Reasoning and Planning with Large Language Models in Code Development (Survey for KDD 2024 Tutorial)

Hao Ding*
AWS AI Labs
Santa Clara, CA, USA
haodin@amazon.com

Ziwei Fan
AWS AI Labs
Santa Clara, CA, USA
zwfan@amazon.com

Ingo Gühring
AWS AI Labs
Berlin, Germany
ingogue@amazon.de

Gaurav Gupta
AWS AI Labs
Santa Clara, CA, USA
gauravaz@amazon.com

Wooseok Ha
AWS AI Labs
Santa Clara, CA, USA
wooseoha@amazon.com

Jun Huan
AWS AI Labs
Santa Clara, CA, USA
lukehuan@amazon.com

Linbo Liu
AWS AI Labs
Santa Clara, CA, USA
linbol@amazon.com

Behrooz Omidvar-Tehrani
AWS AI Labs
Santa Clara, CA, USA
omidvart@amazon.com

Shiqi Wang
AWS AI Labs
New York, NY, USA
wshiqi@amazon.com

Hao Zhou
AWS AI Labs
New York, CA, USA
zhuha@amazon.com

ABSTRACT

Large Language Models (LLMs) are revolutionizing the field of code development by leveraging their deep understanding of code patterns, syntax, and semantics to assist developers in various tasks, from code generation and testing to code understanding and documentation. In this survey, accompanying our proposed lecture-style tutorial for KDD 2024, we explore the multifaceted impact of LLMs on the code development, delving into techniques for generating a high-quality code, creating comprehensive test cases, automatically generating documentation, and engaging in an interactive code reasoning. Throughout the survey, we highlight some crucial components surrounding LLMs, including pre-training, fine-tuning, prompt engineering, iterative refinement, agent planning, and hallucination mitigation. We put forward that such ingredients are essential to harness the full potential of these powerful AI models in revolutionizing software engineering and paving the way for a more efficient, effective, and innovative future in code development.

KEYWORDS

Large Language Model, Code Development, Code Generation, Code Migration, Application Modernization

*The list of authors is arranged alphabetically.

KDD '24, 25 August 2024 – 29 August 2024, Barcelona, Spain

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Hao Ding, Ziwei Fan, Ingo Gühring, Gaurav Gupta, Wooseok Ha, Jun Huan, Linbo Liu, Behrooz Omidvar-Tehrani, Shiqi Wang, and Hao Zhou. 2024. Reasoning and Planning with Large Language Models in Code Development (Survey for KDD 2024 Tutorial). In *Proceedings of KDD 2024 (KDD '24).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Large Language Models (LLMs) have emerged as a transformative force in the field of code development, ushering in a paradigm shift that promises to revolutionize the way software is created, tested, and maintained [26, 114]. These powerful AI models, trained on vast amounts of code and natural language data [19, 59], have the potential to dramatically accelerate the development process, improve code quality, and enhance the overall developers experience. The impact of LLMs on code development is far-reaching and multifaceted. By leveraging their deep understanding of code patterns, syntax, and semantics, LLMs can assist developers in a wide range of tasks, from code generation and completion to error detection and bug fixing. They can also facilitate the creation of the more robust and comprehensive test suites, ensuring that a software is thoroughly validated before deployment. Furthermore, LLMs can aid in code understanding and documentation, making it easier for developers to navigate and maintain complex codebases.

It is to note that several surveys related to LLMs for code exist in the community, but their focus is different from our current work. For example, [146] is concerned with a deep dive into the code evaluation. The works of [36, 50, 103] are more focused on the software engineering (SE) side of the problem, while the authors in [138, 139] are concerned with the natural language (NL) aspect of the code generation. A unification of SE and NL is provided in the survey

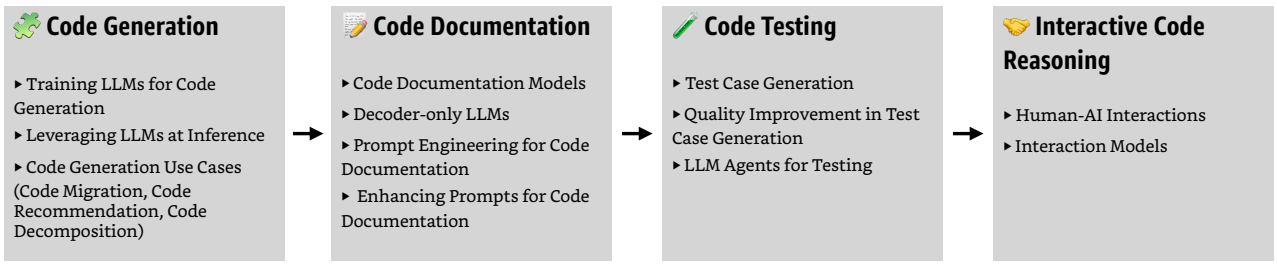

Figure 1: Survey structure according to the sequence of components in a language model-augmented software development lifecycle.

work of [145]. Our work, in contrast, is providing a more holistic view of code reasoning, planning, generation, and refinement. We further note that while survey [54] addresses reasoning in LLMs, it does not pertain to code, which is the focus of our work.

We propose a lecture-style tutorial for KDD 2024, where we explore various ways in which the LLMs are being applied to code development and the significant impact they are having on the field. This survey accompanies the tutorial and provides a thorough examination of the subject matter. The sections of our survey are organized according to the sequence of components in an *LLM-augmented software development lifecycle*, i.e., code generation, code dodumentation, code testing, and interactive code reasoning (Figure 1). We will discuss the following the topics:

Code Generation (Section 2): We begin by examining the use of LLMs in code generation, where we delve into different techniques to generate high-quality, syntactically correct, and semantically meaningful code.

Code Documentation and Summarization (Section 3): Next, we investigate the role of LLMs in code understanding and documentation. We first discuss summarization models and then specific techniques developed through prompt design.

Code Testing (Section 4): We then move on to the application of LLMs in test generation, discussing how they can be leveraged to create comprehensive test cases that cover a wide range of scenarios and edge cases.

Interactive Code Reasoning (Section 5): Finally, we delve into the concept of interactive code reasoning, where LLMs are employed to engage in dialogue with developers, answering questions, providing explanations, and offering suggestions for code improvements.

Throughout this survey, we highlight the various aspects of LLM ecosystem that are crucial to their success in code development, including pre-training, fine-tuning, prompt engineering, iterative refinement, agent planning, and hallucination mitigation. By understanding these components and their interplay, we can harness the full potential of LLMs to revolutionize the way we develop software and pave the way for a more efficient, effective, and innovative future in code development.

2 CODE GENERATION

In this section, we explore the field of code generation with LLMs by focusing on three key aspects. First, we delve into the training process of LLMs specifically tailored for code generation (Section 2.1). Next, we examine different techniques to leverage LLMs in generating codes, such as prompt engineering, iterative refinement, and agent planning (Section 2.2). Finally, we showcase few real-world use cases where code generation with LLMs has been successfully applied (i.e., code migration, code recommendation, and code decomposition), demonstrating its potential to revolutionize the software development practices across different domains and programming languages (Section 2.3).

2.1 Training LLMs for Code Generation

In this section, we briefly describe the latest status for large code generation models and illustrate the existing SOTA approaches. So far, code generation models have emerged as a powerful AI application and widely used in multiple popular products to help boost developers coding efficiency including Microsoft Copilot[1], Amazon Q Developer[2], OpenAI ChatGPT[3], Anthropic Claude[4], and Google Gemini[5].

Popular code models in literature. In public literature, various code generation models have been proposed and achieved impressive performance. Codex [25] is one of the milestones for code models. It followed GPT-3 architecture and achieved reasonable code capability, outperforming most other fine-tuned models based on previous GPT architectures like GPT-NEO [18] and GPT-J [119]. Then many popular models emerge, including CodeGen [85, 86] and AlphaCode [73]. Besides straightforward left-to-right auto-regression, InCoder [44] allows code infilling with a bidirectional context. Recently, StarCoder [71] achieves SOTA performance through BigCode Community trained on the Stack training dataset [62]. It inspires many follow-up works including WizardCoder [77], SantaCoder [12], and MagiCoder [130]. On the other hand, as Llama family are open-released, its fine-tuned variance CodeLlama [100] plays an important role for pushing the performance of public models.

[1] *https://copilot.microsoft.com*
[2] *https://aws.amazon.com/q/developer/*
[3] *https://chat.openai.com*
[4] *https://www.anthropic.com/claude*
[5] *https://gemini.google.com/app*

Training for Code Generation Models. A common approach to developing high-performing code generation models is to fine-tune general foundation models with code data. Notable examples include Codex [25] and CodeLlama [100]. Alternatively, models like StarCoder [71] and CodeGen [85, 86] are pretrained from scratch on code, achieving superior code completion performance but having a more limited understanding of natural language. Generally, training data for these models is sourced from open-source repositories such as GitHub and publicly available platforms like StackOverflow, LeetCode, and coding contests. The Stack [62], by the BigCode Community, is a notable open-source dataset. Recent advancements have been made by supervised fine-tuning on data synthesized or filtered by large foundation models like GPT-4, enhancing code quality, variance, and volume [47, 72, 77, 130]. Mistral 7B [56] showcases advanced performance with an improved model architecture featuring grouped-query attention [11] and sliding window attention [17, 29]. Furthermore, Mixtral 8x7B [57] advances performance with its Sparse Mixture of Experts (SMoE) approach.

Code model generation efficiency. Model inference efficiency is crucial for the practical deployment of large code models. In [43, 63, 129], the authors extensively investigated various static and dynamic model quantization strategies for large code models, proposing optimal settings with empirical support. Speculative decoding is another approach to enhance model performance in code generation, where most tokens generated by expensive large models are replaced with those from smaller, more cost-effective models [69].

Enhancing code generation. In addition to directly prompting the LLMs for single code solution, recent research works have proposed to prompt the LLMs to generate multiple solutions and then filtering them with test cases [22, 73, 105]. AlphaCode [73] generates a very large number of possible solutions, and then cluster-and-filter the samples to obtain a small set of candidate submissions (at most 10), to be evaluated on the hidden test cases. Later, some works have been proposed to boost the generation quality through iterative self revisions. Self-Edit [141] executes the generated code on the example test case provided in the question and wrap execution results into a supplementary comment. Utilizing this comment as guidance, the fault-aware code editor is employed to correct errors in the generated code. Chain-of-Thought (CoT) Prompting [128] guides a language model to produce a sequence of concise sentences that mirror the cognitive steps that a human might take when addressing a problem. It initially found success in solving the math problems, and is later introduced to code generation in [52, 53, 65]. CodeChain [65] is a novel inference framework to improve the code generation in LLMs through a chain of sub-module based self-revisions. It first introduces chain-of-thought prompting to instruct LLMs to decompose their solutions into modular segments. Each modular segment represents an abstract function that is intended for a high-level logical sub-task. The sub-modules are then grouped into clusters and the centroid modules are sub-sampled and augmented to CoT prompt for generating new modularized solutions. AlphaCodium [98] proposes a two-phase workflow to improve the code generation accuracy. The pre-processing phase represents a linear flow where the LLM reasons about the problem, in natural language. Next, the code iterations phase includes iterative stages where the LLM generates, runs, and fixes a solution code against given and artificially-generated tests. Comparing with direct prompt, AlphaCodium shows better performance with different LLMs including DeepSeek-33B, GPT-3.5, and GPT-4.

Evaluation benchmark for code models. The development of evaluation benchmarks for synthetic problems has been pivotal in the recent advancements of large code models. In [25], the authors introduced the executable evaluation benchmark, HumanEval, along with execution metric pass rates. Around the same time, the MBPP (Mostly Basic Programming Problems dataset) and MathQA were also released [15]. Both HumanEval and MBPP have become widely adopted evaluation benchmarks in the community. Additionally, numerous other datasets have been proposed to assess various aspects of code. Works like [48, 73] derive from code contests, targeting more challenging problems. Other notable datasets include ODEX [126] and JuICe [8] for open-domain code problems, MBXP [14] and MultiPL-E [20] for multilingual evaluation, EvalPlus [75] for more rigorous test suites, ReCode [122] for code robustness evaluation, and CrossCodeEval [34] for cross-file code evaluation. Moreover, SWE-Bench [58] evaluates real GitHub issues in an interactive manner, among other specific domain code evaluation benchmarks [55, 64]. As of 9 May 2024, Amazon Q Developer Agent has the highest score on the SWE-Bench leaderboard with 13.82% of resolved cases[6]. It is encouraged to evaluate any code models with comprehensive evaluation benchmarks for model transparency and fair comparisons, following general LLM guidelines [74].

2.2 Leveraging LLMs for Code Generation

In this section, we discuss several techniques that can be leveraged at the inference stage for a high-quality code generation; including prompt engineering, iterative refinement, agent planning, and hallucination mitigation.

Prompt engineering. The goal of prompt engineering is to craft input text in a way that guides the model to produce desired outputs or behaviors. Prompt engineering can be categorized into the following two approaches: (i) intrusive approaches, where LLMs are considered as semi-black box models whose gradients and parameters are partially available, and (ii) non-intrusive approaches, where LLMs are considered as black box models that are only available for inference.

Intrusive approaches for prompt engineering. Intrusive approaches assume the (partial) access to the parameters and gradients of the LLMs. The family of pre-trained language models are considered as promptless fine-tuning intrusive approach, such as BERT [33] and RoBERTa [76]. Another line of work leverages Reinforcement Learning strategies for prompt optimization. For instance, RLPROMPT [32] finds the optimal and efficient prompt using reinforcement learning approach. It develops a policy network that is efficient in its parameter usage, generating an optimized discrete prompt through training with rewards. However, intrusive approaches become extremely expensive especially for the modern LLMs with billions of parameters.

[6] *https://www.swebench.com/*

Non-intrusive approaches for prompt engineering. Non-intrusive approaches to prompt engineering are more cost-efficient than intrusive methods, which require gradient computation. These approaches modify prompts at the input text level rather than the token level, avoiding the granularity and computational expense associated with token-level modifications. Instead, the entire input text is adjusted before being tokenized with a 1-to-1 mapping for model processing. Prompt augmentation techniques improve prompts by instructing and guiding the model to perform specific downstream tasks. In-context learning [35], for example, presents few-shot learning examples to help LLMs adapt to new domains and tasks that differ from their training data. Another effective technique is chain-of-thought prompting [113], which is useful for multi-step reasoning tasks. This method provides the model with a sequence of intermediate steps, leading to a final answer. Additional prompt augmentation techniques include instruction following [90] and program execution [87]. Unlike these static prompt strategies, automatic prompt optimization (APO) [96] employs an iterative approach to refine prompts without modifying model parameters. In each iteration, APO updates the current prompt based on identified flaws, generated by the same or another language model. Furthermore, a prompt pattern catalog presented in [134] enhances prompt engineering for ChatGPT. This catalog includes 16 patterns such as meta-language creation, flipped interaction, persona, question refinement, alternative approaches, and reflection patterns. These patterns significantly enhance the LLM's ability to generate improved responses.

Iterative refinement. For even the most expert human developers, writing code is rarely a one-shot effort. Instead, it typically involves multiple iterations and refinements of an initial draft. This process is often guided by feedback from various stages of the software development life cycle, including static code analysis tools, failing tests, and peer reviews. A similar iterative refinement process can be applied to enhance code generation with large language models (LLMs). This iterative refinement can be categorized into two main approaches: (i) decoupling, and (ii) in-context learning.

Iterative refinement with decoupling. In [67, 133, 142], a base model is proposed to generate an initial draft, which is then refined by a decoupled refinement model over multiple iterations based on feedback from tests. Decoupling allows for the training of a much smaller model that specializes in specific correction tasks. However, a significant challenge is the creation of datasets for training the corrector, as supervised data on how to improve a first draft is typically not available. To address this, [133] propose a training data generation algorithm that pairs generations of different quality, using one to improve upon the other, thereby creating training data for the corrector. Additionally, [21] employ human feedback on failed generations to fine-tune a repair model.

Iterative refinement with in-context learning. In contrast to approaches that use separate models for initial drafts and refinement, some works utilize the same model for both stages, relying solely on *in-context learning*. Feedback is either provided through external validation, such as running tests [28, 28, 98, 106], or by prompting the LLMs to reflect on the generated code [28, 66, 80]. SELF-DEBUG [28] leverages LLMs to debug their own predicted code without human assistance. The approach prompts the model to execute the generated code, explain it line-by-line, compare the explanation to the problem description, and produce feedback to fix any errors. This method is inspired by the "rubber duck debugging" technique used by human programmers. Additionally, the authors explore using unit test results as feedback. SELF-DEBUG is tested across various code generation tasks, such as text-to-SQL and text-to-Python generation. In [80], the authors rely purely on LLM-generated feedback and test their methodology in a broader setup, encompassing natural language, mathematical reasoning, and coding tasks. Similarly, in [66], the focus is on generating modularized programs and iteratively revising them based on previous iterations. In [28, 98, 106], feedback from both public and AI-generated tests is fed back to the model. However, a critical analysis in [89] reveals that "self-repair is not a silver bullet." The authors find that self-repair is more effective with a higher sampling budget and emphasize that its success strongly depends on the model's ability to interpret the provided feedback.

Agent planning and workflow optimization. Numerous frameworks for LLM agents have been proposed in the literature sharing a fundamental similarity: they all establish various agents and coordinate tasks among them through memory-based planning by employing prompt engineering techniques, particularly the Chain-of-Thought prompting [128]. The primary distinction among these frameworks lies in their capability to enable distinct features for the agents.

General-purpose LLM agents. Prominent frameworks for general-purpose use of LLM agents include LangChain Agents [6], Amazon Bedrock Agents [3], and Microsoft AutoGen [135]. LangChain necessitates a degree of programming proficiency, whereas AutoGen and Bedrock Agents simplify the agent creation process, with Bedrock further providing a no-code option. OpenAgents [136] introduces a visual Integrated Development Environment (IDE) for agent configuration. In addition to facilitating code-free development, Bedrock Agents incorporate security measures compliant with general data protection regulation (GDPR). While there are additional general-purpose agents available, many alternatives complicate the development process and increase resource demands, particularly when leveraging GPT-4 technologies [24, 104, 144]. Several applications are built using the aforementioned frameworks over a variety of use cases such as simulating human behavior [27, 70, 93, 94, 109], simulating different personas [125] and formulating expertise [137], collaborative software development [49], improving negotiation skills [45], and incorporating multi-modality and speech [107]. Other frameworks, such as Eco Assistant [140] and KwaiAgents [91] utilize LLM agents to enhance the performance of LLMs. Eco Assistant focuses on improving the cost-efficiency of question answering, while KwaiAgents aims to optimize information retrieval tasks.

LLM Agents for code development. MetaGPT [49] introduces a meta-programming framework for LLM-based multi-agent systems, integrating human-like standard operating procedures (SOPs) to streamline workflows and enhance collaboration among agents. This approach significantly reduces the errors and improves the quality

of code generation by breaking down a complex task into subtasks managed by the specialized agents. Extensive experiments demonstrate MetaGPT's superior performance on collaborative software engineering benchmarks, achieving state-of-the-art results and showcasing its robustness and efficiency in handling the complex software development tasks. In [37], a novel approach is proposed to enhance the reasoning and factual correctness of LLMs through a multi-agent debate system. By having multiple instances of a language model debate, a question over several rounds, the approach leads to the generation of more accurate and factually correct responses. This method outperforms the traditional single-agent models across various tasks, suggesting a promising avenue for improving LLMs' capabilities in both mathematical reasoning and factual information generation. AgentCoder [53] is also a multi-agent framework consisting of three agents: the programmer agent, the test designer agent, and the test executor agent. The programmer agent focuses on the code generation and refinement based on the test executor agent's feedback. It uses the Chain-of-Thought process including problem understanding and clarification, algorithm and method selection, pseudo-code creation, and code generation. The test designer agent generates test cases, and test executor agent runs the code with the test cases and writes feedback to the programmer. Each agent uses a distinct LLM (or conversation) and can excel in its specific task and can be individually updated or replaced with more sophisticated models.

Some LLM Agent tools called "AI Software Engineers" are on the rise, leveraging LLM agents to be software engineering companions in code development. Devin AI [5] is one example whose goal is to be the world's first fully autonomous AI software engineer. In SWE-Bench, Devin successfully resolves 13.86% of issues, exceeding the previous highest unassisted baseline of 1.96% for Anthropic Claude V2. Even when given the exact files to edit ("assisted"), the best previous model only resolves 4.80% of issues. Additionally, Microsoft has introduced AutoDev [116], a fully automated AI-driven software development framework. When evaluated on the HumanEval dataset, AutoDev obtains promising results with 91.5% and 87.8% of Pass@1 for code generation and test generation respectively.

Hallucination mitigation. Hallucinations in LLMs refer to instances where the model generates plausible but incorrect or nonsensical information. Ensuring high-quality and diverse datasets for LLMs is essential in reducing hallucinations [118]. Techniques such as retrieval-augmented generation (RAG) and supervised fine-tuning (SFT) leverage the external knowledge bases and domain-specific datasets, therefore, highlighting the importance of dataset quality and diversity. For example, in the context of code generation, RAG has evolved beyond text-based retrieval to incorporating code demonstrations directly into the generation prompts [83]. SFT enhances LLM capabilities by collecting validated high-quality datasets which align with human preferences for generated response, which is effective across various tasks including code generation [25, 100]. These approaches are instrumental in enhancing the model accuracy, reliability, and the generation of contextually appropriate outputs. By grounding LLM responses through verifiable information and tailoring models to specific domains, the incidence of hallucinations can be mitigated, underscoring the critical role of data in developing trustworthy generative models.

2.3 Use Cases of LLM-based Code Generation

In the context of code generation, LLMs can be broadly categorized into two main types: (i) *code as an input prompt* and (ii) *natural language (NL) to code output*. The first category includes use cases such as code-to-code (e.g., code translation for version updates within the same language, code migration between different languages, code debugging, and code completion to assist in the development process), code-to-text (e.g., code summarization to extract relevant insights), and code-to-motif (e.g., code clustering for grouping similar functionalities for better disjoint deployment, and code classification for easy maintenance of complex codebases). The second category encompasses text-to-code scenarios, such as generating Python code from a natural language description. This section reviews common use cases from both categories.

Code migration. Code migration is an application of code generation where code is both the input and output of the process. In the rapidly evolving field of software development, code migration stands as a critical challenge, necessitating efficient and reliable methodologies for translating code across language versions and programming languages. Recent advancements in LLMs have ushered in a transformative era for this domain, offering unprecedented capabilities for automating and enhancing the code migration process. Amazon Q Developer Agent for Code transformation [2] is among the first industry solutions which leverages LLM for code migration. In [28], the authors explore the potential of code transformation within the framework of SELF-DEBUG for iteratively debugging the generated code. They apply this framework to TransCoder dataset for translating code from C++ to Python, incorporating unit test results and code explanation to guide the migration. Their results show significant improvements in code transformation outcome compared to baseline methods, showcasing the capabilities of LLMs for automating the code migration process. Moreover, SteloCoder [92] introduces a decoder-only Language Model (LLM) for translating multiple programming languages into Python, based on the StarCoder LLM. SteloCoder utilized advanced techniques including Mixture of Experts (MoE), Low-Rank Adaptive (LoRA) methods, and a curriculum learning approach with self-instructed data. It trained separate expert models for each source language and employed an adaptive MoE gating module to select the appropriate expert for the translation tasks. Self-instructed data, consisting of source language tag tokens paired with program snippets, enriched the learning process. SteloCoder adopted a curriculum training strategy, initially fine-tuning on code snippet-based data before progressing to full program datasets, thus gradually enhancing its language translation capabilities.

Code recommendation. GitHub Copilot [4] and AWS Q Developer [1] are two AI-driven development tools designed to enhance productivity in software development by offering code recommendations. The input can be either a code snippet to complete or a natural language description. GitHub Copilot, developed by GitHub in collaboration with OpenAI, is built on the GPT language model, offering features like code suggestions, autocompletion, and generation across various languages and frameworks, thereby improving coding efficiency for a broad range of projects. AWS Q Developer, developed by Amazon Web Services, is designed to provide code

suggestions, architecture recommendations, performance optimizations, and best security practices specifically tailored for AWS cloud development.

Code decomposition. Code decomposition is a crucial process for modernizing applications by transitioning from monolithic architectures to smaller microservices [7]. First, by generating a detailed and comprehensive documentation, LLMs enable developers to better identify the logical modules and functionalities for transition into microservices. Next, after identifying candidate groups through various approaches such as graph neural networks (GNNs) [84], code documentation generated by LLMs can provide functional summaries for each group. This reduces the required time and effort for manual inspection of each group and aids the developers in assessing their suitability and qualities for microservice architecture.

3 CODE DOCUMENTATION

Code documentation and summarization refers to the process of generating descriptive natural language texts that articulate the functionality and structure of software code. Such high-quality summaries are critical for developers in understanding and maintaining code effectively. Despite its significance, producing high-quality code documentation is challenging and time-consuming due to the complex nature of code development, including the complexity of code applications, the diversity of programming languages, and the nuance of code functionality. These challenges have sparked research interest into automatic code documentation aimed at generating precise and informative summaries that can provide the connection between code and its intended functionality and structure. In particular, with the success of LLMs in generating natural languages for various tasks, there is an increasing focus on investigating the capabilities of LLMs in facilitating the code documentation process.

In this section, we discuss the progression and effectiveness of various approaches in the realm of code summarization and documentation. Initially, we delve into the capabilities of pre-trained models which have been specifically designed to interpret and engage with both programming and natural languages. Furthermore, we examine the role of decoder-only LLMs in the context of code documentation. Through examples we illustrate how customized prompting can substantially improve LLM output, potentially surpassing traditional models. The critical role of prompt engineering is underscored by innovations like ASAP and PromptCS, which have been shown to enhance LLM effectiveness in code documentation tasks. These methodologies either incorporate semantic facts into prompts or utilize a prompt agent that crafts prompts to guide LLMs towards generating more precise code summaries.

3.1 Code Documentation Models

CodeBERT [42] is a pre-trained model designed to work with both natural and programming languages. Based on the BERT architecture [33], it is trained on a large corpus of code from multiple programming languages, making it effective for tasks such as code search, documentation, and completion. CodeBERT excels in natural language code search and documentation generation, enabling developers to query the model in natural language to retrieve relevant code snippets or documentation. CodeT5 [124], a pre-trained encoder-decoder model extending the T5 model [97], treats NLP problems as text-to-text tasks. CodeT5 is designed for programming language tasks, including code documentation, generation, and translation among different languages. It leverages code semantics conveyed by developer-assigned identifiers, demonstrating strong capabilities in code generation and understanding tasks such as defect detection and clone detection. To address the limitations of using a single unified model for multiple downstream tasks, CodeT5+ [123] employs a family of encoder-decoder LLMs. These components can be flexibly combined to suit a wide range of code tasks. CodeT5+ shows substantial performance improvements on many downstream tasks, including code summarization, compared to state-of-the-art baselines.

Decoder-only LLMs for code documentation. Recently, decoder-only LLMs (e.g., OpenAI's ChatGPT, Anthropic's Claude) have become increasingly popular for tasks involving both natural and programming languages, serving as the foundation for various code understanding tools. [112] conducted a preliminary evaluation of ChatGPT's capability in zero-shot code summarization. They designed several heuristic questions to collect feedback from ChatGPT, aiming to identify effective prompts that guide the model to generate code summaries. Using these prompts, they instructed ChatGPT to generate summaries for code snippets and assessed their quality by calculating metrics such as BLEU, METEOR, and ROUGE-L, comparing them against ground-truth summaries. Evaluated on the CSN-Python dataset, their results showed that ChatGPT's performance in code summarization underperformed compared to widely-used models such as NCS [117], CodeBERT, and CodeT5. Similar findings were reported by [111] for the CSN-Java dataset. Conversely, [9] studied the code summarization capabilities of LLMs using few-shot prompts. Their study demonstrated that by employing 10 examples in a few-shot prompt, Codex [26] can surpass the performance of code summarization models, including CodeBERT and CodeT5. Additionally, in GPTutor [23], the authors introduced a ChatGPT-powered programming tool for code explanation as a Visual Studio Code extension. Its specially designed prompts enabled it to outperform vanilla ChatGPT and GitHub Copilot in a preliminary evaluation.

3.2 Prompt Engineering for Code Documentation

As prompt engineering emerges as a critical element in optimizing the efficacy of LLMs, a large body of research is dedicated to improving prompts for code documentation tasks. In [10], the authors propose Automatic Semantic Augmentation of Prompts (ASAP), a new approach aimed at enhancing the construction of prompts through the integration of semantic facts. For the purpose of code summarization, ASAP includes semantic facts such as repository name, function's fully qualifed name, signature, AST tags, and data flow graph. ASAP demonstrates the improved performance of LLMs compared to the conventional few-shot prompting baseline built using vanilla BM25 [99]. In [46], the authors study the capabilities of LLMs in generating comments with multiple intents, i.e., generating multiple comments that summarize various aspects of a

given code. Their findings show the dependency of summarization performance on the adequacy of the model's prompting strategies. For example, when the number of demonstration examples provided for in-context learning is less than 10, the performance of LLM is sub-optimal compared to the state-of-the-art approach for multiple-intent comments generation; whereas the model is provided with a sufficient number of examples, its performance to generate multiple-intent comments significantly improves. Moreover, when the demonstration examples are similar to the target code snippet, the LLM demonstrates enhanced performance in comparison to when examples are selected randomly. In [101], LLMs with static code analysis are leveraged to develop a method capable of abstracting high-level explanations of the source code. Utilizing a chain-of-thought prompting strategy, the approach begins by understanding lower-level software abstractions, such as individual functions, then progressively synthesizes these insights to generate summaries of more complex structures, such as classes or components.

Enhancing prompts for code documentation. Manually crafting prompts for code summarization can be challenging, highlighting the need for a prompt agent capable of generating tailored prompts. In PromptCS [111], the authors introduce a new prompt learning framework designed specifically for code summarization tasks. PromptCS is built with two core components: a prompt agent and an LLM. The prompt agent's role is to craft prompts that effectively guide the LLM in generating accurate code summaries. This is achieved by training the prompt agent on code snippets and their corresponding ground-truth summaries. The training process involves using the LLM and deep learning models to encode the code snippets and pseudo prompts into embeddings, which then serve as inputs. The outputs are the predicted summaries, and the training proceeds by minimizing the discrepancy between these predictions and the ground-truth summaries. Once trained, the prompt agent can produce prompt embeddings for new code snippets, guiding the LLM to generate predicted summaries. The effectiveness of PromptCS has been demonstrated on the CodeXGLUE dataset, a comprehensive collection of code snippets and their associated descriptions across various programming languages. Compared to traditional instruction prompting methods with zero-shot or few-shot learning, PromptCS shows improvements across several metrics, such as BLEU, METEOR, ROUGE-L, and SentenceBERT. Additionally, it achieves comparable, and in some cases superior, performance to task-oriented fine-tuning approaches while significantly reducing training costs.

4 TEST GENERATION

The related work in test generation explores the integration and application of LLMs in enhancing software testing methodologies and software engineering practices [39, 41, 68, 102, 108, 121]. A common theme across these studies is the innovative use of LLMs to either generate test cases, improve test coverage, or serve as the autonomous agents in software testing, showcasing the versatility and potential of LLMs in automating and optimizing the various aspects of software development and testing.

A key difference among the LLM-based test generation works is their focus areas and methodologies. While [68, 102] concentrate on leveraging LLMs for test case generation with specific emphasis on search-based testing and unit test generation, respectively, the work in [108] adopt a Reinforcement Learning approach to refine the quality of LLM-generated test cases. On the other hand, comprehensive reviews and surveys are provided in [121] and [39] on the broader application of LLMs in software testing and engineering, identifying challenges, contributions, and future research directions. Lastly, [41] proposes a conceptual framework for LLM-driven testing agents, highlighting the potential of conversational AI in testing processes.

4.1 Test Case Generation with Language Models

In [68], the authors present an innovative approach to enhance the efficiency of search-based software testing (SBST) through the integration of LLMs, specifically OpenAI's Codex. CodaMosa, the proposed algorithm, addresses the challenge of coverage plateaus in SBST by leveraging Codex to generate example test cases for under-covered functions when traditional SBST stalls, thereby redirecting the search towards more fruitful areas. The evaluation of CodaMosa across 486 benchmarks demonstrated its ability to achieve statistically significant higher coverage in a majority of cases, showcasing the method's effectiveness over SBST and LLM-only baselines. The approach not only increases the test coverage but also introduces a novel way of integrating arbitrary Python test cases into SBST, enhancing the exploration capabilities of SBST. Despite its success, the approach's reliance on the quality and relevance of the LLM-generated test cases could be seen as a limitation, as well as the potential for increased computational overhead due to the integration of LLMs into the testing process. Nonetheless, CodaMosa represents a significant step forward in automated test generation, particularly for handling complex software systems that challenge the traditional testing methods. In [102], the authors presents an empirical evaluation of LLMs like GPT-3 for automated unit test generation in JavaScript without additional training or few-shot learning. The approach provides the LLM with prompts containing function signatures, implementations, and documentation examples. If a test fails, the model is re-prompted to generate a fix. The tool TestPilot implementing this approach for 25 npm packages achieves 70.2% statement coverage and 52.8% branch coverage, significantly outperforming prior feedback-directed techniques. Experiments show all prompt components contribute to effectiveness. Most tests have low similarity to existing tests. Additional experiments with different LLMs suggest the approach works across models but benefits from larger training sets. Overall, the paper demonstrates LLMs can automatically generate high-quality unit tests without extra training, enabling practical applications for test generation. Limitations include dependence on documentation quality and limited evaluation of real fault finding.

4.2 Quality Improvement in Test Case Generation

In [108], the authors delves into the evaluation of LLMs in generating software test cases, particularly focusing on their adherence to best practices and the presence of test smells, such as missing

assert statements or focal method invocations. The authors identify that LLM-generated tests often fail to meet best practice standards and propose a novel solution named Reinforcement Learning from Static Quality Metrics (RLSQM). RLSQM leverages Reinforcement Learning, specifically Proximal Policy Optimization (PPO), to train models using rewards based on static quality metrics, aiming to optimize test case generation towards higher quality and adherence to best practices. The technique is shown to significantly enhance the quality of generated test cases by up to 21%, outperforming GPT-4 on multiple metrics and producing nearly 100% syntactically correct code. The study underscores the potential of RLSQM to improve software testing efficiency and reliability, though it does not extensively discuss the limitations of this approach, such as the potential computational overhead or challenges in defining comprehensive quality metrics that encompass all aspects of best practices in test case generation.

4.3 LLM Agents for Testing

In [41], the authors discuss the integration of LLMs into the software testing process, presenting a novel approach to enhance the test automation through LLM-driven testing agents. It introduces a taxonomy of LLM-based testing agents categorized by their autonomy level, illustrating how varying degrees of autonomy can benefit developers by providing a conversational framework for testing, which can make the testing process more efficient and effective. The paper argues that LLMs, with their extensive training on software test data, can act as a knowledgeable assistant in testing, capable of understanding the developer intentions and suggesting or generating test cases autonomously. It highlights the potential for LLMs to identify the unexpected issues through dialogue, leveraging their "hallucination" capabilities positively. Additionally, the paper outlines the architecture of autonomous systems like AutoGPT, which, equipped with memory management and the ability to interact with external tools, can significantly advance software testing automation. Despite these advantages, the paper also acknowledges limitations, such as the need for specialized middleware to facilitate LLM interaction with testing tools and the challenge of ensuring that LLM-driven testing aligns with the specific project requirements.

5 INTERACTIVE CODE REASONING

Despite recent advancements in LLM-based code development, several fundamental challenges persist. For instance, the Claude V2 LLM reports a limited success rate of 4.8% in addressing GitHub issues within the SWE-Bench benchmark, even when supported by an oracle file retriever [58]. This highlights the complexity of code development and the necessity for a collaborative approach between humans and AI to enhance outcomes [78]. As suggested in [132], the focus should now shift towards creating effective interaction models for human-AI collaboration. These models should aim to leverage the strengths of both humans and AI, enabling them to work together more efficiently.

5.1 Designing Human-AI Interactions

A considerable body of research is dedicated to the development of effective human-AI interactions, scrutinizing the foundational elements of such partnerships. This encompasses tasks such as delineating the responsibilities allocated to AI systems [95], formulating principles for crafting interactions with AI [13, 30, 131, 143], and fostering suitable trust levels [38, 61, 79, 110]. Despite these insights, evidence suggests that practitioners encounter challenges in implementing these broad strategies directly into their fields [31, 82, 115]. However, LLMs demonstrate heightened efficacy when tailored to particular domains, indicating a need for domain-specific interaction designs to enhance the human-AI partnership. Consequently, there is a pressing demand for forthcoming research to articulate experience design frameworks that are finely tuned to the requirements of distinct domains.

5.2 Interaction Models

A focal area of research is the exploration of human-AI collaborations within specific applications. These collaborations are generally classified into three models: (*i*) *co-creation*, where humans and AI collaboratively generate outputs [16, 40, 60, 81, 88], (*ii*) *complementary work* [51, 120], in which humans and AI independently produce akin outputs, and (*iii*) *hand-offs*, involving humans delegating distinct tasks to AI [127, 132].

In [78], the human-AI partnership is examined in the context of code migration, an example of the hand-off model. This study leverages Amazon Q Developer Agent for Code transformation [2] to conduct semi-structured interviews with 11 participants. It discusses the roles of humans as directors and reviewers in the partnership and defines a trust framework based on various model outcomes to earn trust with LLMs. The findings highlight the importance of developer involvement and trust-building in enhancing the productivity and efficiency of AI-assisted code migrations. As directors, humans guide the AI to produce the desired output by bringing their existing knowledge and setting preferences. As reviewers, humans evaluate, edit, and correct the AI-generated code, holding it to the same standards as code written by human teammates, and ensuring it meets quality and security requirements.

6 CONCLUSION

In conclusion, we discuss the transformative impact of Large Language Models (LLMs) on code development. LLMs are revolutionizing code generation, testing, and documentation, by leveraging their deep grasp of code's intricacies. This survey (accompanying our proposed tutorial for KDD 2024) highlights essential techniques such as pre-training, fine-tuning, and prompt engineering, which are vital for maximizing LLMs' effectiveness for code development. This discussion underscores the significance of adopting LLMs in software development practices, pointing towards a future where software creation is more efficient, innovative, and aligned with the evolving demands of technology.

REFERENCES

[1] [n. d.]. Amazon Q Developer. https://aws.amazon.com/q/developer/.
[2] [n. d.]. Amazon Q Developer Agent for Code Transformation. https://aws.amazon.com/q/developer/code-transformation/. Accessed: 2024-06-03.
[3] [n. d.]. Bedrock Agents. https://aws.amazon.com/bedrock/agents/.
[4] [n. d.]. Github Copilot. https://github.com/features/copilot/. Accessed: 2024-03-03.

[5] [n. d.]. Introducing Devin, the first AI software engineer. https://www.cognition.ai/blog/introducing-devin.

[6] [n. d.]. LangChain Agents. https://python.langchain.com/docs/modules/agents/.

[7] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. 2023. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering* (2023).

[8] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv preprint arXiv:1910.02216* (2019).

[9] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[10] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2023. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). *arXiv preprint arXiv:2304.06815* (2023).

[11] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[12] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[13] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction *(CHI '19)*.

[14] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2022. Multi-lingual Evaluation of Code Generation Models. *CoRR* abs/2210.14868 (2022). https://doi.org/10.48550/arXiv.2210.14868 arXiv:2210.14868

[15] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[16] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

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

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

[19] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[20] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. *arXiv preprint arXiv:2208.08227* (2022).

[21] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749* (2023).

[22] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. arXiv:2207.10397 [cs.CL]

[23] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. 2023. GPTutor: a ChatGPT-powered programming tool for code explanation. *arXiv preprint arXiv:2305.01863* (2023).

[24] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. 2023. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288* (2023).

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

[26] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[27] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*.

[28] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[29] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).

[30] Nazli Cila. 2022. Designing Human-Agent Collaborations: Commitment, Responsiveness, and Support *(CHI '22)*.

[31] Lucas Colusso, Cynthia L Bennett, Gary Hsieh, and Sean A Munson. 2017. Translational resources: Reducing the gap between academic research and HCI practice. In *Proceedings of the 2017 conference on designing interactive systems*. 957–968.

[32] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P Xing, and Zhiting Hu. 2022. Rlprompt: Optimizing discrete text prompts with reinforcement learning. *arXiv preprint arXiv:2205.12548* (2022).

[33] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[34] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems* 36 (2024).

[35] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[36] Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. 2023. Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey. arXiv:2308.01191 [cs.SE]

[37] Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving Factuality and Reasoning in Language Models through Multiagent Debate. *arXiv preprint arXiv:2305.14325* (2023).

[38] Upol Ehsan, Q. Vera Liao, Michael Muller, Mark O. Riedl, and Justin D. Weisz. 2021. Expanding Explainability: Towards Social Transparency in AI Systems *(CHI '21)*.

[39] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[40] Judith E Fan, Monica Dinculescu, and David Ha. 2019. Collabdraw: an environment for collaborative sketching with an artificial agent. In *Proceedings of the 2019 on Creativity and Cognition*. 556–561.

[41] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards Autonomous Testing Agents via Conversational Large Language Models. *arXiv preprint arXiv:2306.05152* (2023).

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

[43] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *arXiv preprint arXiv:2210.17323* (2022).

[44] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[45] Yao Fu, Hao Peng, Tushar Khot, and Mirella Lapata. 2023. Improving language model negotiation with self-play and in-context learning from ai feedback. *arXiv preprint arXiv:2305.10142* (2023).

[46] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. (2024).

[47] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks Are All You Need. *arXiv preprint arXiv:2306.11644* (2023).

[48] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).

[49] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).

[50] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE]

[51] Cheng-Zhi Anna Huang, Curtis Hawthorne, Adam Roberts, Monica Dinculescu, James Wexler, Leon Hong, and Jacob Howcroft. 2019. The bach doodle: Approachable music composition with machine learning at scale. *arXiv preprint arXiv:1907.06637* (2019).

[52] Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation. arXiv:2308.08784 [cs.SE]

[53] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2024. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. arXiv:2312.13010 [cs.CL]

[54] Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards Reasoning in Large Language Models: A Survey. arXiv:2212.10403 [cs.CL]

[55] Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. *arXiv preprint arXiv:2211.09374* (2022).

[56] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[57] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[58] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2023).

[59] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1073–1085.

[60] Pegah Karimi, Jeba Rezwana, Safat Siddiqui, Mary Lou Maher, and Nasrin Dehbozorgi. 2020. Creative sketching partner: an analysis of human-AI co-creativity. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*. 221–230.

[61] Sunnie S. Y. Kim, Elizabeth Anne Watkins, Olga Russakovsky, Ruth Fong, and Andrés Monroy-Hernández. 2023. "Help Me Help the AI": Understanding How Explainability Can Support Human-AI Interaction *(CHI '23)*.

[62] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[63] Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. 2022. FP8 Quantization: The Power of the Exponent. *arXiv preprint arXiv:2208.09225* (2022).

[64] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*. PMLR, 18319–18345.

[65] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules. arXiv:2310.08992 [cs.AI]

[66] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=vYhglxSj8j

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

[68] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.

[69] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[70] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2024. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems* 36 (2024).

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

[72] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463* (2023).

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

[74] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110* (2022).

[75] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[76] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[77] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568* (2023).

[78] Ishaani M, Behrooz Omidvar-Tehrani, and Anmol Anubhai. 2024. Evaluating Human-AI Partnership for LLM-based Code Migration. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems - Late Breaking Work*.

[79] Shuai Ma, Ying Lei, Xinru Wang, Chengbo Zheng, Chuhan Shi, Ming Yin, and Xiaojuan Ma. 2023. Who Should I Trust: AI or Myself? Leveraging Human and AI Correctness Likelihood to Promote Appropriate Trust in AI-Assisted Decision-Making *(CHI '23)*.

[80] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2024).

[81] Andrew M McNutt, Chenglong Wang, Robert A Deline, and Steven M Drucker. 2023. On the design of ai-powered code assistants for notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.

[82] Jane N Mosier and Sidney L Smith. 1986. Application of guidelines for designing user interface software. *Behaviour & information technology* 5, 1 (1986), 39–46.

[83] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[84] Hoa Xuan Nguyen, Shaoshu Zhu, and Mingming Liu. 2022. A Survey on Graph Neural Networks for Microservice-Based Cloud Applications. *Sensors* 22, 23 (2022), 9492.

[85] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv preprint arXiv:2305.02309* (2023).

[86] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[87] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114* (2021).

[88] Changhoon Oh, Jungwoo Song, Jinhan Choi, Seonghyeon Kim, Sungwoo Lee, and Bongwon Suh. 2018. I lead, you help but only with enough details: Understanding user experience of co-creation with artificial intelligence. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–13.

[89] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. *arXiv preprint arXiv:2306.09896* (2023).

[90] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[91] Haojie Pan, Zepeng Zhai, Hao Yuan, Yaojia Lv, Ruiji Fu, Ming Liu, Zhongyuan Wang, and Bing Qin. 2023. Kwaiagents: Generalized information-seeking agent system with large language models. *arXiv preprint arXiv:2312.04889* (2023).

[92] Jialing Pan, Adrien Sadé, Jin Kim, Eric Soriano, Guillem Sole, and Sylvain Flamant. 2023. SteloCoder: a Decoder-Only LLM for Multi-Language to Python Code Translation. *arXiv preprint arXiv:2310.15539* (2023).

[93] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[94] Joon Sung Park, Lindsay Popowski, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2022. Social simulacra: Creating populated prototypes for social computing systems. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–18.

[95] Marc Pinski, Martin Adam, and Alexander Benlian. 2023. AI Knowledge: Improving AI Delegation through Human Enablement. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–17.

[96] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495* (2023).

[97] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[98] Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering. *arXiv preprint arXiv:2401.08500* (2024).

[99] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[100] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[101] Satrio Adi Rukmono, Lina Ochoa, and Michel RV Chaudron. 2023. Achieving High-Level Software Component Summarization via Hierarchical Chain-of-Thought Prompting and Static Code Analysis. In *2023 IEEE International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 7–12.

[102] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[103] Xinyu She, Yue Liu, Yanjie Zhao, Yiling He, Li Li, Chakkrit Tantithamthavorn, Zhan Qin, and Haoyu Wang. 2023. Pitfalls in Language Models for Code Intelligence: A Taxonomy and Survey. arXiv:2310.17903 [cs.SE]

[104] Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. 2024. Small llms are weak tool learners: A multi-llm agent. *arXiv preprint arXiv:2401.07324* (2024).

[105] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural Language to Code Translation with Execution. arXiv:2204.11454 [cs.CL]

[106] Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366* (2023).

[107] Yu Shu, Siwei Dong, Guangyao Chen, Wenhao Huang, Ruihua Zhang, Daochen Shi, Qiqi Xiang, and Yemin Shi. 2023. Llasm: Large language and speech model. *arXiv preprint arXiv:2308.15930* (2023).

[108] Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation. *arXiv preprint arXiv:2310.02368* (2023).

[109] Theodore R Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. 2023. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427* (2023).

[110] Lingyun Sun, Zhuoshu Li, Yuyang Zhang, Yanzhen Liu, Shanghua Lou, and Zhibin Zhou. 2021. Capturing the Trends, Applications, Issues, and Potential Strategies of Designing Transparent AI Agents *(CHI EA '21)*.

[111] Weisong Sun, Chunrong Fang, Yudu You, Yuchen Chen, Yi Liu, Chong Wang, Jian Zhang, Quanjun Zhang, Hanwei Qian, Wei Zhao, et al. 2023. A Prompt Learning Framework for Source Code Summarization. *arXiv preprint arXiv:2312.16066* (2023).

[112] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *arXiv preprint arXiv:2305.12865* (2023).

[113] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261* (2022).

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

[115] Linda Tetzlaff and David R Schwartz. 1991. The use of guidelines in interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 329–333.

[116] Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2024. AutoDev: Automated AI-Driven Development. *arXiv preprint arXiv:2403.08299* (2024).

[117] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. *arXiv e-prints* (2020), arXiv–2005.

[118] Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, Jason Wei, Chris Tar, Yun-Hsuan Sung, Denny Zhou, Quoc Le, and Thang Luong. 2023. FreshLLMs: Refreshing Large Language Models with Search Engine Augmentation. arXiv:2310.03214 [cs.CL]

[119] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 billion parameter autoregressive language model.

[120] Fengjie Wang, Xuye Liu, Oujing Liu, Ali Neshati, Tengfei Ma, Min Zhu, and Jian Zhao. 2023. Slide4N: Creating Presentation Slides from Computational Notebooks with Human-AI Collaboration *(CHI '23)*.

[121] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[122] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022. ReCode: Robustness Evaluation of Code Generation Models. *arXiv preprint arXiv:2212.10264* (2022).

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

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

[125] Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2023. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *arXiv preprint arXiv:2307.05300* (2023).

[126] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481* (2022).

[127] Thomas Weber, Heinrich Hußmann, Zhiwei Han, Stefan Matthes, and Yuanting Liu. 2020. Draw with me: Human-in-the-loop for image restoration. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*. 243–253.

[128] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[129] Xiaokai Wei, Sujan Gonugondla, Wasi Ahmad, Shiqi Wang, Baishakhi Ray, Haifeng Qian, Xiaopeng Li, Varun Kumar, Zijian Wang, Yuchen Tian, et al. 2023. Greener yet Powerful: Taming Large Code Generation Models with Quantization. *arXiv preprint arXiv:2303.05378* (2023).

[130] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).

[131] Justin D Weisz, Michael Muller, Jessica He, and Stephanie Houde. 2023. Toward general design principles for generative AI applications. *arXiv preprint arXiv:2301.05578* (2023).

[132] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? Human-AI partnerships in code translation. In *26th International Conference on Intelligent User Interfaces*. 402–412.

[133] Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2023. Generating Sequences by Learning to Self-Correct. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hH36JeQZDaO

[134] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[135] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155* (2023).

[136] Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, et al. 2023. Openagents: An

open platform for language agents in the wild. *arXiv preprint arXiv:2310.10634* (2023).

[137] Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. 2023. ExpertPrompting: Instructing Large Language Models to be Distinguished Experts. *arXiv preprint arXiv:2305.14688* (2023).

[138] Yichen Xu and Yanqiao Zhu. 2022. A Survey on Pretrained Language Models for Neural Code Intelligence. arXiv:2212.10079 [cs.SE]

[139] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. arXiv:2212.09420 [cs.SE]

[140] Jieyu Zhang, Ranjay Krishna, Ahmed H Awadallah, and Chi Wang. 2023. EcoAssistant: Using LLM assistant more affordably and accurately. *arXiv preprint arXiv:2310.03046* (2023).

[141] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 769–787. https://doi.org/10.18653/v1/2023.acl-long.45

[142] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. *arXiv preprint arXiv:2305.04087* (2023).

[143] Qiaoning Zhang, Matthew L Lee, and Scott Carter. 2022. You Complete Me: Human-AI Teams and Complementary Expertise *(CHI '22)*.

[144] Wenqi Zhang, Yongliang Shen, Linjuan Wu, Qiuying Peng, Jun Wang, Yueting Zhuang, and Weiming Lu. 2024. Self-Contrast: Better Reflection Through Inconsistent Solving Perspectives. *arXiv preprint arXiv:2401.02009* (2024).

[145] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. arXiv:2311.07989 [cs.CL]

[146] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2024. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. arXiv:2311.10372 [cs.SE]