martic, M.; Legg, S.; Amodei, D. Deep Reinforcement Learning from Human Preferences. In Proceedings of the Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017.
179. Ling, L.; Tan, C.W. Human-assisted Computation for Auto-grading. In Proceedings of the IEEE International Conference on Data Mining Workshops, Singapore, 17–20 November 2018; pp. 360–364.
180. Ziegler, D.M.; Stiennon, N.; Wu, J.; Brown, T.B.; Radford, A.; Amodei, D.; Christiano, P.; Irving, G. Fine-tuning Language Models from Human Preferences. *arXiv* **2019**, arXiv:1909.08593.
181. Stiennon, N.; Ouyang, L.; Wu, J.; Ziegler, D.; Lowe, R.; Voss, C.; Radford, A.; Amodei, D.; Christiano, P.F. Learning to Summarize with Human Feedback. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 3008–3021.
182. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training Language Models to Follow Instructions with Human Feedback. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 27730–27744.
183. Hendler, J. Understanding the Limits of AI coding. *Science* **2023**, *379*, 548 . [CrossRef] [PubMed]
184. Chen, B.; Zhang, F.; Nguyen, A.; Zan, D.; Lin, Z.; Lou, J.G.; Chen, W. CodeT: Code Generation with Generated Tests. In Proceedings of the International Conference on Learning Representations, Virtual, 25–29 April 2022.
185. White, A.D.; Hocky, G.; Ansari, M.; Gandhi, H.A.; Cox, S.; Wellawatte, G.P.; Sasmal, S.; Yang, Z.; Liu, K.; Singh, Y.; et al. Assessment of Chemistry Knowledge in Large Language Models That Generate Code. *Digit. Discov.* **2023**, *2*, 368–376. [CrossRef] [PubMed]
186. Howard, J.; Ruder, S. Universal Language Model Fine-tuning for Text Classification. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, Melbourne, Australia, 15–20 July 2018; pp. 328–339.
187. Wei, J.; Bosma, M.; Zhao, V.; Guu, K.; Yu, A.W.; Lester, B.; Du, N.; Dai, A.M.; Le, Q.V. Finetuned Language Models are Zero-Shot Learners. In Proceedings of the International Conference on Learning Representations, Virtual, 25–29 April 2022.
188. Kingma, D.P.; Welling, M. Auto-encoding Variational Bayes. *arXiv* **2013**, arXiv:1312.6114.
189. Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. Generative Adversarial Networks. *Commun. ACM* **2020**, *63*, 139–144. [CrossRef]
190. Settles, B. *Active Learning Literature Survey*; University of Wisconsin: Madison, WI, USA, 2009.
191. Cohn, D.A.; Ghahramani, Z.; Jordan, M.I. Active Learning with Statistical Models. *J. Artif. Intell. Res.* **1996**, *4*, 129–145. [CrossRef]
192. Settles, B.; Craven, M.; Friedland, L. Active Learning with Real Annotation Costs. In Proceedings of the NIPS Workshop on Cost-sensitive Learning, Vancouver, BC, Canada, 8–13 December 2008.
193. He, J.; Vechev, M. Large Language Models for Code: Security Hardening and Adversarial Testing. *arXiv* **2023**, arXiv:2302.05319.
194. Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. Asleep at the Keyboard? Assessing the Security of Github Copilot's Code Contributions. In Proceedings of the IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 22–26 May 2022; pp. 754–768.
195. Peace, A.G.; Galletta, D.F.; Thong, J.Y. Software Piracy in the Workplace: A Model and Empirical Test. *J. Manag. Inf. Syst.* **2003**, *20*, 153–177.
196. Reavis Conner, K.; Rumelt, R.P. Software piracy: An Analysis of Protection Strategies. *Manag. Sci.* **1991**, *37*, 125–139. [CrossRef]
197. Limayem, M.; Khalifa, M.; Chin, W.W. Factors Motivating Software Piracy: A Longitudinal Study. *IEEE Trans. Eng. Manag.* **2004**, *51*, 414–425. [CrossRef]
198. De Laat, P.B. Copyright or Copyleft?: An Analysis of Property Regimes for Software Development. *Res. Policy* **2005**, *34*, 1511–1532. [CrossRef]
199. Kelty, C.M. Culture's Open Sources: Software, Copyright, and Cultural Critique. *Anthropol. Q.* **2004**, *77*, 499–506. [CrossRef]
200. The United States Copyright Office, Library of Congress. Copyright Registration Guidance: Works Containing Material Generated by Artificial Intelligence. 2023. Available online: https://www.federalregister.gov/d/2023-05321 (accessed on 26 April 2023).
201. Zheng, L.; Joe-Wong, C.; Tan, C.W.; Chiang, M.; Wang, X. How to Bid the Cloud. In Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM), London, UK, 17–21 August 2015; pp. 71–84.

202. Zheng, L.; Joe-Wong, C.; Brinton, C.; Tan, C.W.; Ha, S.; Chiang, M. On the Viability of a Cloud Virtual Service Provider. In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, Antibes Juan–les–Pins, France, 14–18 June 2016; pp. 235-–248.
203. Guo, S. INTITNI/CopilotForXcode: The Missing GitHub Copilot and ChatGPT Xcode Source Editor Extension. Available online: https://github.com/intitni/CopilotForXcode (accessed on 18 May 2023).