Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets

Hamed Jelodar, Mohammad Meymani, Roozbeh Razavi-Far
{h.jelodar,mohammad.meymani79,roozbeh.razavi-far}@unb.ca
Canadian Institute for Cybersecurity, Faculty of Computer Science, University of New Brunswick
Fredericton, NB, Canada

Abstract

Large language models (LLMs) and transformer-based architectures are increasingly utilized for source code analysis. As software systems grow in complexity, integrating LLMs into code analysis workflows becomes essential for enhancing efficiency, accuracy, and automation. This paper explores the role of LLMs for different code analysis tasks, focusing on three key aspects: *1) what they can analyze and their applications, 2) what models are used and 3) what datasets are used, and the challenges they face.* Regarding the goal of this research, we investigate scholarly articles that explore the use of LLMs for source code analysis to uncover research developments, current trends, and the intellectual structure of this emerging field. Additionally, we summarize limitations and highlight essential tools, datasets, and key challenges, which could be valuable for future work.

Keywords:Natural Language Processing, Large Language Models, Source Code, Pre-training, Transformers

ACM Reference Format:
Hamed Jelodar, Mohammad Meymani, Roozbeh Razavi-Far . 2025. Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets . In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In a broader context, large language models (LLMs) have been increasingly applied in the analysis of source code, leading to advancements in software engineering, bug detection, and code optimization [1]. LLMs are particularly useful for understanding code patterns, structure, and functionality. They help identify hidden patterns within large codebases, improve code quality, and automate tasks. By extracting semantic structures from complex source code, LLMs enable developers to better understand the behavior and logic of code, facilitating easier maintenance and enhancement. These models also play a key role in understanding how different pieces of code interact, much like how they are used to analyze interactions. LLMs are applied in various fields for source codes, including code summary [2–5] [6–8], code generation [9–12], showcasing their versatility and potential in both real-world application and research. For instance, in code summarization, the authors in [4] explored LLMs to generate concise natural language descriptions of code snippets. It investigates the effectiveness of different prompting techniques, evaluates various LLMs' code summarization abilities, and explores how model settings affect performance. The study finds that simpler prompting techniques like zero-shot prompting may outperform more advanced ones and that the impact of LLM settings varies by language. Additionally, LLMs struggle more with logic programming languages. Finally, they found that CodeLlama-Instruct with 7B parameters is better than other methods for code-summary tasks. Fig 1 show a process of code-summary based on an LLM model. Also, in other works, the authors in [13] analyzed biases in LLMs that unintentionally favored service providers like Google or Amazon during code generation. Using a dataset of 17,014 prompts, they found that these models exhibited strong preferences for these providers, even without specific user requests. This highlighted concerns about the biases in AI-generated content, particularly in coding applications.

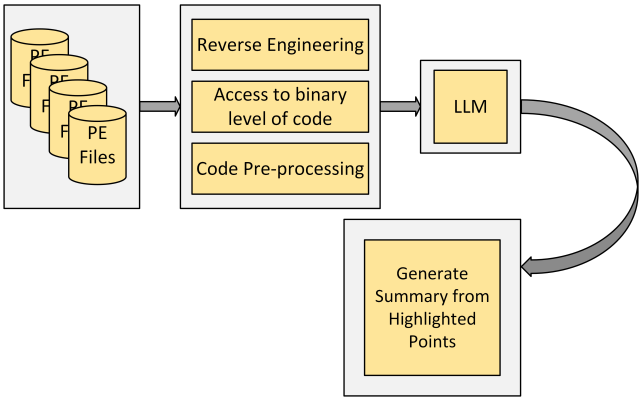

Figure 1: Binary-code summary using LLM

Despite the growing interest in utilizing LLMs for source code analysis, as evidenced by numerous recent papers, the research community still lacks a comprehensive survey that consolidates existing LLM-based code analysis studies, identifies the challenges encountered, and outlines future research directions [14]. We aim to fill this gap by conducting a systematic survey of LLM applications in various source code analysis tasks.

Conference acronym 'XX, Woodstock, NY

ACM ISBN XXXX-X/2025/03
https://doi.org/XXXXXXX.XXXXXXX

Generally, this survey will examine different perspectives, such as foundational theories, technical approaches, datasets, performance benchmarks, and future research opportunities. We believe that this survey will provide a clear overview of the current state of the field and highlight promising avenues for future advancements in this rapidly evolving research area.

Since LLMs have demonstrated exceptional capabilities in natural language processing, they are becoming increasingly important for understanding and analyzing source code. Our motivation is to investigate the application of LLMs in source code analysis, with a focus on various aspects such as models, datasets, and practical applications. The main goal of this work is to provide a comprehensive overview of how LLMs can be utilized for source code analysis. In summary, this paper makes four main contributions:

- We investigate scholarly articles that explore the use of LLMs for source code analysis to uncover research developments, current trends, and the intellectual structure of this emerging field
- We examine the applications of LLMs in various areas of code-analysis challenges, such as code generation, code decompiling and code summarization.
- We review and introduce prominent recent LLM models (Main/fine-tuned models) for code analysis.
- We introduce some of the most widely used datasets for LLM-based source code analysis, highlighting their strengths and limitations.

There are a few studies that could be related to our research [15–18]. For example, in [15], the authors investigated the role of LLMs in code understanding. In [16, 19, 20], only code generating ability of LLMs is addressed. In [17], code understanding and code generation are addressed. In [18], code summarization, code generation and security analysis are addressed. To the best of our knowledge, this is the first work to investigate different aspects of language models in code-related tasks, including code understanding, code disassembly, code decompilation, code summarization, code generation, comment generation, and security analysis.

Moreover, we provide a novel code-analysis taxonomy, classifying important LLMs according to their respective families as shown in Figure 5. Additionally, we include notable fine-tuned models in our taxonomy, emphasizing their specific enhancements and applications in various code-related tasks. In Figure 7, we present a comprehensive timeline of free datasets for code evaluation, showcasing their evolution and impact on the field. Furthermore, in Table 1, we illustrate the coverage of code-related tasks in our survey compared to related surveys, highlighting key gaps and contributions that differentiate our work.

Table 1: Our work vs. other surveys- Code Understanding (CU), Code Disassembling (CD), Code Decompiling (CdC), Code Summarization (CS), Code Generation (CG), Comment Generation (ComG), Security Analysis (SA)

Survey	Code-related Tasks						
	CU	CD	CdC	CS	CG	ComG	SA
[16]	✗	✗	✗	✗	✓	✗	✗
[19]	✗	✗	✗	✗	✓	✗	✗
[17]	✓	✗	✗	✗	✓	✗	✗
[15]	✓	✗	✗	✗	✗	✗	✗
[18]	✗	✗	✗	✓	✓	✗	✓
[20]	✗	✗	✗	✗	✓	✗	✗
Our Work	✓	✓	✓	✓	✓	✓	✓

The rest of the paper is organized as follows. In section 2, we provide a background on traditional tools, non-LLM Models, and LLM models for source code analysis. In section 3 we discuss a wide range of applications for source code analysis, providing a comprehensive table for each application. In section 4, we discuss the area of domain adaptation on code analysis. In section 5, we discuss the most popular models for source code analysis, providing a wide, novel taxonomy for these models. In section 6, we discuss free benchmarks for code analysis, providing a table and a timeline for these datasets. In section 7, we address the potential future gaps, discussing the current limitations of the existing models and providing potential solutions. Finally, in Section 8, we provide a summary of our findings.

2 BACKGROUND

This section explains NLP and LLM techniques for source code analysis, providing a background on how these methods can be applied to source code analysis.

2.1 Traditional tools for source code analysis

There are traditional tools that can be helpful for source code analysis. For real-time debugging and analysis, dynamic analysis tools and debuggers like OllyDbg[21] and x64dbg [22] enable analysts to inspect the behavior of running programs.

These tools provide the ability to interact with a program during execution, enabling users to modify its behavior, observe runtime data, and identify vulnerabilities or bugs.

Decompiling and disassembling tools [23, 24] are essential for reverse engineering compiled binaries, allowing analysts to inspect and understand the internal workings of a program. Disassemblers such as Ghidra [25] are widely used for converting binary code into assembly language, serving as a non-LLM method, as illustrated in Fig 2.

These tools break down machine code into human-readable assembly instructions, helping reverse engineers analyze and debug low-level program behavior. Disassemblers are especially valuable for understanding how a program operates at the CPU instruction level and are frequently used in malware analysis and security research [26]. Capstone [27] and Binary Ninja [28] are also key players in this category, providing lightweight disassembly capabilities for a range of CPU architectures .

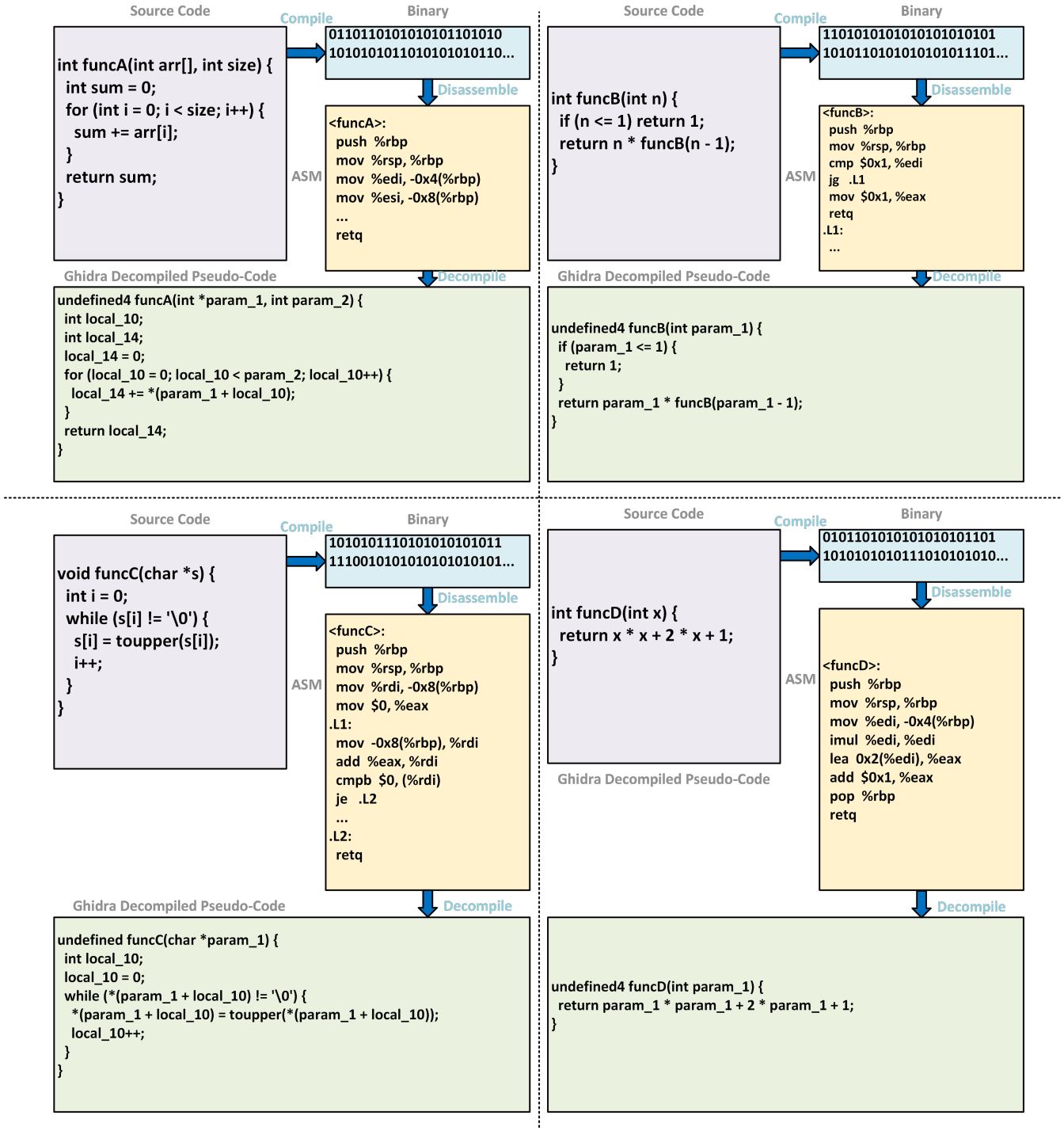

Figure 2: The Process of Code Decompiling in Different tasks using Ghidra.

2.2 Non-LLM models for code analysis

Traditional NLP methods for source code analysis focus on taking advantage of the structured nature of programming languages using statistical [29], rule-based [30], and classical machine learning techniques. One of the foundational steps is lexical analysis[31], where code is tokenized into meaningful units like keywords, operators, and identifiers. This process enables tasks like syntax highlighting, code formatting, and even plagiarism detection. In parallel, syntax analysis uses parsers to generate Abstract Syntax Trees (ASTs)[32], capturing the hierarchical structure of the code for applications such as syntax error detection and automated code completion [32–35] .

Moreover, semantic analysis examines the deeper meaning of the code, including type checking and symbol resolution [36]. Complementing this, feature engineering techniques manually extract metrics such as cyclomatic complexity, coupling, and dependencies to assess code quality and maintainability. Another important area is code similarity [37, 38] and clone detection [39, 40], which employs techniques like token-based comparisons and AST matching to identify duplicate code segments. Additionally, information retrieval techniques like cosine similarity [41, 42] enable effective code search and plagiarism detection, making them valuable for large-scale code repositories.

2.3 Transformer and LLM Models for Code

Transformers and LLMs have revolutionized source code analysis and generation, using their ability to process both natural and programming languages effectively [17, 19, 43–45]. Fig. 3 shows a general view of the application of LLMs for source code analysis based on different NLP techniques. In more detail, some researchers have used pre-trained techniques or LLM models for code analysis. For example, OpenAI's Codex [46] and GPT-4, along with specialized models such as CodeBERT [47] and GraphCodeBERT [48], are trained on large code datasets and related documentation.

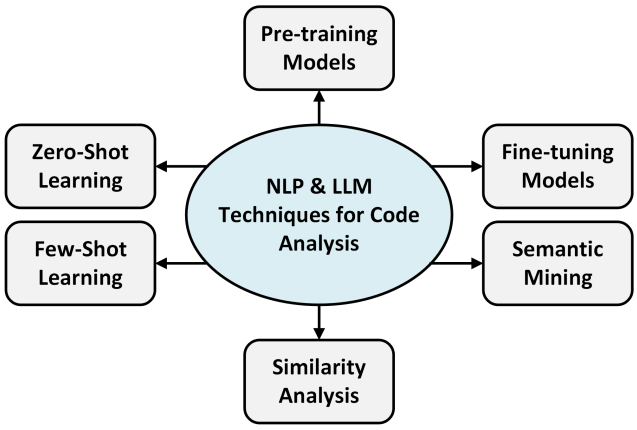

Figure 3: Application of LLM models for different tasks for source code analysis

Another notable model is Codex [49], which powers GitHub Copilot for intelligent auto-completion, enabling developers to code more efficiently. Meanwhile, CodeBERT incorporates program structure to enhance code search and summarization tasks. These models are capable of translating code between languages, refactoring legacy code, and generating documentation with minimal human input, ultimately reducing development time and boosting productivity. Additionally, specialized tools like StarCoder [50] and PolyCoder [51] extend AI-assisted coding capabilities to niche programming languages, thus broadening the impact of AI technologies beyond mainstream languages. However, in Section 5, we will discuss more about the recent pre-trained LLM models.

Also, some researchers used fine-tuning techniques on LLMs after pretraining on diverse datasets like GitHub [23], CodeSearchNet [5], and Stack Overflow[52]. This strategy enhances the models' ability to understand complex coding patterns and generate code for specific tasks. Additionally, they are valuable for tasks like code refactoring, performance optimization, and complex software design, offering widespread benefits across various industries.

2.4 Why are LLMs important for code understanding?

LLMs are essential for code understanding due to their ability to analyze, generate, and optimize code effectively. They help developers by identifying syntax errors [53], debugging issues [54], and improving code quality through suggestions for refactoring or adherence to best practices. LLMs also enhance productivity by automating repetitive tasks such as documentation generation, code completion, and creating boilerplate code. Additionally, they excel at explaining code in natural language, making complex concepts more accessible to new developers and teams.

Another significant advantage of LLMs is their support for advanced use cases like cross-language code translation [55–57], and testing automation [58, 59]. These capabilities enable developers to work across languages more efficiently and automate testing processes, saving time and improving software quality. As LLMs advance, their role in software development is set to grow, making them essential tools for modern development workflows [44, 60]. This capability bridges the gap between human and machine understanding, streamlining the development process and fostering collaboration.

3 Applications of LLM for source code analysis

LLMs demonstrate strong potential in streamlining tasks, ranging from code understanding to code summarization. By using LLMs, these models can effectively understand and generate programming languages, enhancing the efficiency of code-related tasks. Table 2 shows the applications of LLMs for source code analysis, based on previous works, highlighting key advancements in this field. In this Section, we discuss these LLM tasks for code analysis in more details.

Table 2: Application of LLM models for different tasks for source code analysis - Code Understanding (CU), Code Disassembling (CD), Code Decompiling (CdC), Code Summarization (CS), Code Generation (CG), Comment Generation (ComG), Security Analysis (SA)

App	Explanation
CU	Analyze and comprehend the logic and functionality of code.
CD	Uses LLM to reverse-engineer compiled code into readable formats.
CdC	Converts compiled code into high-level readable formats using LLM.
CS	Creates concise explanations of code functionality using LLM.
CG	Creates executable code from natural language inputs using LLM.
ComG	Creates descriptive comments to explain code functionality using LLM.
SA	Detects vulnerabilities and ensures code security using LLM.

3.1 LLM for Code Understanding

LLMs for code understanding represent an advanced computational approach rooted in natural language processing and machine learning research [1, 61, 62]. These models analyze the structural and semantic relationships within codebases to identify patterns, functionality, and underlying intent. Table 3 shows some significant works based on LLM for code understanding.

In [61], authors explored the use of a LLM-based conversational UI integrated into an IDE to aid code comprehension. They found that tool benefits vary between students and professionals, with usage influenced by learning styles and AI tool experience. In [62], the survey provides an overview of the benefits of incorporating code into LLMs' training data. It examines how code improves LLM performance and strengthens their role as intelligent agents (IAs).

They found that code training improved programming skills, reasoning, and the generation of formalized functions for diverse applications. It also enabled LLMs to interact with evaluation modules for self-improvement.

In [63], the authors studied the use of LLMs to support developers in code understanding. They designed an LLM-based conversational assistant that personalizes interactions based on inferred user mental states. The findings give useful insights for building LLM tools for beginners. In [64], The authors studied natural language (NL) outlines as a tool for AI-assisted software development. NL outlines, summarizes, and partitions code, enabling bidirectional updates between code and text. They found LLMs generate high-quality outlines, benefiting tasks like code understanding, maintenance, and generation. Authors introduced CodeMMLU to evaluate the code understanding capabilities of CodeLLMs using multiple-choice QA.

In [65], the authors studied the challenges in code generation using LLMs like ChatGPT and DeepSeek-Coder, focusing on the frequent misalignment of generated code with given specifications, particularly for complex programming tasks. To address this issue, they introduced a model, which combines thought-eliciting and feedback-based prompting strategies. In [66], the authors studied enhancing JavaScript source code comprehension by integrating graph alignment into LLMs. Using graph-based representations like ASTs, their model better captures structural and semantic relationships within the code.

Also other researchers [67], studied how code documentation quality affects LLMs code understanding. They found that incorrect documentation significantly hinders LLMs' comprehension, while incomplete or missing documentation has minimal impact. This suggests that LLMs rely more on code content than on documentation for understanding, highlighting the importance of accurate documentation to avoid misleading LLMs.

3.2 LLM for Code Disassembling

Disassembly is a particularly challenging task, especially when dealing with obfuscated executables. These executables often contain junk bytes, which are deliberately inserted to confuse disassemblers and induce errors during the analysis process [77]. Table 3 shows some significant work based on code disassembling. In this Section, we investigate and discuss the application of LLMs for code disassembling.

In [73], the authors explored using LLMs for disassembling obfuscated executables, aiming to improve instruction boundary identification. They developed a hybrid model combining traditional disassemblers with LLMs, fine-tuned using Llama 3 8B and the LLM2Vec method. This approach highlights the potential of LLMs in enhancing traditional disassembly techniques, particularly for handling obfuscation challenges.

Similarly, [75] introduced a model for assembly sequence evaluation in disassembly processes. This model aids in decision-making by utilizing event graphs and reinforcement learning, where LLMs play a critical role in defining the reward function. Deep Q-learning was employed with ground truth sequences, enabling improved decision-making during disassembly. This integration of LLMs demonstrates their utility in learning-based methods for evaluating and optimizing disassembly workflows. In another study, [78] focused on improving the quality of decompiled C++ code by refining decompiler outputs to produce recompilable and functional source code. The authors proposed a two-step approach based on LLMs: first, iteratively fixing syntax errors to ensure recompilability and second, identifying and correcting memory errors at runtime.

Additionally, [85] examined GPT-4's performance in binary reverse engineering through a two-phase study. The research assessed its ability to interpret human-written and decompiled code, followed by complex malware analysis. Using both automated metrics, such as BLEU scores, and manual evaluation, the study found that while GPT-4 demonstrates strong general code comprehension, its effectiveness diminishes in more intricate technical and security-focused analyses.

Moreover, [78] proposed Nova, a generative language model designed specifically for assembly code. To address the challenges posed by assembly code's low information density and diverse optimizations, Nova employs a hierarchical attention mechanism for semantic understanding and contrastive learning objectives to handle optimization variations. The model demonstrated superior performance in binary code decompilation and similarity detection compared to existing methods, showcasing LLMs' growing proficiency in specialized code domains.

Table 3: Models for Code Understanding (CU), Code Disassemble (CD), and Code Decompiling (CdC)

Reference	Task	Challenge	Models	Tokenizer/ Architecture	Dataset	GPU System
CodeMMLU [68]	CU	Evaluating the CU capabilities of CodeLLM using multiple-choice QA.	Claude-3-sonnet, GPT-4o, Meta-Llama-3-70B, Mixtral-8x7B, DeepSeek-moe-16b		10,000 multiple-choice questions covering diverse programming domains	
[69]	CU	Detecting subtle inconsistencies between code and its NL description	GPT-3.5 and GPT-4		HumanEval-X benchmark [70], encompassing six programming languages:	
CodeT5+[71]	CU	A flexible encoder-decoder LLM for code.		CodeT5/T5	Multilingual code corpora, CodeSearchNet	
[72]	CU	Real-world reverse engineering scenarios and binary code understanding.	CodeLlama, DeepSeek, CodeGen, Mistra, Vicuna, ChatGPT	CodeLlama, DeepSeek, CodeGen, Mistra, Vicuna, ChatGPT		
[73]	CU	Judging the correctness of provided code solutions.	DeepSeek, CodeGen		APPS (Hendrycks et al., 2021) dataset [74]	
[75]	CU	Understanding long-context code.	Mistra, Vicuna, ChatGPT		RepoQA benchmark [72]	
DISASLLM [73]	CD	Combining the traditional disassemblers with LLMs	-Masked next token prediction (MNTP) -Llama Model	Llama3		NVIDIA H100 GPU, AMD EPYC 9124 16-core
[75]	CD	Assembly sequence evaluation in disassemble process	Deep Q-learning, GPT-3.5-turbo	GPT	Google Code Jam (GCJ) [76]	
[77]	CD	Binary Reverse Engineering	GPT-4	GPT	Google Code Jam	
Nova$^+$ [78]	CD	A generative LM for assembly code.	DeepSeek-Coder	DeepSeek-Coder	-AnghaBench [79] and The Stack for pretraining -BinaryCorp-3M [80] for fine-tuning	NVIDIA RTX A100 GPUs
LM4Decompile [81]	CdC	Decompilation of binary code into high-level source code	Sequence-to-sequence prediction (S2S)	DeepSeekCoder	ExeBench [82]	NVIDIA A100-80GB GPU
DeGPT [26]	CdC	Refining decompiled C code	GPT-3.5-turbo-0613"	GPT	Code Contest dataset	
[83]	CdC	Binary code analysis	Lora, LLM4Decompile-6.7b, DeepSeek-chat, GPT4, LLM4Decompile-6.7b+FAE, (LlamaFactory, FlashAttention used for fine-tuning)	Fine-tune the LLM4Decompile-6.7b		A100-SXM4-80GB
[84]	CdC	-Binary tasks -Binary Summarization -Binary function name recovery tasks	GPT4, CodeLlama-7b, CodeLlama-13b, CodeLlama-34b, GPT3.5, Claude3, Gemini-Pro	GPT4		

Collectively, these studies highlight the transformative potential of LLMs in code disassembly and related tasks, offering new avenues for handling obfuscation, improving decompiled outputs, and advancing binary reverse engineering.

3.3 LLM for Code Decompiling

Using LLMs for code decompiling improves both comprehension and generation of code. Accurate documentation is essential for optimizing LLM performance in this process [26, 84]. Table 3 shows some significant works based on code decompiling. In this Section, we investigate and discuss the application of LLMs for code decompiling.

In [81], the authors introduced LLM4Decompile, a series of open-source models (ranging from 1.3B to 33B parameters) designed for binary code decompilation, addressing the limitations of traditional tools like Ghidra. The authors employed DeepSeekCoder, an

advanced architecture specifically designed for binary code decompilation, optimizing the LLM training process to enhance code readability and executability. Additionally, they introduced two variants: LLM4Decompile-End for direct decompilation and LLM4Decompile-Ref for refining Ghidra outputs. These advancements showcase the transformative potential of LLMs in the decompilation process, enhancing the overall efficiency and accuracy of code translation from binary to high-level languages.

In [26], the authors presented DecGPT, a two-step approach using GPT-3.5-turbo-0613 to refine decompiled C code. The first step focuses on correcting syntax errors, and the second step addresses memory errors that occur during runtime. This approach demonstrates the ability of LLMs to handle common decompilation challenges and improve the quality of the decompiled code, making it more readable and functional for developers.

In other work [24], authors introduced WaDec to investigate the challenges in decompiling WebAssembly (Wasm) binaries, which are compact yet difficult to analyze and interpret. This fine-tuned LLM model converts Wasm binaries into comprehensible source code. The authors found that WaDec significantly improved the decompilation of WebAssembly binaries, paving the way for more efficient analysis of this widely used format. In [26], the authors presented additional techniques for improving decompilation processes, including Self-Constructed Context Decompilation (sc2 dec), which uses in-context learning, and Fine-grained Alignment Enhancement (FAE), which aligns assembly and source code at the statement level using debugging data. Their model, fine-tuned using the LLM4Decompile-6.7B model, further enhances the decompilation process by improving the accuracy of the reconstructed source code.

In [84], the authors proposed a framework for binary analysis. They evaluated various LLMs, with GPT-4 as the primary model for their framework, demonstrating the potential of LLMs in binary code analysis. Alongside these advancements, tools like PYLINGUAL, a Python library for decompiling, also contribute to the decompiling landscape, offering additional resources for handling complex decompilation tasks. Basically, these studies collectively highlight the growing role of LLMs in code decompiling, showcasing their ability to improve the readability, accuracy, and efficiency of decompiled code, as well as their potential to address challenges in decompiling various binary formats.

3.4 LLM for Code Summarization

Automated code summarization focuses on generating natural language summaries for code fragments, such as methods or functions [2, 4, 86]. In general, a code summary is a brief description, typically one sentence, that plays a key role in developer documentation [86]. In this Section, we investigate and discuss the application of LLM for code summarization.

In [2], the authors conducted a study where they distilled GPT-3.5's code summarization abilities into smaller, more accessible models. They carefully examined the role of model and dataset sizes in the knowledge distillation process for code summarization. The authors fine-tuned a pre-trained language model to generate code summaries using genersted-examples by GPT-3.5 as training data.

In [2], the authors studied how to simplify GPT-3.5's code summarization abilities into smaller models. They explored the impact of model and dataset sizes in the distillation process and fine-tuned a pre-trained model to generate code summaries using samples from GPT-3.5 as training data.

In [3], the authors explored how LLMs interpret binary code semantics through a large-scale study. The evaluation included models like ChatGPT and Llama 2 reflecting a broad examination of LLM capabilities. Similarly, in [4], researchers investigated LLM-based code summarization, examining different prompting techniques and their impact on performance. They found simpler approaches, such as zero-shot prompting, often outperformed more complex methods, though results varied across programming languages. In particular, CodeLlama-Instruct with 7B parameters emerged as a strong performer, though challenges remained with logic programming languages.

In [5], the authors focused on automatic semantic augmentation of prompts. By integrating tagged semantic facts, they helped LLMs better structure their reasoning when generating code summaries. Tested on the CodeSearchNet dataset [87], this method improved performance in both summarization and code completion tasks. Additionally, [88] presented a context-aware approach, designing prompt templates for six common contexts. They created a diverse, context-focused dataset and fine-tuned two pre-trained code models to produce meaningful and inclusive code comments.

In [89], the authors introduced the use-seq loss function for source code summarization, prioritizing semantic similarity at the sentence level. They demonstrated its advantage over traditional categorical cross-entropy, fine-tuning models like LLaMA and evaluating them using BLEU, METEOR, and human ratings on a funcom-java-long dataset. In addition, authors [90], proposed improving code-summary alignment using a multitask learning approach. They developed three summary-specific tasks: masked language modeling (MLM) [91] , and unidirectional language modeling (ULM), action word prediction (AWP), enhancing the encoder's effectiveness in creating inclusive and meaningful summaries.

3.5 LLM for Code Generation

LLMs allow automatic code generation from natural language inputs [92, 93]. Table 6 shows some significant works based for code generation. In this section, we investigate and discuss the application of LLM for code generation.

In [9], the authors studied the non-determinism in code generation, specifically examining the variability in its outputs when identical prompts were used. This research aimed to quantify this non-determinism and understand its impact on the reliability and reproducibility of LLMs in software engineering. By comparing the semantic, syntactic, and structural differences in the generated code

under varying temperature settings, they found significant variability. This study highlighted the challenges that non-determinism poses for developers and researchers when using ChatGPT in coding tasks.

In [12], the authors analyzed social biases in code generated by LLMs like GPT-4 and GPT-3.5-turbo, focusing on age, gender, and race. They found 13.47% to 49.10% of outputs displayed gender bias using a novel bias testing framework. To address this, they evaluated mitigation strategies such as zero-shot, one-shot, few-shot, and Chain-of-Thought prompts [94], both with and without feedback refinement. While direct prompt engineering had limited success, feedback-driven strategies significantly reduced bias, cutting GPT-4's bias rate from 59.88% to 4.79%. This highlights the importance of execution feedback in reducing biases in code generation.

In [92], the authors created FlowGen, a framework that uses LLM agents to simulate software process models. These agents refine code through reasoning and prompts. Tested with GPT-3.5 on benchmarks like HumanEval, adding CodeT to FlowGenScrum achieved the best Pass@1 scores.

In [95], the research also focused on TDD for code generation, they investigated if and how Test-Driven Development(TDD) can be incorporated into AI-assisted code-generation processes. For experiment they used GPT-4, Llama 3 methods based on MBPP and HumanEval datasets. Further exploration of TDD in LLM-based code generation suggests that combining traditional software engineering practices with AI-driven approaches can significantly improve the quality of generated code. Researchers have pointed out that incorporating a testing-first approach, such as TDD, helps in ensuring that the code not only meets functional requirements but also adheres to desired performance and stability metrics.

3.5.1 Code generation and security. There are some researches that focused on the security of code generation by LLMs. For instance, in [14] focused on evaluating the security of AI-generated code by introducing CodeSecEval, a dataset with 180 samples covering 44 critical vulnerability types. They assessed LLMs on both code generation and repair tasks, revealing that these models often produce insecure code due to training on unsanitized open-source data.

Also, in [96], the authors proposed a framework to assess LLM-generated code for security and functionality. They introduced CWEval-Bench, a multilingual benchmark with 119 tasks covering 31 CWE types. Their evaluation showed that many functionally correct code outputs are insecure.

3.5.2 Code generation and explanation. LLMs not only are revolutionizing code generation, but also are quite useful in generating clear explanation for code. Tools based on LLM technology, enable developers to receive code summary and explanation only by highlighting the specific part of the code without any need to query a prompt. Studies have indicated that such tools improve task completion rates, especially for experienced developers, by reducing the load related to manual searches through documentation [15].

3.6 LLM for Comment Generation

Code review is a key part of modern software development, helping to improve both the quality of systems and the skills of developers. Table 8 shows some significant works based for comment generation. Recently, studies have explored LLM-based methods to automatically generate review comments, making the process more efficient. In this section, we investigate and discuss the application of LLMs for comment generation [97–103]

In [97], the authors studied the integration of LLMs into the code review process, focusing on their ability to generate review comments and their acceptance by human reviewers. They conducted a large-scale empirical user study within two organizations—Mozilla (open-source) and Ubisoft (closed-source)—to evaluate the effectiveness and reception of LLM-generated comments in real-world settings. The model used for this study was based on OpenAI's GPT-4. Accepted comments generated by LLMs were just as likely to prompt future revisions of the modified patch as human-written comments, indicating that LLM-generated feedback can play a valuable role in the code review process. This highlights the potential for LLMs to automate and enhance the efficiency of code reviews.

In [104], the authors fine-tuned Llama models with QLoRA to improve code review comments and explored enhancing inputs with function call graphs and summaries, highlighting the value of added code context. This work emphasizes the potential of augmenting LLMs with additional code context to improve comment quality.

In [99], the authors proposed SCCLLM, which retrieves top-k code snippets for in-context learning to generate smart contract comments. Their study highlights LLMs' potential in automating code review and comment generation.

3.7 LLM for Security Code Analysis

LLMs generate insecure code by default, up to 65% of the generated codes contain vulnerabilities. Table 4 shows some significant works for LLM in security code. However, a skilled engineer can manually guide an LLM to generate a 100% secure code. To gain the best possible results from an LLM prompts need to be optimized. The prompts need to be both interpretable for an AI agent to comprehend and precise to gain better results. Moreover, breaking a complex prompt into multiple simple prompts can generally lead to higher performance [105].

Code (Security) analysis is the process of detecting and fixing vulnerabilities in a code to prevent attacker's from breaching the system. Code analysis could be either static or dynamic. In static code analysis, logic and syntax faults are handled. On the other hand, in dynamic code analysis, the code is analyzed during the run time to solve memory problems, bottlenecks and so on [106].

LLMs can enhance security code analysis by detecting vulnerabilities early in the development phase [107]. They help identify bugs and insecure coding practices, such as weak authentication or improper data handling, and provide recommendations for improving code security [108].

For example, some researchers In [109] focused on evaluating the security vulnerabilities of Large Language Model-based Code Completion Tools (LCCTs), specifically GitHub Copilot and Amazon Q. They investigated two primary threats: jailbreaking attacks, which bypass built-in safety mechanisms to generate harmful or unethical code, and training data extraction attacks, where sensitive information from the model's training data is unintentionally exposed. The study serves as a call for greater scrutiny in deploying LLM-powered code completion tools, ensuring they adhere to stricter security and ethical guidelines.

Also in [**?**], The authors focused on evaluating the security and quality of code generated by Large Language Models (LLMs) across multiple programming languages. They analyzed models like GPT-4o, Claude-3.5, Gemini-1.5, Codestral, and LLaMA-3 using a dataset of 200 programming tasks, categorized into six groups. Their primary goal was to assess how well these models adhere to modern security practices and whether they produce maintainable, high-quality code. Their findings indicate that while LLMs are effective in automating code generation, their security performance varies significantly across languages.

3.7.1 Frameworks. In order to securely generate a source code using an LLM, some frameworks have been proposed, including SecCode [107], SALLM [110] and LLMSecGuard [111] to do the prompt engineering task in an automatic manner.

SecCode operates in 3 main stages, code generation, code vulnerability detection (code analysis) and fixing, and code security refinement. The process begins when the user enters prompt p_1 and code C is generated using an LLM model like ChatGPT. The generated code is not necessarily secure. In the second stage, an automated prompt P_2 is generated by the system leading to the detection and fixing vulnerabilities using a rewarding system called Encouragement Prompting (EP). EP rewards the system if a vulnerability is found and fixed. On the other hand, EP penalizes the system if some new vulnerabilities are introduced or the existing vulnerabilities remain after fixing process [107].

Moreover, after initial fixes, the code undergoes further security checks using CodeQL. If CodeQL finds additional hidden vulnerabilities, a prompt $p3$ is automatically generated leading to further code security enhancement by LLM and this process repeats over and over until CodeQL can't find anymore vulnerabilities [107].

SALLM includes 3 components, a dataset of python prompts, a code generation and repair module, and a security evaluation system on model's performance. At first, a dataset of security-centric is created by collecting prompts from different sources. These prompts are used to generate code by feeding them into the LLM, which returns the top k results for each prompt. After that, SALLM evaluates the security of the generated code using static and dynamic analysis. Finally, SALLM iteratively refines the code until no vulnerability is found [110].

LLMSecGuard takes the user's prompt, enhances it, and forwards it to an LLM to generate the code. Then, it sends the code static code analysis engines including Semgrep and Weggli to measure how secure the code is. This process is repeated throughout the iterations until certain conditions are met [111].

3.7.2 Security and code Documentation. The quality of code documentation is another crucial factor influencing the effectiveness of LLMs assisting developers. Accurate and detailed documentation improves LLM performance, while incorrect or misleading documentation can significantly hinder code understanding task. Surprisingly, the absence of documentation often has less of a negative impact compared to flawed comments or descriptions, meaning that the quality of the documentations and comments should be strictly preserved [65].

We can observe that these frameworks work along with existing LLMs and code analysis systems. So, they do not inherently contain these modules. In table 4, a comparison between some recent security analysis frameworks is shown:

Table 4: Recent Security Code Generation frameworks

Framework	Models	Code Analysis Tool	Dataset	Language(s)
SecCode [107]	GPT-3.5 DeepSeek-Coder	CodeQL	LLMSecEval [112] SecurityEval [113] Holmes [114]	C Python
SALLM [110]	CodeGen models StarCoder GPT models	CodeQL	LLMSecEval [112] SecurityEval [113] SALLM [110]	C Python
LLMSecGuard [111]	Llama2	Semgrep Weggli	CyberSecEval [115]	Java

Table 5: Recent models for code summarization

Ref	Year	Challenge	Models	Tokenizer/ architecture	Dataset	GPU System
[2]	2024	-Fine-tuning for code summarization	GPT-3.5	GPT	-Collected summaries for 2.15m Java	
[3]	2024	LLMs' understanding of binary code	- GPT-4, - ChatGPT (GPT-3.5), - Llama 2 - Code Llama - BinT5		44 open-access software project	NVIDIA A100 GPU
[3]	2024	LLM evaluation for code summary	CodeBERT -GraphCodeBERT -Polyglot CodeBERT -Polyglot GraphcodeBERT - CodeT5 -CodeT5+	-	CodeSearchNet	
[4]	2024	A heuristic for ChatGPT's prompts.	NCS, CodeBERT, and CodeT5 ChatGPT		CSN-Python dataset CodeSearchNet	
[86]	2024	Exploring ChatGPT's summarization	ChatGPT			
[116]	2024	Fine-tuning for code summarization	Llama 7B	Llama 7B	Funcom-java-long dataset.	AMD 5900X CPU, 128GB memory, and two Nvidia A5000 GPU
[88]	2024	Fine-tuning for code summarization	Fine-tune the Llama2-7B	Llama2-7B m	Python code dataset	NVIDIA A100-SXM 40GB-RAM GPU
[117]	2024	Fine-tuning for code summarization	CodeLlama-7B-instruct	CodeLlama-7B-instruct mode	http://2https//www.kaggle.com/datasets /pelmers/github-repository-metadata -with-5-stars/versions/83 https://clang.llvm.org/	NVIDIA V100 16Gb GPU
[90]	2024	-Automatic code summarization -Conducted experiments on an Ericsson software project	Llama-70B CodeLlama Mixtral LLM		Ericsson and open-source projects (Guava and Elasticsearc)	Tesla A100 GPU
[118]	2024	Evaluate code summary similarity	roBERTa, MPNet		2 million code summaries (pre-0t).	d 4 NVIDIA A100 GPUs, each with 80 GB of VRAM

Table 6: Recent models for Code Generation

Reference	Year	Challenge	Models	Tokenizer /architecture	Dataset	GPU System
[9]	2024	Code generation	ChatGPT	GPT/ChatGPT	CodeContests [119], APPS, and HumanEval	
[10]	2024	Prompt engineering in code bias	PALM-2/GPT			
[9]	2023	Hallucinations in code generation	ChatGPT CodeLLama-7B5, DeepSeek-Coder-7B6	-	HumanEval HALLUCODE [11]	
FlowGen [12]	2024	Emulating software with LLMs.	GPT-3	GPT	HumanEval, HumanEval-ET, MBPP (Mostly Basic Python Programming),	
TiCoder [120]	2024	Improve code suggestion accuracy.	OpenAI code-davinci-002, text-davinci-003, OpenAI GPT-3.5-turbo, GPT-4-turbo, GPT-4-32k, Salesforce CodeGen-6B, CodeGen2.5-7B	-	Two Python datasets, HumanEval and MBPP	
ClarifyGPT [92]	2024	Enhance code generation by identifying ambiguities	GPT-4 and ChatGPT	GPT	MBPP-sanitized , HumanEval, HumanEval-ET and MBPP-ET, CoderEval [121]	-
[122]	2024	Code generation with image recognition.	GPT-4	GPT		
[93]	2024	LLM search for code generation.	OpenAI's GPT-4o-mini a.	OpenAI's GPT-4o-mini	MBPP+, HumanEval+ 28 , and LiveCodeBench MBPP	
Deceptprompt [123]	2023	Instructions for code	1) CodeLlama7B 2) StarChat-15B (fine-tuned on StarCoder15B); 3) WizardCoder-3B and 4) WizardCoder15B. T		A dataset that covers 25 different CWE types	
NL2ProcessOps [124]	2024	Retrieval Augmented Generation (RAG) to streamline code generation.	GPT-4 (gpt-4-0125-preview)	GPT-4	GitHub Copilot	
[125]	2024	Improved cost-accuracy trade-offs.	Codegen-mono, WizardCoder-V1.0, WizardCoder-Python-V1.0		HumanEvalm, MBPP-sanitized, APPS-test	NVIDIA Geforce RTX 3090 GPUs, each with 24GB of VRAM
[126]		Code generation for audio programming.	MaxMSP, gpt-4-0125-preview			

Table 7: Models for pre-training and LLMs

Reference	Details	Architecture/Pre-trained Model	Pre-trained Domain/Dataset	GPU System
AnchorCoderP [23]	A pre-trained model based on Llama	CodeLlama-7B	CodeSearchNet	4 NVIDIA A40 48GB GPUs
MONOCODER [127]	Smaller LMs for specific domains	GPT-3.5 r	HPC-specific dataset	4 NVIDIA A40 48GB GPUs
CommitBART [128]	Based on GitHub commits	BART [27] architecture	GitHub commits/ 7 programming languages	
[129]	Improving logic and instruction-following.	GPT2, LLaMA (Tiny-Llama v1.1)	Ten programming languages and other datasets (Wikipedia)	8 H100 GPUs
[130]	Code-aware program repair	GPT	A very large software codebase 4.04 million methods from 1,700 open-source projects	one 56-core server with one NVIDIA TITAN V and three Xp GPUs
TreeBERT [131]	An LLM for programming language	TreeBert/Bert	Python and Java corpus (CuBERT)	
NatGen [132]	An LLM for code generation	CodeT5	CodeSearchN	2 Nvidia GeForce RTX 3090 GPUs
[133]	Evaluating pre-trained transformers	seBERT	Stack Overflow, Jira, GitHub,	8x NVIDIA Tesla V100 32G GPUs

Table 8: Recent Models for comment generation

Reference	Year	Challenge/Goal	Models	Tokenizer /architecture	Dataset	GPU System
RevMate [97]	2024	Comment generation	GPT4o	GPT		
AUTOGENICS [52]	2024	Inline comment generation	Gemini 1.5 Pro - GPT-4		400 code snippets (200 Python + 200 Java)	
[99]	2024	Improving review comments.	Llama models -GPT-3.5 QLoRA technique to fine-tune - CodeT5	GPT	CodeReviewer dataset [134]	NVIDIA RTX 5000 GPU machine with 16 GB VRAM
[100]	2024	Multi-intent comment generation	Codex model code-davinci-002.		Funcom and TLC	
SCCLLM [99]	2024	Automatic comment generation	CodeBERT		A large corpus from Etherscan.io	GeForce RTX4090 GPU (24GB graphic memory)

4 Pre-training for domain adaptation on code analysis

Pre-training involves initially training a model on a large, general-purpose dataset, such as predicting the next word in a sequence. Following pre-training, the model undergoes fine-tuning on a smaller, domain-specific dataset to optimize its performance on specialized tasks like text generation. Figure 4 illustrates the pre-training pipeline for LLMs. As indicated in Table 7 and corroborated by prior research, datasets like CodeSearchNet, HPCorpus, and commit data significantly boost the model's ability to understand and perform in specific domains, such as code generation. For instance, in [135], the authors introduced AnchorCoder, a novel approach designed to enhance LLMs for code generation while minimizing computational resource requirements. AnchorCoderP [135], pre-trained on CodeSearchNet from scratch, was evaluated across various language modeling tasks, revealing attention weight sparsity patterns where information consolidated at distinct anchor points.

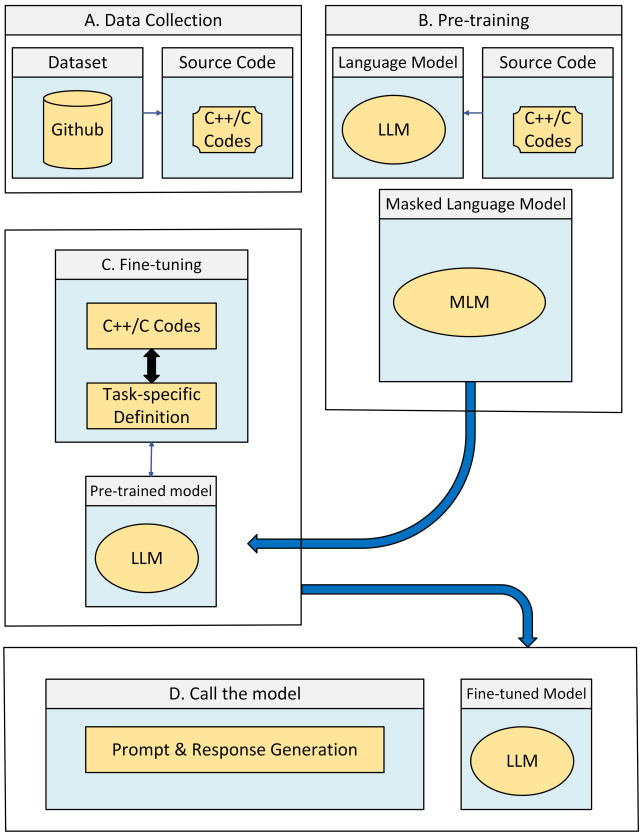

Figure 4: A general view of the pre-training of an LLM based on masked language modeling

In [127], the authors proposed MonoCoder, a domain-specific language model tailored for high-performance computing (HPC) code and tasks. Pre-trained on HPCorpus—a dataset comprising C and C++ programs mined from GitHub—MonoCoder is focused exclusively on HPC-related code. In [128], the authors examined GitHub commits, which combine code changes with descriptive natural language messages to aid software developers in understanding software evolution. They pre-trained an LLM based on the BART model using 7.99 million commits across seven programming languages, resulting in a method called CommitBART. Similarly, in [129], the authors examined the effects of pre-training LLMs, such as GPT-2 (GPT2-Large) and TinyLlama v1.1, on programming languages versus natural languages for logical inference tasks. Decoder-based models were trained from scratch on datasets containing different programming languages, including Python, C, and Java, and other datasets (such as Wikipedia,) all under the same conditions.

In a different study, [130] focused on pre-training and fine-tuning a GPT model with a large software codebase for automatic program repair (APR). They implemented a code-aware search strategy that prioritized compilable patches and solutions aligned with the length of the buggy code. Evaluations on the Defects4J [136] and QuixBugs [137] benchmarks demonstrated that CURE outperformed existing APR methods, fixing 57 and 26 bugs, respectively.

In [131], the authors introduced TreeBERT, a tree-based model aimed at improving tasks like code summarization and documentation. TreeBERT was pre-trained on datasets with different programming languages, using a hybrid objective that combined Tree Masked Language Modeling (TMLM) with Node Order Prediction (NOP). Additionally, in [132], the authors presented a new pre-training objective called Naturalizing for source code models built on CodeT5. Fine-tuned for tasks such as code generation, translation, and refinement, this model achieved state-of-the-art performance, particularly in zero-shot and few-shot learning tasks.

Furthermore, in [133], the authors evaluated the performance of pre-trained transformer models in software engineering tasks, including code summarization, bug detection, and understanding developer intent. They pre-trained BERT and SBERT on various datasets such as StackOverflow, GitHub issues, and Jira issues. Comparing BERT models (base and large) trained on software engineering data (e.g., code corpora) with those trained on general-domain data (e.g., Wikipedia). Table 10 shows the performance of different BERT models for [Mask] prediction. The study found that domain-specific pre-training yielded better results for tasks like bug prediction and code summarization.

Generally, domain-specific pre-training for LLMs has emerged as a critical strategy for achieving superior performance in specialized tasks. By tailoring models to understand the unique nuances of a domain, researchers can unlock efficiencies and capabilities that general-purpose models may not achieve. These advancements not only enhance task-specific accuracy but also open up opportunities for groundbreaking applications, particularly in areas where domain expertise is paramount.

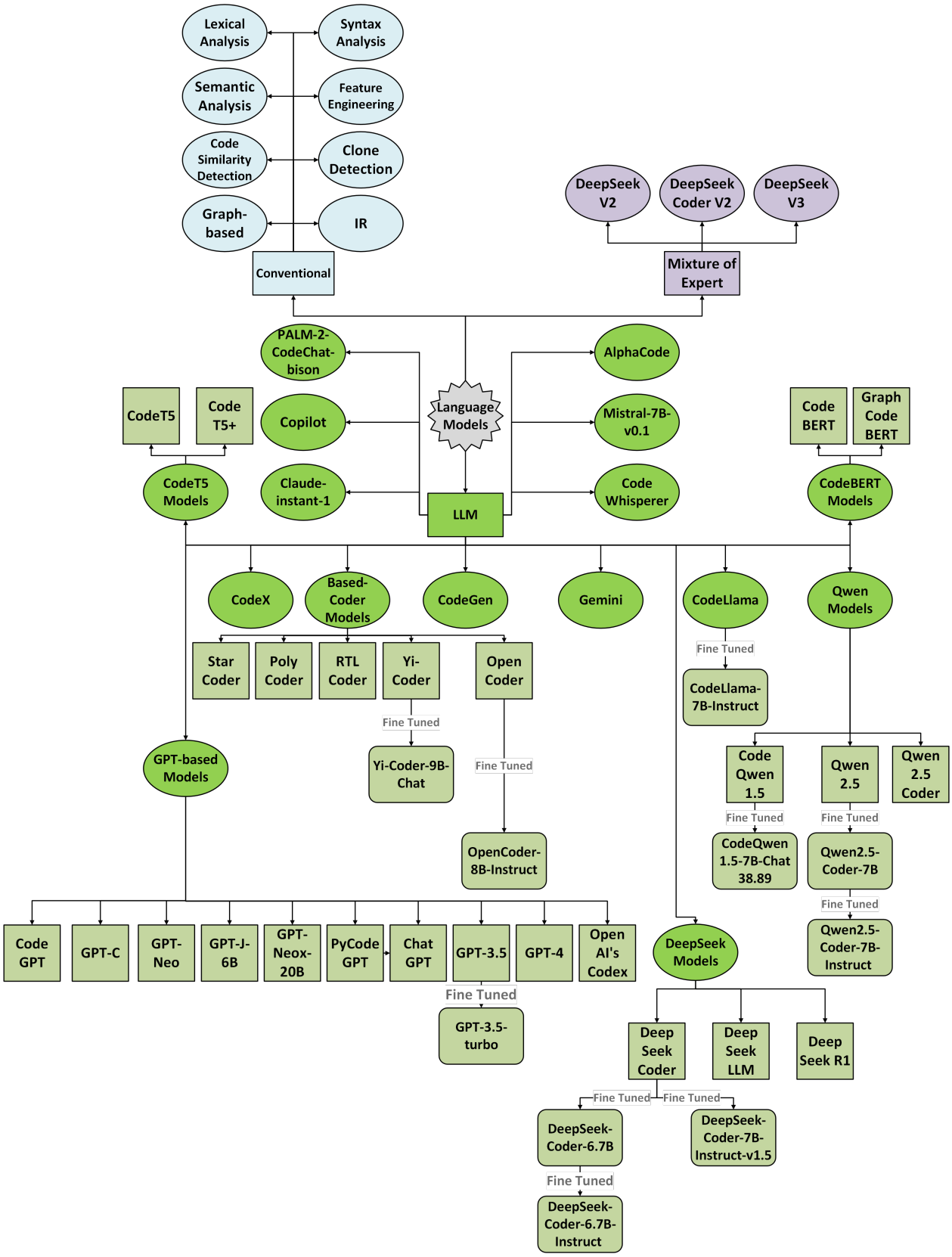

Figure 5: A taxonomy of recent NLP and LLM models for source code analysis

5 Popular models: language models for source code analysis

Although there are many LLM models for source code analysis, in this Section, we will focus only on the most famous models based on previous works which include CodeBERT, CodeT5, GPT, and DeepSeek models. In Figure 5, a taxonomy of language models for code is shown. Also, in figure 6 we bring the most important LLMs based on their technical report date.

5.1 CodeBERT

CodeBERT is a pre-trained bimodal transformer model for programming language (PL) and natural language (NL) tasks, such as documentation generation and code search. It outperforms models like RoBERTa by combining replaced token detection (RTD) and masked language modeling (MLM) during training. CodeBERT uses both NL-PL pairs and unimodal data, boosting performance [47].

5.2 CodeT5 Models

CodeT5-based model are specially designed for code-related tasks [138]. These models include CodeT5 and CodeT5+.

CodeT5 is a Transformer-based model, built on T5 architecture, which is applicable to both generating and understanding the code. CodeT5 employs multi-task learning enabling it to perform a variety of code-related tasks such as flaw detection, code translation and summarization. CodeT5 introduces identifier-aware pre-training objectives using crucial structural and semantic information in programming languages (PL). CodeT5 outperforms models like CodeBERT, PLBART, and GraphCodeBERT in multiple metrics across code-related tasks [138].

CodeT5+ is an extension to CodeT5, which is more flexible in architecture. Unlike CodeT5, it uses 3 architectures including encoder-only, decoder-only and encode-decoder. [68].

5.3 Generative Pre-trained Transformer (GPT) Models

The family of GPT models encompass different implementations including GPT-C [139], CodeGPT [140], GPT-Neo [141], GPT-J-6B [142], GPT-NeoX-20B [143], PyCodeGPT [144], ChatGPT [145], GPT-4 [146] and OpenAI's Codex. Here we explain OpenAI's Codex [46] and GPT-4 in more detail.

OpenAI's Codex is an open-source, GPT model written in Python which is evaluated on the HumanEval dataset, outperforming models like GPT-3 and GPT-J in functional correctness, solving 28.8% of tasks with a single sample and up to 70.2% with repeated sampling strategies to overcome complex prompts [46].

GPT 4 is a large-scale multi-modal model processing both text and image inputs and returning text outputs. It competes human-level performance on numerous tasks. The model achieves a better performance compared to previous versions, particularly on diverse language and reasoning tasks. GPT-4's high-level capabilities are achieved with advancements in pre-training and fine-tuning using reinforcement learning from human feedback (RLHF) [146].

5.4 DeepSeek Models

There are number of models implemented for DeepSeek family. We introduce the most important ones including DeepSeek-Coder [147], DeepSeek LLM [148], DeepSeek-V3 [149] and DeepSeek-R1 [150].

DeepSeek-Coder is a series of open-source language models, ranging from 1.3B to 33B parameters, optimized for code intelligence in software development. Supporting 87 programming languages and using a 16K token context window, these models excel at tasks like code generation and completion, outperforming GPT 3.5 and Codex [147, 148].

DeepSeek-V3, a Mixture-of-Experts (MoE) model with 671 billion parameters, uses Multi-head Latent Attention (MLA). It rivals top models like GPT-4 and Claude 3.5, with enhanced performance on Chinese tasks through Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL). Its advanced optimization sets a new benchmark for open-source models, offering high performance with resource efficiency [149].

5.5 Qwen Models

There are various types of models in Qwen family, Qwen2.5-Coder [151] and Qwen2.5 [152] are the most important models.

The Qwen2.5-Coder is a series of LLMs developed by Alibaba's Qwen Team which is available in six sizes of 0.5B, 1.5B, 3B, 7B, 14B, and 32B parameters. These models are trained on 5.5 trillion tokens, with a focus on different code related tasks such as code generation, completion, debugging, and reasoning. They employ file-level and repository-level pretraining, and advanced strategies for fine-tuning. The Qwen2.5-Coder-32B model achieves the best results across HumanEval, MBPP and so on, outperforming open-source models like DeepSeek-Coder, and even competes with strong models like GPT-4o. The series is open-source and designed to facilitate real-world software development and automated programming [151].

Qwen2.5 is also developed by the same developer as Qwen2.5-coder, including mixture-of-Experts (MoE) models in sizes of 0.5B, 1.5B, 3B, 7B, 14B, 32B, and 72B parameters. It significantly improves over previous versions with pre-training on 18 trillion tokens, expanded instruction fine-tuning, and reinforcement learning for enhanced customized personalization. The series supports long-context processing (up to 1M tokens in Qwen2.5-Turbo) and optimized for reasoning, coding, mathematics, and multilingual tasks. Qwen2.5 outperforms major open-source models like Llama-3-405B while maintaining a strong cost-performance trade-off. [152].

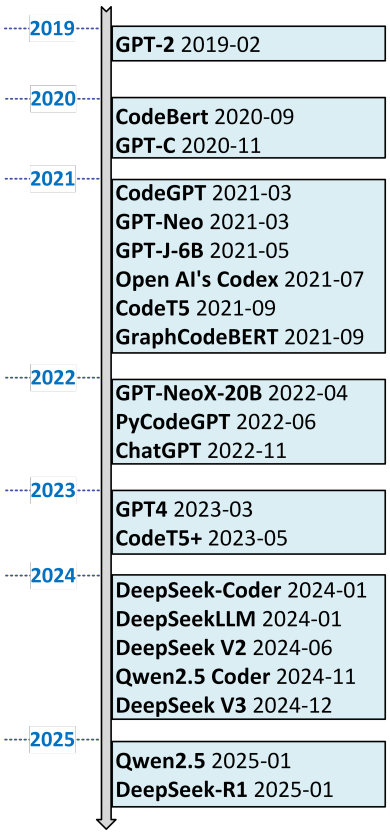

Figure 6: LLM model's timeline for source code analysis

6 Datasets used for LLM code analysis

Using public datasets for LLM training and source code analysis is essential for advancing the capabilities of large language models in programming-related tasks. Here, we provide a list of datasets that can be utilized for various LLM tasks, such as code summarization and and code analysis.

Table 9 shows a list of free datasets that we can we use for various LLM tasks for source code analysis. For example. CodeXGLUE is a widely-used benchmark that includes datasets for tasks like code summarization, translation, and completion [140]. It supports multiple programming languages, including Python, Java, and JavaScript. The data is often provided in formats like JSON or CSV, with fields for code, documentation, and input-output pairs tailored for fine-tuning.

Figure 7 show a time-line of free datasets for code analysis, we can mention famous datasets such as CodesearchNet, CodexGLU, TheStack, and CodeNet, which are key in code search and programming language research. CodesearchNet enables code retrieval through natural language queries, covering multiple programming languages. CodexGLUE focuses on improving code generation and completion via natural language processing. TheStack offers a large collection of code snippets across various languages, supporting cross-lingual code search.

Another notable dataset is CodeNet[140], is a large-scale collection containing around 14 million code samples, each representing a solution to one of nearly 4,000 coding problems. The dataset includes code written in over 50 programming languages, with a particular focus on C++, C, Python, and Java. This dataset, developed by IBM, provides a vast collection of labeled code for program understanding and classification tasks. These datasets are crucial for developing LLM models for code analysis and intelligent code search. In figure 7 the initial release date for each data set is shown:

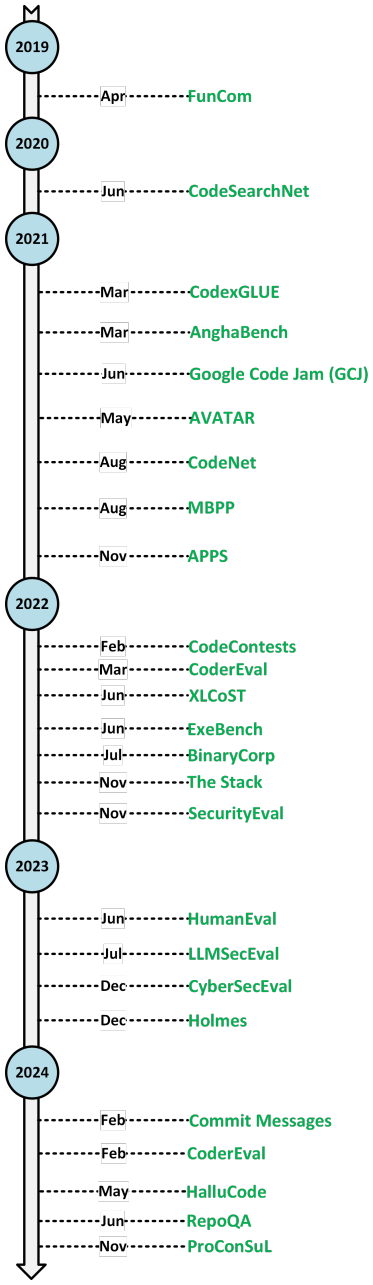

Figure 7: Time Line of LLM Datasets.

Table 9: Publicly available datasets for source code analysis tasks.

Datasets	Details/Langauages	Tasks	Link
CodeSearchNet	2M (comment, code) pairs Ruby	Fine-tunning/ Pre-training	https://github.com/github/CodeSearchNet
CommitMessages	CommitMessages	Pre-training	https://github.com/src-d/datasets/tree/master/CommitMessages
CodeNet	CodeNeta total of 13,916,868 submissions, divided into 4053 problems / C/C++	Fine-tunning	https://github.com/IBM/Project_CodeNet
XLCoST Dataset	for code generation, code translation (code-to-code), code summarization (code-to-text), and code synthesis (text-to-code).	Fine-tunning	https://github.com/reddy-lab-code-research/XLCoST
The stack	6TB of code, 358 languages.		https://huggingface.co/datasets/bigcode/the-stack
AVATAR	A collection of 9515 programming problems and their solutions written (Java and Python)	Function translation	https://github.com/wasiahmad/AVATAR
MBPP	1,000 crowd-sourced Python problems	Pre-training	https://huggingface.co/datasets/google-research-datasets/mbpp
ProConSuL	GitHub repositories written in C/C++	SFT dataset/ Code summarization	https: //github.com/trinity4ai/ProConSuL.
CodeXGLUE	A benchmark for program understanding 14 datasets for different tasks: code-code, text-code, code-text, and text-text)	General	https://github.com/microsoft/CodeXGLUE

Table 10: Prediction of words for [MASK] Tokens by pre-training on BERT models (A snapshot taken from [133])

1) Pathlib is a python library used for handling [MASK].	paths	applications systems programs software data	applications software programs languages systems	paths files urls URLs pathnames	paths path directories filenames strings
2) I have to discuss this with the other [MASK].	developers	elders girls men officers council	men members elders officers people	guys people developers person users	developers team maintainers people devs

7 Discussion and Future work

Regarding the application of LLMs for source code analysis presents several challenges and opportunities for future research. Based on our studies, there are key areas that demand further exploration, which hold significant potential for advancing this field. In this context, we identify seven critical issues and observe that the following gaps remain insufficiently addressed. These gaps highlight promising directions for future work in source code analysis using LLMs.

7.1 Limitations of Publicly Available Datasets for LLM Tasks

Although there are existing datasets for source code such as CodeSearchNet, The Stack or Commit Messages , there remain significant limitations when it comes to finding specialized datasets tailored for specific tasks related to fine-tuning LLMs for source code analysis. Tasks such as code summarization, code disassembly, code decompiling, and comment generation require datasets that focus on the nuances of these specific activities.

While general-purpose code datasets can provide a foundation for training LLMs on basic code generation tasks, they often do not address the complexities and requirements of these advanced tasks [18, 153]. Another important aspect to consider is the quality of source code. For instance, in [154], the authors encountered challenges with the quality of the assembly code search dataset, as it depended on docstrings from sources like CodeSearchNet. In CodeSearchNet's evaluation, 32.8% of docstrings were irrelevant to the source code. Furthermore, comment generation involves aligning natural language with specific code functionalities, a task that is not always explicitly covered by standard code datasets. Our survey reveals a notable gap in the availability of specialized datasets for fine-tuning LLMs on specific tasks, including code summarization, disassembly, decompilation, and comment generation. While existing code datasets can support basic code generation tasks, they fail to address the complexities involved in these advanced code analysis tasks. This limitation underscores the need for the creation of more targeted datasets that focus on these specific areas, enabling LLMs to be fine-tuned for tasks that require deeper understanding and nuanced interactions with source code. Developing such datasets would be a critical step toward improving the capabilities of LLMs in real-world software analysis and development environments.

7.2 Long Code analysis and Token Size

Working with long code in a language model requires careful handling to ensure that the model doesn't exceed its token limit and that context is preserved across different chunks of the code. For example, some models like GPT-4 have a maximum token limit [155–158] , which includes both the prompt and the response. This means that when working with large codebases, only a portion of the code may be processed at a time, potentially leading to truncation or loss of important context. As a result, it's crucial to find ways to manage large code inputs effectively.

For example in [159] , the authors encountered challenges and limitations due to the token size constraints of transformer-based models, impacting the generation of CodeBERT embeddings. To fit within the 512-token limit, we had to truncate the tokens, potentially leading to the loss of some syntactic information from the code snippets.

To address this challenge, it would be helpful to consider methods for chunking the code into smaller sections while maintaining logical consistency [160]. For instance, breaking down code into functions, classes, or modules and analyzing them incrementally can help reduce the token count per request while allowing the model to process the code in smaller, more manageable parts. However, careful attention must be given to ensure that context is not lost between chunks, which could result in incorrect analysis or suggestions[161, 162]. This is particularly critical for codebases that are interdependent, where variables or functions defined earlier may be referenced later.

Moreover, hybrid approaches combining language models with other techniques can be effective for handling large-scale code analysis. Non-LLM methods [163], such as static code analyzers, or even specialized machine learning models trained for code tasks, could be used in collaboration with LLMs to perform complementary tasks like detecting syntax errors, identifying security vulnerabilities, or suggesting refactoring improvements. These methods can help alleviate some of the limitations of LLMs, particularly when it comes to scalability and accuracy in analyzing long code.

Additionally, newer models or extensions, such as those designed to handle longer token sequences like Longformer [164] or Reformer, could be explored to overcome token limitations inherent in traditional transformer-based models [165]. These models are designed to process longer input sequences more efficiently by using techniques like sparse attention mechanisms, which allow them to handle larger contexts without overwhelming the model's capacity.

7.2.1 Long code analysis and Prompt chaining. Prompt chaining is a technique used to break down complex tasks into smaller, sequential prompts, enabling efficient processing of large-scale data while overcoming token limitations. A related study worth mentioning is LLM4FL [166], where the authors tackled token limitations and complexity in fault localization. Their approach utilizes a divide-and-conquer strategy, incorporating prompt chaining and multiple LLM agents to enhance efficiency. It splits large-scale coverage data into manageable chunks, allowing each LLM agent to analyze a subset of the code independently. By integrating Spectrum-Based Fault Localization (SBFL) rankings, LLM4FL refines fault detection across multiple iterations, cross-referencing results between agents to improve accuracy. This structured approach ensures efficient fault localization in complex systems while overcoming performance degradation with long inputs.

7.3 Most Used Models: DeepSeek vs. GPT-4 Family

In our review, most of the work focused on two main families of models: Deepseek models, such as [11, 24, 68, 72, 84, 150], and

GPT models, such as [2, 23, 26, 69, 75]. In table 11 we make a comprehensive comparison between these two models.

Table 11: DeepSeek-R1 vs GPT-4: strong (S), super strong (SS), super super strong (SSS), software engineering (SE), reinforcement learning from human feedback (RLHF)

Criteria	GPT-4	DeepSeek-R1
Input Type	text, multimedia	text
Developer	OpenAI	DeepSeek-AI
Training Approach	Transformation-based Pre-Training + RLHF	RL + Cold Start
Training Data	Public + Licensed	Public + Qwen + Llama
Architecture	Transformer	Transformer
Fine Tuning	RLHF	RL + Distillation
Reasoning Performance	S	SS
Mathematical Ability	S	SSS
Code Generation	S	SSS
General Knowledge	S	S
Context Length	Limited	Longer
Safety Mechanism	Anti Adversary	Avoids Bias
Is Open Source?	No	Yes
Special Ability	General Purpose, Supporting Multi-media	High Performance for Reasoning
limitations	Not Fully Reliable Biased Answers	Excelling only in English and Chinese, Prompt Sensitivity, Limited Capability in SE Tasks
Release Date	2023-03	2025-01
Cost	High	Low
Resource Consumption	High	Low

Regarding the table 11, while both models use transformer architectures and use public datasets, DeepSeek-R1 integrates additional data from Qwen and Llama, along with distillation techniques for fine-tuning. Performance-wise, DeepSeek-R1 surpasses GPT-4 in reasoning, mathematical ability, and code generation, demonstrating "super super strong" (SSS) capabilities in these areas. Additionally, DeepSeek-R1 features a longer context length and an open-source framework, whereas GPT-4 remains proprietary and employs anti-adversarial safety mechanisms.

However, in summary, both DeepSeek and GPT models are powerful language models for source code analysis tasks. The choice between these models can depend on the specific targets and tasks, as each may offer distinct advantages depending on the nature of the problem being addressed [147, 167, 168]. For example, DeepSeek is an open-source model that excels in tasks requiring detailed semantic understanding and structured code analysis. On the other hand, GPT models [169], known for their ability to generate human-like text, are highly versatile and can handle a wide range of tasks, from code completion to generating documentation and explanations for complex code segments.

7.4 Quality of LLMs for code summarization task

Recent findings have evaluated the effectiveness of LLMs for code summarization tasks [2, 4, 86, 117, 170–172]. While certain aspects of code summarization are gaining attention in the era of LLMs, but they are still affected by some challenges. One of the major limitations of LLMs for code summarization is their inability to validate code correctness or test its runtime behavior [120, 145]. While they can generate summaries based on code syntax, they cannot run the code to check for issues like bugs or edge cases[173]. In some works in [174], the authors studied LLM-based code summarization, compared automated evaluation methods (including GPT-4) with human assessments, and found that GPT-4 had the strongest correlation. They also tested five prompting techniques for Java, Python, and C, discovering that simpler zero-shot prompting could outperform more advanced methods depending on the LLM and language.

8 Conclusion

NLP and LLMs are transforming code analysis in computer science by understanding both the structure and intent behind code. In this study, we explored the applications of LLMs for the code analysis, focusing on their ability to assist in tasks such as code summarization, code generation, comment generation, and disassembling, and decompiling code. We also investigated datasets and famous LLM models for the source code analysis. We believe this research opens up new opportunities for using LLMs in software development, offering valuable insights for both developers and researchers in the field. However, based on our understanding of the surveyed topic, we summarize the major lessons we learned as follows:

- High-quality datasets and model selection are crucial for effective source code analysis. The accuracy and performance of LLMs depend significantly on the training data used, as models learn to recognize patterns and generate meaningful outputs based on the data they have been exposed to. Datasets such as CodeXGLUE, CodeNet, and GitHub repositories provide diverse code examples that help train LLMs to understand different programming languages, styles, and structures. However, dataset quality varies, and biases present in training data can impact model reliability. Additionally, selecting the right LLM model for a specific task is essential. OpenAI Codex, Code Llama, and AlphaCode each offer different strengths, and their performance varies depending on the complexity of the code analysis required. Choosing the most suitable model and training it on high-quality, well-labeled datasets can significantly enhance the effectiveness of AI-driven code analysis.
- Interdisciplinary research is key to advancing LLM-based code analysis. This field lies at the intersection of natural

language processing, machine learning, and software engineering, and progress in any of these areas directly impacts the performance of LLMs in code analysis. Collaboration between researchers from different domains can lead to innovative approaches, such as integrating formal methods with machine learning to enhance the accuracy of automated code verification. Additionally, combining techniques from job scheduling, optimized data storage, and efficient code representation can help improve the scalability and effectiveness of LLM-driven tools. By fostering interdisciplinary collaboration, researchers can develop more advanced AI-powered systems that understand and process source code more effectively.

- Customization and fine-tuning of LLMs enhance their performance for specific programming domains. While general-purpose LLMs demonstrate impressive capabilities, they may not always provide optimal results for specialized fields such as embedded systems, cybersecurity, or financial software. Training or fine-tuning models on domain-specific datasets can significantly improve their accuracy and relevance for targeted applications. For instance, an LLM trained on financial transaction processing code may better understand compliance requirements and security constraints compared to a general-purpose model. Customization allows developers to tailor AI-driven tools to their unique needs, leading to more precise and context-aware code analysis.

We are standing at the forefront of LLM-powered source code analysis, and its integration into software engineering presents both exciting opportunities and significant challenges. As LLMs continue to evolve, their ability to automate and enhance various aspects of coding will improve, but further research is needed to optimize their efficiency, accuracy, and security. We hope this study serves as a foundation for future research, inspiring further exploration into the capabilities and limitations of LLMs in source code analysis.

References

[1] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50, 2025.

[2] Chia-Yi Su and Collin McMillan. Distilled gpt for source code summarization. *Automated Software Engineering*, 31(1):22, 2024.

[3] Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601*, 2023.

[4] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959*, 2024.

[5] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[6] Xi Ding, Rui Peng, Xiangping Chen, Yuan Huang, Jing Bian, and Zibin Zheng. Do code summarization models process too much information? function signature may be all that is needed. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–35, 2024.

[7] Marco Russodivito, Angelica Spina, Simone Scalabrino, and Rocco Oliveto. Black-box reconstruction attacks on llms: A preliminary study in code summarization. In *International Conference on the Quality of Information and Communications Technology*, pages 391–398. Springer, 2024.

[8] Eitan Farchi, Shmulik Froimovich, Rami Katan, and Orna Raz. Automatic generation of benchmarks and reliable llm judgment for code tasks. *arXiv preprint arXiv:2410.21071*, 2024.

[9] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. Llm is like a box of chocolates: the non-determinism of chatgpt in code generation. *arXiv preprint arXiv:2308.02828*, 2023.

[10] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. Bias assessment and mitigation in llm-based code generation. *arXiv preprint arXiv:2309.14345*, 2023.

[11] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[12] Feng Lin, Dong Jae Kim, et al. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*, 2024.

[13] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Qingshuang Bao, Weipeng Jiang, Chao Shen, and Yang Liu. Unveiling provider bias in large language models for code generation. *arXiv preprint arXiv:2501.07849*, 2025.

[14] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*, 2024.

[15] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[16] Sathvik Joel, Jie JW Wu, and Fatemeh H Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*, 2024.

[17] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. A survey on language models for code. *arXiv preprint arXiv:2311.07989*, 2023.

[18] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*, 2023.

[19] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[20] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[21] Yuschuk Oleh. Ollydbg. *https://www.ollydbg.de/. Accessed: 2022-02-15*, 2022.

[22] Seokwoo Choi, Taejoo Chang, and Yongsu Park. Unsafengine64: A safengine unpacker for 64-bit windows environments and detailed analysis results on safengine 2.4. 0. *Sensors*, 24(3):840, 2024.

[23] Aleksandr Romanov, Anna Kurtukova, Anastasia Fedotova, and Alexander Shelupanov. Authorship identification of binary and disassembled codes using nlp methods. *Information*, 14(7):361, 2023.

[24] Xinyu She, Yanjie Zhao, and Haoyu Wang. Wadec: Decompiling webassembly using large language model. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 481–492, 2024.

[25] N. S. Agency. Ghidra. *https://ghidra-sre.org/, 2019. [Online; accessed 28-October-2020]*, 2020.

[26] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID*, volume 267622140, 2024.

[27] Capstone: The ultimate disassembly. *[Online]. Available: http://www.capstone-engine.org/*, 2022.

[28] Vector35. Binary ninja. *https://binary.ninja*, 2022.

[29] Essa Imhmed, Edgar Ceh-Varela, and Scott Kilgore. Identifying code quality issues for undergraduate students using static analysis and nlp. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1527–1533. IEEE, 2023.

[30] Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. Deep nlp-based co-evolvement for synthesizing code analysis from natural language. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 141–152, 2021.

[31] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. Graphcode2vec: Generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 524–536, 2022.

[32] Anthony J Rose, Christine M Schubert Kabban, Scott R Graham, Wayne C Henry, and Christopher M Rondeau. Malware classification through abstract syntax trees and l-moments. *Computers & Security*, 148:104082, 2025.

[33] Artyom V Gorchakov, Liliya A Demidova, and Peter N Sovietov. Analysis of program representations based on abstract syntax trees and higher-order markov chains for source code classification task. *Future Internet*, 15(9):314, 2023.

[34] Egor Spirin, Egor Bogomolov, Vladimir Kovalenko, and Timofey Bryksin. Psiminer: A tool for mining rich abstract syntax trees from code. In *2021*

IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pages 13–17. IEEE, 2021.

[35] NA Gribkov, TD Ovasapyan, and DA Moskvin. Analysis of decompiled program code using abstract syntax trees. *Automatic Control and Computer Sciences*, 57(8):958–967, 2023.

[36] Leon Wehmeier, Sebastian Eilermann, Oliver Niggemann, and Andreas Deuter. Task-fidelity assessment for programming tasks using semantic code analysis. In *2023 IEEE Frontiers in Education Conference (FIE)*, pages 1–5. IEEE, 2023.

[37] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *Acm computing surveys (csur)*, 54(3):1–38, 2021.

[38] Shamsa Abid, Xuemeng Cai, and Lingxiao Jiang. Measuring model alignment for code clone detection using causal interpretation. *Empirical Software Engineering*, 30(2):46, 2025.

[39] Mona Nashaat, Reem Amin, Ahmad Hosny Eid, and Rabab F Abdel-Kader. An enhanced transformer-based framework for interpretable code clone detection. *Journal of Systems and Software*, page 112347, 2025.

[40] Jorge Martinez-Gil. Source code clone detection using unsupervised similarity measures. In *International Conference on Software Quality*, pages 21–37. Springer, 2024.

[41] M Maruf Öztürk. A cosine similarity-based labeling technique for vulnerability type detection using source codes. *Computers & Security*, 146:104059, 2024.

[42] Jens Krinke and Chaiyong Ragkhitwetsagul. Code similarity in clone detection. *Code Clone Analysis: Research, Tools, and Practices*, pages 135–150, 2021.

[43] Hadi Ghaemi, Zakieh Alizadehsani, Amin Shahraki, and Juan M Corchado. Transformers in source code generation: A comprehensive survey. *Journal of Systems Architecture*, page 103193, 2024.

[44] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

[45] Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, 2024.

[46] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[47] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[48] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[49] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[50] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[51] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[52] Suborno Deb Bappon, Saikat Mondal, and Banani Roy. Autogenics: Automated generation of context-aware inline comments for code snippets on programming q&a sites using llm. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 24–35. IEEE, 2024.

[53] Omer Nahum, Nitay Calderon, Orgad Keller, Idan Szpektor, and Roi Reichart. Are llms better than reported? detecting label errors and mitigating their effect on model performance. *arXiv preprint arXiv:2410.18889*, 2024.

[54] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30(2):1–28, 2025.

[55] Jahnavi Kumar, Venkata Lakshmana Sasaank Janapati, Mokshith Reddy Tanguturi, and Sridhar Chimalakonda. I can't share code, but i need translation–an empirical study on code translation through federated llm. *arXiv preprint arXiv:2501.05724*, 2025.

[56] Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2448–2449, 2024.

[57] Xuan Li, Shuai Yuan, Xiaodong Gu, Yuting Chen, and Beijun Shen. Few-shot code translation via task-adapted prompt learning. *Journal of Systems and Software*, 212:112002, 2024.

[58] Betim Sherifi, Khaled Slhoub, and Fitzroy Nembhard. The potential of llms in automating software testing: From generation to reporting. *arXiv preprint arXiv:2501.00217*, 2024.

[59] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. Towards autonomous testing agents via conversational large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1688–1693. IEEE, 2023.

[60] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pages 65–73, 2024.

[61] Jonan Richards and Mairieli Wessel. What you need is what you get: Theory of mind for an llm-based code understanding assistant. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 666–671. IEEE, 2024.

[62] Kensen Shi, Deniz Altınbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, et al. Natural language outlines for code: Literate programming in the llm era. *arXiv preprint arXiv:2408.04820*, 2024.

[63] Zhao Tian, Junjie Chen, and Xiangyu Zhang. Test-case-driven programming understanding in large language models for better code generation. *arXiv preprint arXiv:2309.16120*, 2023.

[64] Theodor Vadoce, James Pritchard, and Callum Fairbanks. Enhancing javascript source code understanding with graph-aligned large language models. 2024.

[65] William Macke and Michael Doyle. Testing the effect of code documentation on large language model code understanding. *arXiv preprint arXiv:2404.03114*, 2024.

[66] Dung Nguyen Manh, Thang Phan Chau, Nam Le Hai, Thong T Doan, Nam V Nguyen, Quang Pham, and Nghi DQ Bui. Codemmlu: A multi-task benchmark for assessing code understanding capabilities of codellms. *arXiv preprint arXiv:2410.01999*, 2024.

[67] Ziyu Li and Donghwan Shin. Mutation-based consistency testing for evaluating the code understanding capability of llms. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pages 150–159, 2024.

[68] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[69] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. How far have we gone in stripped binary code understanding using large language models. *arXiv preprint arXiv:2404.09836*, 2024.

[70] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.

[71] Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. Codejudge-eval: Can large language models be good judges in code understanding? *arXiv preprint arXiv:2408.10718*, 2024.

[72] Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025*, 2024.

[73] Huanyao Rong, Yue Duan, Hang Zhang, XiaoFeng Wang, Hongbo Chen, Shengchen Duan, and Shen Wang. Disassembling obfuscated executables with llm. *arXiv preprint arXiv:2407.08924*, 2024.

[74] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[75] Liqiao Xia, Jiazhen Pang, Chengxi Li, Ruoxin Wang, and Pai Zheng. Large language models empower the reliability of disassembly in remanufacturing. *Manufacturing Letters*, 41:1728–1733, 2024.

[76] Juraj Petrik and Daniela Chuda. The effect of time drift in source code authorship attribution: Time drifting in source code - stylochronometry. In *Proceedings of the 22nd International Conference on Computer Systems and Technologies*, CompSysTech '21, page 87–92, New York, NY, USA, 2021. Association for Computing Machinery.

[77] Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*, 2023.

[78] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Nova $^+$: Generative language models for binaries. *arXiv preprint arXiv:2311.13721*, 2023.

[79] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code*

Generation and Optimization (CGO), pages 378–390, 2021.

[80] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2022.

[81] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*, 2024.

[82] Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael FP O'Boyle. Exebench: an ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 50–59, 2022.

[83] Yunlong Feng, Dechuan Teng, Yang Xu, Honglin Mu, Xiao Xu, Libo Qin, Qingfu Zhu, and Wanxiang Che. Self-constructed context decompilation with fined-grained alignment enhancement. *arXiv preprint arXiv:2406.17233*, 2024.

[84] Zian Su, Xiangzhe Xu, Ziyang Huang, Kaiyuan Zhang, and Xiangyu Zhang. Source code foundation models are transferable binary analysis knowledge bases. *arXiv preprint arXiv:2405.19581*, 2024.

[85] Saman Pordanesh and Benjamin Tan. Exploring the efficacy of large language models (gpt-4) in binary reverse engineering. *arXiv preprint arXiv:2406.06637*, 2024.

[86] Kelly Lam. *Code Summarization and Program Synthesis with Large Language Models*. PhD thesis, Massachusetts Institute of Technology, 2024.

[87] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[88] Xinglu Pan, Chenxiao Liu, Yanzhen Zou, Xianlin Zhao, and Bing Xie. Context-focused prompt tuning pre-trained code models to improve code summarization. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1344–1349. IEEE, 2024.

[89] Chia-Yi Su and Collin McMillan. Semantic similarity loss for neural source code summarization. *Journal of Software: Evolution and Process*, 36(11):e2706, 2024.

[90] Chunrong Fang, Weisong Sun, Yuchen Chen, Xiao Chen, Zhao Wei, Quanjun Zhang, Yudu You, Bin Luo, Yang Liu, and Zhenyu Chen. Esale: Enhancing code-summary alignment learning for source code summarization. *IEEE Transactions on Software Engineering*, 2024.

[91] Yonatan Bitton, Gabriel Stanovsky, Michael Elhadad, and Roy Schwartz. Data efficient masked language modeling for vision and language. *arXiv preprint arXiv:2109.02040*, 2021.

[92] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE):2332–2354, 2024.

[93] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024.

[94] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025.

[95] Noble Saji Mathews and Meiyappan Nagappan. Test-driven development and llm-based code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1583–1594, 2024.

[96] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395*, 2024.

[97] Doriane Olewicki, Leuson Da Silva, Suhaib Mujahid, Arezou Amini, Benjamin Mah, Marco Castelluccio, Sarra Habchi, Foutse Khomh, and Bram Adams. Impact of llm-based review comment generation in practice: A mixed open-/closed-source user study. *arXiv preprint arXiv:2411.07091*, 2024.

[98] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[99] Junjie Zhao, Xiang Chen, Guang Yang, and Yiheng Shen. Automatic smart contract comment generation via large language models and in-context learning. *Information and Software Technology*, 168:107405, 2024.

[100] Jianhang Xiang, Zhipeng Gao, Lingfeng Bao, Xing Hu, Jiayuan Chen, and Xin Xia. Automating comment generation for smart contract from bytecode. *ACM Transactions on Software Engineering and Methodology*, 2024.

[101] Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. Leveraging large language model to assist detecting rust code comment inconsistency. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 356–366, 2024.

[102] Kaiwei Cai, Junsheng Zhou, Li Kong, Dandan Liang, and Xianzhuo Li. Automated comment generation based on the large language model. In *International Conference on Computer Science and Education*, pages 283–294. Springer, 2023.

[103] Hanzhen Lu and Zhongxin Liu. Improving retrieval-augmented code comment generation by retrieving for generation. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 350–362. IEEE, 2024.

[104] Md Asif Haider, Ayesha Binte Mostofa, Sk Sabit Bin Mosaddek, Anindya Iqbal, and Toufique Ahmed. Prompting and fine-tuning large language models for automated code review comment generation. *arXiv preprint arXiv:2411.10129*, 2024.

[105] Stefan Goetz and Andreas Schaad. " you still have to study"–on the security of llm generated code. *arXiv preprint arXiv:2408.07106*, 2024.

[106] Balder Janryd and Tim Johansson. Preventing health data from leaking in a machine learning system: Implementing code analysis with llm and model privacy evaluation testing, 2024.

[107] Shigang Liu, Bushra Sabir, Seung Ick Jang, Yuval Kansal, Yansong Gao, Kristen Moore, Alsharif Abuadbba, and Surya Nepal. From solitary directives to interactive encouragement! llm secure code generation by natural language prompting. *arXiv preprint arXiv:2410.14321*, 2024.

[108] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. *arXiv preprint arXiv:2501.08200*, 2025.

[109] Wen Cheng, Ke Sun, Xinyu Zhang, and Wei Wang. Security attacks on llm-based code completion tools. *arXiv preprint arXiv:2408.11006*, 2024.

[110] Mohammed Latif Siddiq, Joanna Santos, Sajith Devareddy, and Anna Muller. Sallm: Security assessment of generated code. *arXiv preprint arXiv:2311.00889*, 2023.

[111] Arya Kavian, Mohammad Mehdi Pourhashem Kallehbasti, Sajjad Kazemi, Ehsan Firouzi, and Mohammad Ghafari. Llm security guard for code. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 600–603, 2024.

[112] Catherine Tony, Markus Mutas, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 588–592, 2023.

[113] Mohammed Latif Siddiq and Joanna C. S. Santos. Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, MSR4P&S 2022, page 29–33, New York, NY, USA, 2022. Association for Computing Machinery.

[114] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. *arXiv preprint arXiv:2312.12575*, 2023.

[115] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, Sasha Frolov, Ravi Prakash Giri, Dhaval Kapil, Yiannis Kozyrakis, David LeBlanc, James Milazzo, Aleksandar Straumann, Gabriel Synnaeve, Varun Vontimitta, Spencer Whitman, and Joshua Saxe. Purple llama cyberseceval: A secure coding benchmark for language models, 2023.

[116] Jiliang Li, Yifan Zhang, Zachary Karas, Collin McMillan, Kevin Leach, and Yu Huang. Do machines and humans focus on similar code? exploring explainability of large language models in code summarization. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 47–51, 2024.

[117] Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. ProConSuL: Project context for code summarization with LLMs. In Franck Dernoncourt, Daniel Preoţiuc-Pietro, and Anastasia Shimorina, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 866–880, Miami, Florida, US, November 2024. Association for Computational Linguistics.

[118] Xin Jin and Zhiqiang Lin. Simllm: Calculating semantic similarity in code summaries using a large language model-based approach. *Proceedings of the ACM on Software Engineering*, 1(FSE):1376–1399, 2024.

[119] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[120] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *arXiv preprint arXiv:2404.10100*, 2024.

[121] Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. Learning-based widget matching for migrating gui test cases. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–13. ACM, February 2024.

[122] Heiko Koziolek and Anne Koziolek. Llm-based control code generation using image recognition. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 38–45, 2024.

[123] Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. Deceptprompt: Exploiting llm-driven code generation via adversarial natural language instructions. *arXiv preprint arXiv:2312.04730*, 2023.

[124] Flavia Monti, Francesco Leotta, Juergen Mangler, Massimo Mecella, and Stefanie Rinderle-Ma. Nl2processops: towards llm-guided code generation for process execution. In *International Conference on Business Process Management*, pages 127–143. Springer, 2024.

[125] Boyuan Chen, Mingzhi Zhu, Brendan Dolan-Gavitt, Muhammad Shafique, and Siddharth Garg. Model cascading for code: Reducing inference costs with model cascading for llm based code generation. *arXiv preprint arXiv:2405.15842*, 2024.

[126] William Zhang, Maria Leon, Ryan Xu, Adrian Cardenas, Amelia Wissink, Hanna Martin, Maya Srikanth, Kaya Dorogi, Christian Valadez, Pedro Perez, et al. Benchmarking llm code generation for audio programming with visual dataflow languages. *arXiv preprint arXiv:2409.00856*, 2024.

[127] Tal Kadosh, Niranjan Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Mihai Capota, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, et al. Domain-specific code language models: Unraveling the potential for hpc codes and tasks. *arXiv preprint arXiv:2312.13322*, 2023.

[128] Shangqing Liu, Yanzhou Li, Xiaofei Xie, Wei Ma, Guozhu Meng, and Yang Liu. Automated commit intelligence by pre-training. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–30, 2024.

[129] Fumiya Uchiyama, Takeshi Kojima, Andrew Gambardella, Qi Cao, Yusuke Iwasawa, and Yutaka Matsuo. Which programming language and what features at pre-training stage affect downstream logical inference performance? *arXiv preprint arXiv:2410.06735*, 2024.

[130] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[131] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR, 2021.

[132] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 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*, pages 18–30, 2022.

[133] Julian Von der Mosel, Alexander Trautsch, and Steffen Herbold. On the validity of pre-trained transformers for natural language processing in the software engineering domain. *IEEE Transactions on Software Engineering*, 49(4):1487–1507, 2022.

[134] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Codereviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095*, 2022.

[135] Xiangyu Zhang, Yu Zhou, Guang Yang, Harald C Gall, and Taolue Chen. Anchor attention, small cache: Code generation with large language models. *arXiv preprint arXiv:2411.06680*, 2024.

[136] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[137] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[138] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[139] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 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*, pages 1433–1443, 2020.

[140] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[141] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58(2), 2021.

[142] Ben Wang and Aran Komatsuzaki. Gpt-j-6b: A 6 billion parameter autoregressive language model. http s. *github. com/kingoflolz/mesh-trans former-jax*, 2021.

[143] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[144] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. Cert: continual pre-training on sketches for library-oriented code generation. *arXiv preprint arXiv:2206.06888*, 2022.

[145] TB OpenAI. Chatgpt: Optimizing language models for dialogue. openai, 2022.

[146] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[147] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[148] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*, 2024.

[149] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[150] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[151] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024.

[152] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[153] Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, et al. Epicoder: Encompassing diversity and complexity in code generation. *arXiv preprint arXiv:2501.04694*, 2025.

[154] Zeyu Gao, Hao Wang, Yuanda Wang, and Chao Zhang. Virtual compiler is all you need for assembly code search. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3040–3051, 2024.

[155] Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848*, 2024.

[156] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[157] Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276*, 2023.

[158] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864, 2024.

[159] Samiha Shimmi, Ashiqur Rahman, Mohan Gadde, Hamed Okhravi, and Mona Rahimi. {VulSim}: Leveraging similarity of {Multi-Dimensional} neighbor embeddings for vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1777–1794, 2024.

[160] Jonathan Cordeiro, Shayan Noei, and Ying Zou. An empirical study on the code refactoring capability of large language models. *arXiv preprint arXiv:2411.02320*, 2024.

[161] Mohamed Lashuel, Aaron Green, John S Erickson, Oshani Seneviratne, and Kristin P Bennett. Llm-based code generation for querying temporal tabular financial data. In *2024 IEEE Symposium on Computational Intelligence for Financial Engineering and Economics (CIFEr)*, pages 1–8. IEEE, 2024.

[162] Josu Diaz-de Arcaya, Juan López-de Armentia, Gorka Zárate, and Ana I Torre-Bastida. Towards the self-healing of infrastructure as code projects using constrained llm technologies. In *Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair*, pages 22–25, 2024.

[163] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Enhancing fault localization through ordered code analysis with llm agents and self-reflection. *arXiv preprint arXiv:2409.13642*, 2024.

[164] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[165] Xueqi Yang, Mariusz Jakubowski, Li Kang, Haojie Yu, and Tim Menzies. Sparsecoder: Advancing source code analysis with sparse attention and learned token pruning. *Empirical Software Engineering*, 30(1):1–30, 2025.

[166] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Enhancing fault localization through ordered code analysis with llm agents and self-reflection. *arXiv preprint arXiv:2409.13642*, 2024.

[167] Xingyuan Bai, Shaobin Huang, Chi Wei, and Rui Wang. Collaboration between intelligent agents and large language models: A novel approach for enhancing code generation capability. *Expert Systems with Applications*, page 126357, 2025.

[168] Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Wanxiang Che, Zhiyuan Liu, and Maosong Sun. Chartcoder: Advancing multimodal large language model for chart-to-code generation. *arXiv preprint arXiv:2501.06598*, 2025.

[169] Fatima Barakat AlShannaq, Mohammad M Shehab, Anwar H Al-Assaf, Esra'a Alhenawi, and Shatha Awawdeh. An exploration into the mechanisms and evolution of gpt models. In *Impacts of Generative AI on the Future of Research and Education*, pages 477–498. IGI Global, 2025.

[170] Kang Wang, Guohua Shen, Zhiqiu Huang, and Xinbo Zhang. Exploring chatgpt's code summarization capabilities: an empirical study. In *International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2024)*, volume 13403, pages 685–693. SPIE, 2024.

[171] Jahnavi Kumar and Sridhar Chimalakonda. Code summarization without direct access to code-towards exploring federated llms for software engineering. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 100–109, 2024.

[172] Giriprasad Sridhara, Sujoy Roychowdhury, Sumit Soman, HG Ranjani, and Ricardo Britto. Icing on the cake: Automatic code summarization at ericsson. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 689–700. IEEE, 2024.

[173] Xiang Chen, Chaoyang Gao, Chunyang Chen, Guangbei Zhang, and Yong Liu. An empirical study on challenges for llm application developers. *ACM Transactions on Software Engineering and Methodology*, 2025.

[174] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959*, 2024.