Anh Nguyen-Duc
Pekka Abrahamsson
Foutse Khomh *Editors*

Generative AI for Effective Software Development

Springer

Generative AI for Effective Software Development

Anh Nguyen-Duc • Pekka Abrahamsson •
Foutse Khomh
Editors

Generative AI for Effective Software Development

Springer

Editors

Anh Nguyen-Duc
Department of Business and IT
University of South-Eastern Norway
Bø I Telemark, Norway

Pekka Abrahamsson
Tampere University
Tampere, Finland

Foutse Khomh
Polytechnique Montréal
Montréal, QC, Canada

ISBN 978-3-031-55641-8 ISBN 978-3-031-55642-5 (eBook)
https://doi.org/10.1007/978-3-031-55642-5

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

1 The Rise of Generative AI in Software Engineering

This era marks a significant advancement and application of generative artificial intelligence (Generative AI). At the time of writing this section (December 2023), the significance of Generative AI is unquestionable with a prevailing consensus that the technology will bring about a transformative effect across all sectors of society and industry. Generative AI is most referred to as a technology that (i) leverages deep learning models to (ii) generate humanlike content (e.g., images, words) in response to (iii) complex and varied prompts (e.g., languages, instructions, questions). A McKinsey report highlights the immense potential of Generative AI, estimating its capability to add a value of \$2.6 to \$4.4 trillion across diverse industries. Its impact on software engineering productivity alone could lead to a 20–45% reduction in current annual spending, primarily by streamlining activities such as drafting initial code, code correction, refactoring, root-cause analysis, and system design generation.

Generative AI tools are increasingly becoming a staple in software development, aiding in both managerial and technical project facets. Prominent models like Meta's LLaMA, OpenAI's ChatGPT, GitHub Copilot, and Amazon CodeWhisperer are reshaping traditional concerns about productivity and quality in technology adoption. These tools, with their advanced code generation capabilities and AI-assisted environments, are transforming the software development process by automating routine tasks, enhancing code, and even writing significant code segments. Beyond coding, generative AI aids in areas like requirements engineering and project management. However, generative AI isn't just about task automation; it signifies a fundamental shift in problem-solving within software development. This paradigm shift transcends process automation, redefining human roles, teamwork dynamics, and decision-making in software development. Software developers are transitioning from code-centric roles to AI collaborators, focusing on high-level architecture, setting AI goals, and interpreting AI-generated solutions. This shift enables developers to concentrate on more complex, creative aspects, fostering an interdisciplinary approach with teams comprising domain experts, data scientists, and ethicists to maximize generative AI's potential.

2 Vision of Generative AI in Future Software Development

The advanced machine learning that powers generative AI-enabled products has been decades in the making. But since ChatGPT came off the starting block in late 2022, new iterations of generative AI technology have been released several times a month. In March 2023 alone, there were six major steps forward, introducing significant improvement for software engineering-related innovation.

The future of software engineering holds exciting promises with the integration of generative AI. As technology continues to advance, AI-driven systems are poised to revolutionize the way software is designed, developed, and maintained. One of the most compelling visions is the concept of collaboration between AI agents and software engineers. In this vision, generative AI will work alongside human developers, assisting them in various aspects of the software development lifecycle. These AI collaborators will have the capability to understand natural language requirements, generate code based on high-level descriptions, and even help in debugging and optimizing code. This collaborative partnership between human developers and AI is expected to significantly accelerate the software development process, reduce errors, and open up new possibilities for innovation.

Another key aspect of the future of software engineering is the potential for AI to automate routine and time-consuming tasks. Generative AI models can generate code snippets, templates, and even entire modules based on patterns and best practices learned from vast repositories of code. This automation will allow developers to focus on more creative and complex aspects of software development, such as requirement elicitation, crafting user-centric interfaces, high-level architectural decisions, and ethical considerations and compliance. In 2022, GitHub reported on the impact of Copilot on developer productivity and happiness. Eighty-eight percent of survey respondents replied that they feel more productive with Copilot. Moreover, 74% say they are faster with repetitive tasks and can focus on more satisfying work. We envision that the positive impact of tools like Copilot will expand, benefiting various types of projects and adapting to diverse organizational environments.

Furthermore, generative AI in software engineering will enable the creation of highly personalized and adaptive software systems. These AI-driven applications will be capable of learning from user interactions and preferences, continuously evolving to meet the changing needs of their users. For instance, in the realm of user interface design, AI can generate user interfaces that are tailored to individual preferences and accessibility requirements. This level of personalization will lead to more engaging and user-friendly software experiences, ultimately enhancing user satisfaction and the overall quality of software products. In essence, the future of software engineering with generative AI holds the promise of increased productivity, improved software quality, and the creation of highly customized and adaptive software solutions.

Another significant impact of generative AI in software engineering will be in the realm of personalized software solutions. AI will enable the creation of highly customized software that can adapt to the specific needs and preferences of individual users or businesses. This will be made possible by AI's ability to learn from user interactions and evolve over time. The software will become more intuitive and user-friendly, as it will be able to anticipate user needs and offer solutions proactively. This level of personalization will not only enhance user experience but also open up new avenues for innovation in software design and functionality.

To provide a complete perspective, it's essential to acknowledge the existing challenges of generative AI technologies, which currently stand as key areas of focus for research and development among software engineering scholars and professionals. In general, large language models (LLMs) are still easy to produce hallucination, misleading, inconsistent, or unverifiable information. Models built on top of historically biased data pose problems of fairness and trustworthiness, and when incidents happen, issues about safety and responsibility can arise. LLMs may fall short of mastering generation tasks that require domain-specific knowledge or generating structured data. It is nontrivial to inject specialized knowledge into LLMs. Techniques like prompting, augmenting, fine-tuning, and the use of smaller AI models present potential solutions, yet applying them to specific problems is nontrivial. For software engineering tasks, the current evaluation of generative AI focuses more on code generation tasks, with less emphasis on evaluating or researching other tasks such as requirement generation, code fixes, and vulnerability repair. It is anticipated that exploring these areas will be a significant and influential line of research for the software engineering community.

Finally, generative AI will also play a crucial role in democratizing software development. With AI-assisted coding, individuals who may not have formal training in programming will be able to create and modify software. This will break down barriers to entry in the field of software development and foster a more inclusive and diverse community of software creators. It will empower a wider range of people to bring their unique ideas to life, leading to a more vibrant and innovative software landscape. This democratization will not only spur creativity but also lead to the development of solutions that cater to a broader spectrum of needs and challenges in various industries.

3 Purpose of the Book

The purpose of this book—Generative AI for Effective Software Development—is to provide a comprehensive, empirically grounded exploration of how generative AI is reshaping the landscape of software development across diverse environments and geographies. This book emphasizes the empirical evaluation of generative AI tools in real-world scenarios, offering insights into their practical efficacy, limitations, and impact on various aspects of software engineering. It focuses on the human aspect, examining how generative AI influences the roles, collaborations, and decision-making processes of developers from different countries and cultures. By presenting case studies, surveys, and interviews from various software development contexts, the book aims to offer a global perspective on the integration of generative AI, highlighting how these advanced tools are adapted to and influence diverse cultural, organizational, and technological environments. This multifaceted approach not only showcases the technological advancements in generative AI but also deeply considers the human element, ensuring that the narrative remains grounded in the practical realities of software developers worldwide. While generative AI technologies encompass a wide range of data types, our cases focus mainly on LLMs with text and code generation. The evaluation is done with current models, such as Llama 2 or ChatGPT-4, acknowledging the current limitations associated with them.

4 Structure and Topics

This book is structured to provide a comprehensive understanding of generative AI and its transformative impact on the field of software engineering. The book is divided into four main parts, each focusing on different aspects of generative AI in software development. Below is a detailed outline of the book's structure and the topics covered in each section.

Part I presents the fundamentals of generative AI adoption. The introductory chapter offers a brief overview of generative AI and its growing relevance in the field of software engineering. It also provides a roadmap of the book's structure and chapters.

Part II is a collection of empirical studies on patterns and tools for the adoption of generative AI in software engineering. This section delves into the practical aspects of integrating generative AI tools in software engineering, with a focus on patterns, methodologies, and comparative analyses. In this part, Dae-Kyoo Kim presents a comparative analysis of ChatGPT and Bard, highlighting their complementary strengths for effective utilization in software development (Chap. 2). Jorge Melegati and Eduardo Guerra introduce DAnTE, a taxonomy designed to categorize the automation degree of software engineering tasks (Chap. 3). Jules White and colleagues discuss ChatGPT prompt patterns for enhancing code quality, refactoring, requirements elicitation, and software design (Chap. 4). Krishna Ronanki and co-authors explore the use of generative AI in requirements engineering, focusing on prompts and prompting patterns (Chap. 5). Also on requirements engineering, Chetan Arora, John Grundy, and Mohamed Abdelrazek assess the role of large language models (LLMs) in advancing requirements engineering (Chap. 6).

Part III presents case studies that showcase the application and impact of generative AI in various software development contexts. Particularly, Arghavan Moradi Dakhel and colleagues provide a family of case studies on generative AI's application in code generation tasks (Chap. 7). Dang Nguyen Ngoc Hai et al. explore the CodeBERT approach for automatic program repair of security vulnerabilities (Chap. 8). Väinö Liukko and colleagues present a case study of ChatGPT as a full-stack Web developer (Chap. 9).

Part IV examines how generative AI is reshaping software engineering processes, from collaboration and workflow to management and agile development. To start with, Rasmus Ulfsnes and co-authors provide empirical insights on how generative AI is transforming collaboration and workflow in software development (Chap. 10). Beatriz Cabrero-Daniel, Yasamin Fazelidehkordi, and Ali Nouri discuss the enhancement of software management with generative AI (Chap. 11). Dron Khanna and Anh Nguyen Duc conduct a survey study on the value-based adoption of ChatGPT in Agile software development among Nordic software experts (Chap. 12). Guilherme Pereira and colleagues share early results from a study of generative AI adoption in a large Brazilian company, focusing on the case of Globo (Chap. 13).

Part V is about future directions and education. The final section of the book looks toward the future, exploring emerging trends, future directions, and the role of education in the context of generative AI. Shunichiro Tomura and Hoa Dam discuss generating explanations for AI-powered delay prediction in software projects (Chap. 14). Mohammad Idris Attal and team present a prompt book for turning a large language model into a start-up digital assistant (Chap. 15). Mika Saari and colleagues explore effective approaches to utilize AI tools in programming courses, guiding students in this new era of AI-assisted software development (Chap. 16).

5 What This Book Is and Isn't

This book is not a technical manual on how to code with generative AI tools. The book is also not about customizing or developing generative AI models but rather their application in software engineering. The book offers a strategic, managerial, and process-centric viewpoint, highlighting how generative AI can be a potentially in different software development activities, irrespective of the programming language, software technologies, or development framework.

While this book presents various empirical applications of generative AI in software development, it is not an exhaustive guide on all aspects of software engineering. It is, however, a crucial read for anyone interested in understanding how generative AI is revolutionizing software development and what it means for the future of this field.

The book offers diverse perspectives as it compiles research and experiences from various countries and software development environments, reflecting a global view of generative AI's impact. The book offers non-technical discussions about generative AI in management, teamwork, business, and education.

Advanced generative AI technologies with powerful capacities can come with severe risks to public safety, be it via misuse or accident; hence, safety-ensured mechanisms, i.e., enforcing safety and security standard for development and deployment of generative AI models, processes, and practices for developing responsible generative AI models, are relevant topics. These topics, however, fall outside the scope of this publication. Instead, our focus remains on the current state and capabilities of generative AI technologies, exploring their existing applications and immediate impacts. For a comprehensive understanding of the broader implications and future potential of these technologies, including safety and ethical considerations, readers may need to consult additional specialized resources.

6 Acknowledgments

This book would not have been possible without the massive collaborative effort of our reviewers, authors, and editors. The insights encapsulated within these pages are a product of the knowledge and experiences shared by many software engineering researchers and practitioners. Although the authors and editors are specifically acknowledged in each chapter or callout, we'd like to take time to recognize those who contributed to each chapter by providing thoughtful input, discussion, and review. We extend our gratitude to Khlood Ahmad, Christian Berger, Beatriz Cabero-Daniel, Ruzanna Chitchyan, John Grundy, Eduardo Guerra, Helena Holstrom Olsson, Zoe Hoy, Ronald Jabangwe, Marius Rohde Johannessen, Dron Khanna, Foutse Khomh, Dae-Kyoo Kim, Johan Linåker, Jorge Melegati, Anh Nguyen Duc, Amin Nikanjam, Dimitris Polychronopoulos, Tho Quan, Usman Rafiq, Viktoria Stray, Ingrid Sunbø, Rasmus Ulfsnes, Hironori Washizaki, and Jules White.

This book represents a collaborative effort that extends beyond the boundaries of any single institution or discipline. We are profoundly grateful to the numerous contributors whose expertise, insights, and unwavering dedication have been instrumental in bringing this project to fruition:

- Norwegian University of Science and Technology, Norway
- University of Oslo, Norway
- University of South Eastern, Norway
- SINTEF, Norway
- Chalmers University of Technology, Sweden
- University of Gothenburg, Sweden
- Volvo Cars, Sweden
- Solita Ltd., Finland
- Tampere University, Finland
- Free University of Bozen-Bolzano, Italy
- University of California Irvine, USA
- Vanderbilt University, USA

- Oakland University, USA
- Polytechnique Montreal, Canada
- Pontificia Universidade Catolica do Rio Grande do Sul, Brazil
- Globo, Brazil
- Deakin University, Australia
- Monash University, Australia
- University of Wollongong, Australia
- Waseda University, Japan
- Ho Chi Minh City University of Technology (HCMUT), VNU-HCM, Vietnam

Hanoi, Vietnam — Anh Nguyen-Duc
Tampere, Finland — Pekka Abrahamsson
Montreal, QC, Canada — Foutse Khomh
December 31, 2023

Contents

Part I
Fundamentals of Generative AI

An Overview on Large Language Models

Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Hironori Washizaki

Abstract Generative artificial intelligence (AI), propelled by the advancements in large language models (LLMs), has exhibited remarkable capabilities in various software engineering (SE) tasks and beyond. This development has influenced the research studies in this domain. This chapter offers an overview of LLMs, delving into relevant background concepts while exploring advanced techniques at the forefront of LLM research. We review various LLM architectures, in addition to discussing the concepts of training, fine-tuning, and in-context learning. We also discussed different adaptation approaches to LLMs and augmented LLMs. Furthermore, we delve into the evaluation of LLM research, introducing benchmark datasets and relevant tools in this context. The chapter concludes by exploring limitations in leveraging LLMs for SE tasks.

Keywords Generative AI · Large language models · Transformers · In-context learning

1 Introduction

Generative artificial intelligence (AI) represents a category of AI systems with the ability to create diverse content forms, including text, audio, image, and code [73]. Generative AI models are designed to grasp the patterns and structures present in their training dataset. After the training step, these models can generate new content that exhibits similar characteristics to their training data.

A. Moradi Dakhel (✉) · A. Nikanjam · F. Khomh · M. C. Desmarais
Polytechnique Montréal, Montréal, QC, Canada
e-mail: arghavan.moradi-dakhel@polymtl.ca; amin.nikanjam@polymtl.ca; foutse.khomh@polymtl.ca; michel.desmarais@polymtl.ca

H. Washizaki
Waseda University, Tokyo, Japan
e-mail: washizaki@waseda.jp

A. Nguyen-Duc et al. (eds.), *Generative AI for Effective Software Development*,
https://doi.org/10.1007/978-3-031-55642-5_1

Large language models (LLMs), a subset of Generative AI, have demonstrated remarkable proficiency in tasks related to language comprehension and generation [64, 118]. Notably, transformer-based models such as GPT by OpenAI [9] and PaLM by Google [13] have showcased exceptional language generation capabilities. These models excel in various language-oriented tasks such as text generation [35], question answering [66], translation [9], summarization [94], and sentiment analysis [47]. Moreover, LLMs, such as OpenAI's Codex [12] and Meta's LLaMA-2 [87], are designed to automatically generate code in various programming languages through training on extensive open-source projects. These LLMs have been used to successfully generate code for multiple code-related tasks, including implementing specific functionalities [102], generating test cases [91], and fixing buggy code [105].

The exceptional performance of LLMs has sparked a growing interest in harnessing their potential within software engineering (SE), which is not limited to code-related tasks [114, 117]. Researchers have explored new opportunities to leverage LLMs in different phases of the SE life cycle, including software requirements and design (e.g., specifications generation [107] and requirements classification [76]), software development (e.g., code completion [15] and code search [53]), software testing (e.g., vulnerability detection [32] and fault localization [14]), and software maintenance (e.g., program repair [10] and code review [54]).

In this book chapter, we provide an overview of LLMs. We review transformer-based LLMs' architecture, popular LLMs, and various approaches for adapting LLMs, including pre-training, fine-tuning, in-context learning, and augmentation. Additionally, we delve into resource-efficient methods for adapting LLMs, discuss datasets and evaluation approaches applied in different studies that used LLMs, and briefly explore relevant tools and limitations in employing LLMs for SE tasks.

Chapter Organization Section 2 offers an overview of LLMs and their architectures. Section 3 briefly reviews different methods for adaptation of LLMs. We explore the capability of in-context learning in Sect. 4. Augmented LLMs are discussed in Sect. 5. Section 6 presents a brief review of datasets and evaluation techniques that have been used in studies leveraging LLMs. LLM-related tools for SE tasks are explored in Sect. 7. We conclude this chapter by discussing the limitation of leveraging LLMs for SE tasks in Sect. 8.

2 Large Language Models

LLMs represent a revolutionary advancement in natural language processing (NLP), demonstrating capabilities for general-purpose language understanding and generation [64]. LLMs acquire these remarkable abilities by leveraging massive datasets to learn billions of parameters during training, necessitating substantial computational resources for both training and operational phases [11].

These models are artificial neural networks, with the *transformer* architecture [92], such as Bidirectional Encoder Representations from Transformers (BERT) [24], which are (pre-)trained using self-supervised learning and semi-supervised learning. The transformer architecture, characterized by self-attention mechanisms [92], serves as the fundamental building block for language modeling tasks.

This approach has demonstrated effectiveness across broad applications, ranging from language translation to code generation. Notable examples of these models include OpenAI's GPT [9] series (including GPT-3.5 and GPT-4, utilized in ChatGPT), Google's PaLM [13] (deployed in Bard), and Meta's LLaMA [87].

2.1 *Tokenization*

Tokenization plays a crucial role in NLP by dividing documents, whether they are text or code, into smaller units known as tokens. A token could stand for a word, subword, character, or symbol, depending on the model's type and size [4]. This process helps models in effectively managing diverse languages and input formats [63]. There are several tokenization approaches, like WordPiece [81], Byte Pair Encoding (BPE) [82], SentencePiece [49], and Unigram [48].

2.2 *Attention Mechanism*

The attention mechanism computes a representation of a sequence of tokens in the input by establishing relationships between different tokens in the sequence [68]. This technique serves as a solution to provide multiple meanings or significance to a token, depending on its context and neighboring tokens [92]. It helps learn long-range dependencies in the input sequence, as it allows the model to focus on specific parts of the sequence when processing input, enhancing the model's ability to capture and understand complex relationships within the context [92].

2.3 *Encoder and Decoder*

The Transformer architecture, which forms the core of LLMs, consists of an encoder and a decoder [92]. This architecture is also known as sequence-to-sequence, allowing it to transform an input sequence of tokens into an output sequence, as seen in translation tasks [70, 115].

Each of these components is comprised of multiple layers of embedding, attention, and feed-forward neural networks [37]. The encoder's primary role is to convert each token in the input sequence into a fixed-length vector, capturing semantic information about each token. Conversely, the decoder is responsible for generating the output sequence of tokens by minimizing the gap between the predicted token and the target token [68, 92]. Two very well-known algorithms used in decoders to generate the sequence of output tokens are greedy search and beam search [42]. Greedy search is a decoding strategy where, at each step, the model selects the token with the highest probability as the next token in the sequence [80]. Greedy search is computationally efficient but may result in sequences that are suboptimal on a global scale, as it does not explore alternative possibilities beyond the current best choice [75]. Beam search is a decoding approach that, instead of selecting only the top-scoring token at each step, maintains a set of the most likely sequences, known as the beam. The beam size determines the number of sequences collected at each step [17, 93].

An Encoder-only architecture, or auto-encoding, focuses on the task of encoding and understanding the input sequence. This architecture is valuable for downstream tasks where contextual representation is crucial, but autoregressive generation in the output is not necessary [109]. For tasks like text classification, name prediction, or sentiment analysis, an encoder-only architecture is useful [33, 108]. BERT (Bidirectional Encoder Representations from Transformers), for example, employs an encoder-only architecture with bidirectional self-attention, allowing it to learn a comprehensive representation of input tokens while considering both the left and right context to learn the embedding vectors [24].

Conversely, a Decoder-only architecture, also known as an auto-regressive model, focuses on creative generation. These models predict the next token in a sequence step by step, generating each token based on the previous tokens [29]. In autoregressive models, the next token is produced using a right-shift method, where the input sequence is shifted to the right by incorporating the generated token into the input sequence [31]. In this approach, each generated token becomes part of the input sequence for predicting the next token, and it allows the model to consider the evolving context in each step, generating tokens based on the input sequence and the tokens generated before. A notable example of a decoder-only model is the Generative Pre-trained Transformer (GPT) [31].

2.4 Activation Functions

Activation functions play an important role in introducing non-linearity into a neural model's decision-making process [68]. These functions are applied to the output of each neuron in a neural network and facilitate the learning of complex patterns and relationships within the data [30]. One widely used activation function in large language models is the Rectified Linear Unit (ReLU), which replaces negative input values with zero while leaving positive values unchanged. Mathematically, this function is expressed as $max(0, x)$ [2]. The choice of an activation function depends on the specific requirements of the model. Different activation functions can yield diverse impacts on the learning process, and their appropriateness may vary based on the architecture and characteristics of the processed data [30].

2.5 Prompt

A prompt is an instruction given to a trained LLM at the inference step that enables the model to generate answers for queries it has never seen before [119]. The output generated by LLM can adapt to the context and instructions provided in the prompt without the need for fine-tuning or alignment. Trained LLMs can be prompted using different setups to generate the best answers [99]. Widely used prompt setups will be explored in Sect. 4.

3 Model Adaption

This section explores various methods for adapting an LLM to a specific downstream task, spanning from pre-training to resource-efficient model adaptation.

3.1 Pre-training

Pre-training in LLMs denotes the initial training phase, encompassing both self-supervised learning, where the model predicts masked words or sequences in unlabeled data, and semi-supervised learning, integrating labeled data to fine-tune the model for specific tasks. The term "pre-training" is employed because it anticipates the need for additional training or post-processing steps to adapt the pre-trained model to the desired task [64]. A widely used pre-training objective for LLMs is Masked Language Modeling (MLM) [16]. In this pre-training technique, the goal is to train the model by predicting tokens that are randomly masked within the input sequence.

3.2 Fine-Tuning

Fine-tuning involves taking pre-trained models and refining them through additional training on smaller, task-specific labeled datasets [64]. This process adapts the models' capabilities and enhances their performance for a particular task or domain. Essentially, fine-tuning transforms general-purpose models into specialized ones. An example of such a task is fine-tuning CodeBERT for defect detection task [69].

3.3 Alignment Tuning

LLMs have the potential to generate outputs that are incorrect, harmful, or biased. Adapting LLMs with human feedback can aid in updating the model parameters to mitigate the occurrence of such outputs [68]. Reinforcement Learning using Human Feedback (RLHF) is a well-known technique employed for alignment tuning in LLMs. In RLHF, a fine-tuned model is further trained with human feedback as a part of the reward system [120]. The RLHF process involves collecting human feedback on the outputs of a fine-tuned model. These feedback responses are then used to learn a reward model that can predict a numerical reward for a generated output. Finally, the model is optimized by incorporating this reward model and leveraging RL techniques [120]. This iterative approach of learning from human feedback contributes to enhancing the model's alignment and adapting the model to avoid generating incorrect, harmful, or biased outputs. Human feedback serves as a valuable source for refining the model's parameters, making it more adept at addressing complex human preferences that may be challenging to capture through traditional reward functions [62].

3.4 Resource-Efficient Model Adaptation

LLMs usually include a large amount of parameters, so conducting the full parameter tuning and deploying them are computationally expensive, in terms of memory and processing resources. Hence, researchers developed various techniques for model adaptation in resource-limited settings, either by parameter-efficient tuning or quantization to reduce the memory footprint of LLMs.

Parameter-efficient techniques for LLMs enable the adaptation of LLMs for downstream tasks at reduced computational costs. Among these techniques, Low-Rank Adaptation (LoRA) [41] receives great attention from the community. LoRA operates by fixing the pre-trained model weights and embedding trainable rank decomposition matrices into the layers of the Transformer architecture. It has found broad adoption in parameter-efficient tuning of open-source LLMs such as LLaMA and BLOOM. Notably, LoRA has been widely applied on LLaMA and its variations. One instance is the development of AlpacaLoRA,[1] which is a LoRA-trained version of Alpaca [86] (a fine-tuned 7B LLaMA model leveraging 52K human demonstrations for instruction following).

Given the substantial memory requirements of LLMs during inference, their deployment in real-world applications becomes costly. Several strategies have been proposed to reduce the memory footprint of LLMs, focusing on a prevalent model compression technique known as model quantization. The aim is to enable the uti-

[1] https://github.com/tloen/alpaca-lora.

lization of large-scale LLMs in resource-constrained environments and decreasing inference latency. Two primary approaches to model quantization are:

– Quantization-Aware Training (QAT), which necessitates complete model retraining, like LLM.int8() [22], ZeroQuant [112], and SmoothQuan [106]
– Post-Training Quantization (PTQ), which does not require retraining, like QLoRA [23] and LLM-qat [56]

4 In-Context Learning (ICL)

LLMs demonstrate an ability for In-Context Learning (ICL); meaning that they can learn effectively from a few examples within a specific context. Studies [3, 98, 111] show that LLMs can perform complex tasks through ICL. The fundamental concept of ICL revolves around the model's capacity to learn the patterns through the examples and subsequently make accurate predictions [27]. One advantage of ICL is the possibility of engaging in a dialogue with the model. Second, ICL closely aligns with the decision-making processes observed in humans by learning from analogy [100]. In contrast to traditional training and tuning approaches, ICL operates as a training-free framework, significantly reducing the computational costs associated with adapting the model to new tasks. Moreover, this approach transforms LLMs into black boxes as a service that can be integrated into real-world tasks [85]. Various ICL techniques have been proposed in the literature. In the following section, we will discuss several well-known techniques.

4.1 Few-Shot Learning

Few-shot learning uses a few labeled examples in the prompt to adapt the model for a specific task. This process involves providing contextual demonstration examples as input/output pairs that represent the downstream task. These demonstrations serve to instruct the model on how to reason or use tools and perform actions [3]. This technique enables the use of the same model for various downstream tasks without requiring tuning or changing the model's parameters [9]. The effectiveness of this technique relies on the relevancy of the few examples to the target task, and the format of these examples guides the model in predicting the output format. For instance, authors in [55] employ few-shot learning to demonstrate their method for generating step-by-step solutions that align with the math problems in their training data. The objective of this study is not to impart new skills to the model with few-shot learning; instead, it aims to guide the model in generating solutions in a desired step-by-step format.

4.2 Chain-of-Thought (CoT)

CoT is motivated by the natural step-by-step thinking ability of humans and has been observed to improve the performance of LLMs in solving problems that require multi-step reasoning [98]. The human thought process for tackling a complex problem, such as a complex math problem, involves breaking down the problem into intermediate tasks and solving them to reach the final answer. In CoT, the primary task is decomposed into intermediate tasks, and the LLM then finds answers for these intermediate tasks to resolve the main problem [98]. Another type of CoT is the self-planning approach, which employs LLMs to break down the original task into smaller steps termed plans [46]. The model is then invoked on these provided steps for execution.

4.3 Reasoning+Action (ReAct)

ReAct is an ICL approach that leverages the capabilities of LLMs to generate both reasoning traces and actions on conducting a task in an interleaved manner, which allows the model to engage in dynamic reasoning, creating, maintaining, and adjusting high-level action plan for action (reason to act) [111]. Additionally, ReAct can interact with external environments, such as Wikipedia, to incorporate additional information into its reasoning process (act to reason) [111]. This dynamic interplay between reasoning and action distinguishes ReAct and enhances its performance in tasks requiring decision-making such as question/answering or fact verification [62]. This technique helps overcome the hallucination issue [111].

5 Augmented LLM

LLMs are constrained by their training data. If a user's prompt requires domain-specific knowledge, such as data related to a company's Service Level Agreement (SLA), LLMs may not deliver accurate responses in such cases.

While ICL techniques require users to provide examples in the prompt, such as few-shot learning, augmented LLMs incorporate methods that access external resources and tools to improve the model's performance. This augmentation can be integrated into LLMs either during the training or inference stage [62]. In this section, we explore several categories of augmented LLMs.

5.1 Retrieval Augmented LLM

One of the most common techniques in augmented LLMs is retrieving relevant information from documents. This enables LLMs to more accurately generate output for prompts that require context beyond their training data or to reduce hallucinations in the model's output [8]. This technique also helps bridge the gap between smaller and larger models [43]. Retrieval-augmented LLMs typically consist of two main components: the retriever and the LLM. Various approaches involve incorporating retrieval information into the prompt or using it for fine-tuning either the LLM, the retriever, or both.

The retriever component can either use the prompt as a query [67] or re-prompt the LLM component to generate a query based on the initial prompt [59]. Following this, it applies the query to retrieve documents, which can be general, such as the knowledge base on Wikipedia [111], documents relevant to a specific domain [95], or even a cache of recent prompts [34].

To enhance the retriever's performance, instead of using a sparse bag-of-words vector to find relevant documents [20], all retrieved documents can be encoded into dense vectors [65]. Similarly, the query or prompt can be converted into a dense vector, and then semantic similarity [21] is computed between vectors to identify relevant information.

The retriever can incorporate relevant examples into the prompt to enhance the performance of ICL approaches such as few-shot learning [67]. Another technique involves combining CoT with retrievers. The retriever supports explaining each step in the planning process [38] or guides the reasoning step in CoT [88]. The augmented prompt is then passed to the LLM component to generate the desired output. Notably, these two approaches do not require additional training or tuning.

The retrieval information can also be employed to fine-tune the LLM. An example of such a method is RETRO [8], which is based on the auto-regressive LLM, GPT. RETRO converts external database retrieval into dense vectors, and then it splits input tokens into sequences of chunks, retrieves the nearest neighbors to each chunk in the retrieval database, and encodes them together with input to generate output.

On the flip side, fine-tuning can be applied to the retriever component to enable it to add more relevant examples to the prompt while keeping the LLM frozen. An example of this approach involves employing reinforcement learning techniques, with rewards sent back to the retriever component to improve the relevance of retrieved information for the initial prompt [5]. The other technique involves training both the retriever and LLMs. Retrieval-augmentation generation (RAG) combines a pre-trained retriever with a pre-trained sequence-to-sequence LLM and then fine-tuning them in an end-to-end process on a question-answering task [51].

5.2 Web Augmentation

Instead of solely relying on local storage to retrieve information, various methods involve collecting context relevant to the prompt through Web searches and then incorporating that content into the prompt or using it for fine-tuning the model. This process assists LLMs in generating updated output for prompts, such as inquiries about the temperature. For example, WebGPT [66] can engage with a Web browsing environment to discover answers to questions in the prompt. Another example is BlenderBot [83], which trains a model to generate search queries based on the prompt, then executes the query on a search engine, and incorporates the response relevant to the query into the model through a continuous learning process.

5.3 Tool Augmentation

While RAG relies on a retriever component to provide relevant context for enhancing model performance and refining the inference step, tool augmentation involves using a tool to provide the relevant context. This tool can be applied to the initial output of the LLM to provide feedback or evaluation, thereby augmenting the initial prompt with this feedback and iteratively re-prompting the model to enhance its output [19]. For example, an interpreter could be employed to execute the initial output of the model and augment the initial prompt with an error message, facilitating the model in improving its initial output [79]. Additionally, diverse tools can be employed to execute each step of the prompt after dividing the initial prompt into sub-tasks in CoT setup. An LLM can also serve as a tool, for instance, to generate a plan for a prompt (planning a prompt) [110] or to validate its output using verification questions [25].

6 Dataset and Evaluation

The LLMs used for SE tasks often rely on open-source repositories for training and fine-tuning [12]. Before fine-tuning the model for a specific task, there is a pre-training step on textual data to enhance the language understanding capabilities of the model [91]. Different studies use pre-trained LLMs either for inferring a task or fine-tuning the pre-trained model for specific downstream tasks [26, 74]. Platforms such as GitHub and StackOverflow provide vast code and textual data, serving as resources for tuning LLMs for SE tasks.

Several benchmark datasets are commonly used in evaluating LLMs for diverse SE tasks. Among them, we can point to CodexGLUE [57, 58] dataset, collected for evaluating the general language understanding of LLMs for different code-related tasks. This benchmark includes 14 datasets across 10 different code-related tasks such as clone detection, defect detection, code completion, code translation, and code summarization. For the test case generation task, datasets like ATLAS [97] and Methods2Test [89] are employed to fine-tune and evaluate LLMs for generating test cases in Java. The PROMISE NFR dataset [44], on the other hand, is used in studies leveraging LLMs for classifying project requirements.

Datasets like Humaneval [12] and APPs [39] are also commonly used for evaluating LLMs in tasks requiring code generation, but they often incorporate programming competition tasks. In contrast, CoderEval [113] is a benchmark dataset that collects programming tasks from more real-world programming scenarios.

Regarding the evaluation metrics, given the diversity of SE tasks, a single evaluation metric may not adequately capture the performance of LLMs for different tasks. Studies typically employ a range of metrics based on the specific problem types. Metrics like F1-score or precision find application in tasks such as code classification [40]. For evaluating the generative capability of LLMs, metrics such as BLEU [96], CodeBLEU [78], Exact Match (EM) [90], and Pass@k [12] are commonly used. Metrics like BLEU score and EM are more useful for tasks such as code review or code summarization because the output of the model is textual. But code generation and test generation tasks demand accuracy that extends beyond matching ground truth. An accurate output for these types of tasks should be compiled, effective, and implement the requirements outlined in the task description. Thus, metrics like Pass@k, which execute code on certain test cases, are more practical in these scenarios. In tasks like program repair, the evaluation metric also pertains to the correctness of the code after bug repair [45].

Furthermore, different quality metrics in SE can be employed to evaluate LLM output across different SE tasks. Metrics such as cyclomatic complexity [18], test coverage [79], mutation score [19], code/test smells [84], and vulnerabilities [60, 72] serve as benchmark metrics for assessing the quality of outputs generated by LLMs in diverse SE tasks.

7 Tools or Libraries

Various libraries are available for the training, tuning, and inference of LLMs, including Transformers [101], DeepSpeed [77], BMTrain [7], PyTorch [71], and TensorFlow [1]. Additionally, there are tools designed to facilitate the process of prompting LLMs and building applications with them.

LangChain [50] is a framework tailored for developing applications that leverage LLMs. The primary concept behind this tool involves facilitating the chain of various components around an LLM to build more advanced use cases, such as a Chatbot. LangChain offers diverse prompt templates, short-term and long-term memory access for retrieval setups, and interaction capabilities with different LLMs.

AutoGen [103, 104] is another framework that empowers the development of LLM applications by employing multiple agents capable of communicating with each other to solve different tasks. AutoGen features customizable agents with the core of LLM and also allows human participation and the incorporation of various tools. The framework also supports different prompt templates.

Furthermore, Guidance [36] is a tool that enhances the effective use of various ICL prompts, such as CoT, and simplifies the overall structure for different prompt templates.

The GitHub repository Parameter-Efficient Fine-Tuning (PEFT) [61] provides various efficient tuning approaches for adapting Pre-trained LLMs to downstream applications without fine-tuning all the model's parameters. This repository includes LoRA [41]/AdaLoRA [116] and Prefix Tuning [52]. Additionally, it supports numerous models such as GPT-2 and LLaMA.

8 Discussion and Conclusion

Leveraging LLMs for SE tasks poses several challenges and limitations. One of the challenges is the demand for high-quality data for effective training and tuning of LLMs for different SE tasks. Additionally, the training and tuning processes are resource-intensive and require significant time and computational cost. There is also a lack of effective resource-efficient adaptation methods for LLMs. While the literature has introduced numerous efficient tuning methods as mentioned in Sect. 3.4, the majority of these techniques have been evaluated on small-scale pre-trained language models rather than LLMs. As of now, there remains a notable absence of comprehensive research examining the impact of various efficient tuning methods on large-scale language models across diverse settings or tasks.

Various techniques have been proposed on the prompt side to adapt models for new, unseen tasks, such as ICL. However, one of the limitations of these techniques is the restricted amount of content that can be incorporated into the prompt because of the context window size of LLMs.

On the other side, LLMs are limited by information and knowledge in their training dataset, which limits their adaptability to evolving scenarios. To overcome this limitation, various techniques, like RAG, have been proposed to augment the new information relevant to the prompt into the LLMs either during tuning or inference.

LLMs may also generate hallucinations when producing outputs that are plausible responses but incorrect. Evaluation metrics such as the correct ratio for code generation tasks can aid in detecting hallucinations by identifying code that fails in certain test cases. However, LLMs may occasionally overlook specifications in the task description, which may not be detected with test cases and need human experts to filter them out.

Another limitation pertains to the fact that the outputs of LLMs are sometimes buggy, inaccurate, biased, or harmful. It is necessary to filter these outputs before presenting them to end users. Studies have employed the RLHF technique to enhance the model's output by rewarding good-quality responses. However, a notable limitation is associated with the efforts and time required for learning a reward model based on human feedback.

Moreover, numerous quality evaluation metrics in SE require the execution of the code generated by LLMs, facing challenges when evaluating code that is not self-contained and has dependencies. Exploring the training of a model that can predict code quality could be an interesting direction to address this limitation. Leveraging LLMs as a tool to enhance their own output, such as fixing bugs or generating test cases to evaluate the generated code, also can be beneficial in addressing this limitation.

LLMs also face challenges when addressing complex SE tasks. While these models perform a good performance on benchmark datasets with fewer dependencies that share the same distribution as their training data, they may face challenges in scalability and robustness when deployed in real-world environments, such as software projects. The scalability challenge arises from the size and computational cost of these models, making their deployment and real-time usage challenging. For instance, correctly completing a single programming task may require considering the contexts of various dependencies. As for robustness, the issue lies in the presence of diverse data or prompts in software projects that fall out of the distribution of the LLMs' training data, impacting their performance in real-world environments compared to their performance on benchmark datasets [28].

Another key concern arises from the memorization issue in LLMs, where models generate entire sequences of tokens verbatim from their training data [6]. This problem is triggered, for example, when the prompt precisely matches the content in the model's training data. Consequently, the model generates the sequence of tokens from its training data in the output to complete the prompt rather than generalizing it. Many benchmark datasets in SE are sourced from GitHub or StackOverflow and are already part of the training data for LLMs. Using these benchmarks to evaluate LLMs can impact the quality of evaluation due to the memorization issue. There is a lack of more comprehensive datasets that are not a part of the training data of LLMs to evaluate their performance for different SE tasks. Therefore, another potential future direction could involve constructing benchmark datasets beyond HumanEval to evaluate LLMs for various SE tasks.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: {TensorFlow}: a system for {Large-Scale} machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283 (2016)
2. Agarap, A.F.: Deep learning using rectified linear units (ReLU). Preprint (2018). arXiv:1803.08375
3. Ahmed, T., Devanbu, P.: Few-shot training llms for project-specific code-summarization. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–5 (2022)

4. Ali, M., Fromm, M., Thellmann, K., Rutmann, R., Lübbering, M., Leveling, J., Klug, K., Ebert, J., Doll, N., Buschhoff, J.S., et al.: Tokenizer choice for llm training: Negligible or crucial? Preprint (2023). arXiv:2310.08754
5. Bacciu, A., Cocunasu, F., Siciliano, F., Silvestri, F., Tonellotto, N., Trappolini, G.: Rraml: Reinforced retrieval augmented machine learning. Preprint (2023). arXiv:2307.12798
6. Biderman, S., Prashanth, U.S., Sutawika, L., Schoelkopf, H., Anthony, Q., Purohit, S., Raf, E.: Emergent and predictable memorization in large language models. Preprint (2023). arXiv:2304.11158
7. Bmtrain: Efficient training for big models (2021). https://github.com/OpenBMB/BMTrain
8. Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., Van Den Driessche, G.B., Lespiau, J.B., Damoc, B., Clark, A., et al.: Improving language models by retrieving from trillions of tokens. In: International Conference on Machine Learning, pp. 2206–2240. PMLR (2022)
9. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Adv. Neural Inf. Process. Syst. **33**, 1877–1901 (2020)
10. Cao, J., Li, M., Wen, M., Cheung, S.c.: A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. Preprint (2023). arXiv:2304.08191
11. Chang, Y., Wang, X., Wang, J., Wu, Y., Zhu, K., Chen, H., Yang, L., Yi, X., Wang, C., Wang, Y., et al.: A survey on evaluation of large language models. Preprint (2023). arXiv:2307.03109
12. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. Preprint (2021). arXiv:2107.03374
13. Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., et al.: Palm: Scaling language modeling with pathways. Preprint (2022). arXiv:2204.02311
14. Ciborowska, A., Damevski, K.: Fast changeset-based bug localization with bert. In: Proceedings of the 44th International Conference on Software Engineering, pp. 946–957 (2022)
15. Ciniselli, M., Cooper, N., Pascarella, L., Poshyvanyk, D., Di Penta, M., Bavota, G.: An empirical study on the usage of bert models for code completion. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 108–119. IEEE (2021)
16. Clark, K., Luong, M.T., Le, Q.V., Manning, C.D.: Electra: Pre-training text encoders as discriminators rather than generators. Preprint (2020). arXiv:2003.10555
17. Cohen, E., Beck, C.: Empirical analysis of beam search performance degradation in neural sequence models. In: International Conference on Machine Learning. pp. 1290–1299. PMLR (2019)
18. Dakhel, A.M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., Jiang, Z.M.J.: Github Copilot AI pair programmer: Asset or liability? J. Syst. Software **203**, 111734 (2023)
19. Dakhel, A.M., Nikanjam, A., Majdinasab, V., Khomh, F., Desmarais, M.C.: Effective test generation using pre-trained large language models and mutation testing (2023). https://arxiv.org/abs/2308.16557
20. Dang, V., Bendersky, M., Croft, W.B.: Two-stage learning to rank for information retrieval. In: Advances in Information Retrieval: 35th European Conference on IR Research, ECIR 2013, Moscow, Russia, March 24–27, 2013. Proceedings 35, pp. 423–434. Springer (2013)
21. De Boom, C., Van Canneyt, S., Bohez, S., Demeester, T., Dhoedt, B.: Learning semantic similarity for very short texts. In: 2015 IEEE International Conference on Data Mining Workshop (ICDMW), pp. 1229–1234. IEEE (2015)
22. Dettmers, T., Lewis, M., Belkada, Y., Zettlemoyer, L.: Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. Adv. Neural Inf. Process. Syst. **35**, 30318–30332 (2022)
23. Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L.: Qlora: Efficient finetuning of quantized llms. Preprint (2023). arXiv:2305.14314
24. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. Preprint (2018). arXiv:1810.04805

25. Dhuliawala, S., Komeili, M., Xu, J., Raileanu, R., Li, X., Celikyilmaz, A., Weston, J.: Chain-of-verification reduces hallucination in large language models. Preprint (2023). arXiv:2309.11495
26. Dinella, E., Ryan, G., Mytkowicz, T., Lahiri, S.K.: Toga: A neural method for test oracle generation. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2130–2141 (2022)
27. Dong, Q., Li, L., Dai, D., Zheng, C., Wu, Z., Chang, B., Sun, X., Xu, J., Sui, Z.: A survey for in-context learning. Preprint (2022). arXiv:2301.00234
28. Du, M., He, F., Zou, N., Tao, D., Hu, X.: Shortcut learning of large language models in natural language understanding: A survey. Preprint (2022). arXiv:2208.11857
29. Du, Z., Qian, Y., Liu, X., Ding, M., Qiu, J., Yang, Z., Tang, J.: Glm: General language model pretraining with autoregressive blank infilling. Preprint (2021). arXiv:2103.10360
30. Dubey, S.R., Singh, S.K., Chaudhuri, B.B.: Activation functions in deep learning: A comprehensive survey and benchmark. Neurocomputing **503**, 92–108 (2022)
31. Floridi, L., Chiriatti, M.: Gpt-3: Its nature, scope, limits, and consequences. Minds Mach. **30**, 681–694 (2020)
32. Fu, M., Tantithamthavorn, C.: Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 608–620 (2022)
33. Gao, Z., Feng, A., Song, X., Wu, X.: Target-dependent sentiment classification with bert. IEEE Access **7**, 154290–154299 (2019)
34. Gim, I., Chen, G., Lee, S.s., Sarda, N., Khandelwal, A., Zhong, L.: Prompt cache: Modular attention reuse for low-latency inference. Preprint (2023). arXiv:2311.04934
35. Goyal, T., Li, J.J., Durrett, G.: News summarization and evaluation in the era of gpt-3. Preprint (2022). arXiv:2209.12356
36. Guidance: A programming paradigm to conventional prompting and chaining (2023). https://github.com/guidance-ai/guidance
37. Guo, Y., Zheng, Y., Tan, M., Chen, Q., Li, Z., Chen, J., Zhao, P., Huang, J.: Towards accurate and compact architectures via neural architecture transformer. IEEE Trans. Pattern Anal. Mach. Intell. **44**(10), 6501–6516 (2021)
38. He, H., Zhang, H., Roth, D.: Rethinking with retrieval: Faithful large language model inference. Preprint (2022). arXiv:2301.00303
39. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al.: Measuring coding challenge competence with apps. corr abs/2105.09938 (2021). Preprint (2021). arXiv:2105.09938
40. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H.: Large language models for software engineering: A systematic literature review. Preprint (2023). arXiv:2308.10620
41. Hu, E.J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRA: Low-rank adaptation of large language models. In: International Conference on Learning Representations (2022). https://openreview.net/forum?id=nZeVKeeFYf9
42. Ippolito, D., Kriz, R., Kustikova, M., Sedoc, J., Callison-Burch, C.: Comparison of diverse decoding methods from conditional language models. Preprint (2019). arXiv:1906.06362
43. Izacard, G., Lewis, P., Lomeli, M., Hosseini, L., Petroni, F., Schick, T., Dwivedi-Yu, J., Joulin, A., Riedel, S., Grave, E.: Few-shot learning with retrieval augmented language models. Preprint (2022). arXiv:2208.03299
44. Jane Cleland-Huang, Sepideh Mazrouee, H.L., Port, D.: The promise repository of empirical software engineering data (2007). https://zenodo.org/records/268542
45. Jiang, N., Liu, K., Lutellier, T., Tan, L.: Impact of code language models on automated program repair. Preprint (2023). arXiv:2302.05020
46. Jiang, X., Dong, Y., Wang, L., Shang, Q., Li, G.: Self-planning code generation with large language model. Preprint (2023). arXiv:2303.06689
47. Kheiri, K., Karimi, H.: Sentimentgpt: Exploiting gpt for advanced sentiment analysis and its departure from current machine learning. Preprint (2023). arXiv:2307.10234

48. Kudo, T.: Subword regularization: Improving neural network translation models with multiple subword candidates. Preprint (2018). arXiv:1804.10959
49. Kudo, T., Richardson, J.: Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. Preprint (2018). arXiv:1808.06226
50. Langchain: A primer on developing llm apps fast (2023). https://github.com/langchain-ai/langchain
51. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., et al.: Retrieval-augmented generation for knowledge-intensive nlp tasks. Adv. Neural Inf. Process. Syst. **33**, 9459–9474 (2020)
52. Li, X.L., Liang, P.: Prefix-tuning: Optimizing continuous prompts for generation. Preprint (2021). arxiv:2101.00190
53. Li, X., Gong, Y., Shen, Y., Qiu, X., Zhang, H., Yao, B., Qi, W., Jiang, D., Chen, W., Duan, N.: Coderetriever: A large scale contrastive pre-training method for code search. In: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pp. 2898–2910 (2022)
54. Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., et al.: Automating code review activities by large-scale pre-training. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1035–1047 (2022)
55. Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., Cobbe, K.: Let's verify step by step. Preprint (2023). arXiv:2305.20050
56. Liu, Z., Oguz, B., Zhao, C., Chang, E., Stock, P., Mehdad, Y., Shi, Y., Krishnamoorthi, R., Chandra, V.: Llm-qat: Data-free quantization aware training for large language models. Preprint (2023). arXiv:2305.17888
57. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, et al.: Codexglue: A machine learning benchmark dataset for code understanding and generation (2021). https://github.com/microsoft/CodeXGLUE
58. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al.: Codexglue: A machine learning benchmark dataset for code understanding and generation. Preprint (2021). arXiv:2102.04664
59. Ma, X., Gong, Y., He, P., Zhao, H., Duan, N.: Query rewriting for retrieval-augmented large language models. Preprint (2023). arXiv:2305.14283
60. Majdinasab, V., Bishop, M.J., Rasheed, S., Moradidakhel, A., Tahir, A., Khomh, F.: Assessing the security of github copilot generated code—a targeted replication study. Preprint (2023). arXiv:2311.11177
61. Mangrulkar, S., Gugger, S., Debut, L., Belkada, Y., Paul, S., Bossan, B.: Peft: State-of-the-art parameter-efficient fine-tuning methods (2022). https://github.com/huggingface/peft
62. Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A., et al.: Augmented language models: a survey. Preprint (2023). arXiv:2302.07842
63. Mielke, S.J., Alyafeai, Z., Salesky, E., Raffel, C., Dey, M., Gallé, M., Raja, A., Si, C., Lee, W.Y., Sagot, B., et al.: Between words and characters: a brief history of open-vocabulary modeling and tokenization in nlp. Preprint (2021). arXiv:2112.10508
64. Min, B., Ross, H., Sulem, E., Veyseh, A.P.B., Nguyen, T.H., Sainz, O., Agirre, E., Heintz, I., Roth, D.: Recent advances in natural language processing via large pre-trained language models: A survey. ACM Comput. Surv. **56**(2), 1–40 (2023)
65. Mitra, B., Craswell, N.: Neural models for information retrieval. Preprint (2017). arXiv:1705.01509
66. Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., et al.: Webgpt: Browser-assisted question-answering with human feedback. Preprint (2021). arXiv:2112.09332
67. Nashid, N., Sintaha, M., Mesbah, A.: Retrieval-based prompt selection for code-related few-shot learning. In: Proceedings of the 45th International Conference on Software Engineering (ICSE'23) (2023)