Large Language Models for Software Engineering: Survey and Open Problems

Angela Fan
Generative AI Team
Meta Platforms Inc.
New York, NY, USA

Beliz Gokkaya
PyTorch Team
Meta Platforms Inc.
Menlo Park, CA, USA

Mark Harman
Instagram Product Foundation
Meta Platforms Inc.
London, UK

Mitya Lyubarskiy
Developer Infrastructure
Meta Platforms Inc.
London, UK

Shubho Sengupta
FAIR
Meta Platforms Inc.
Menlo Park, CA, USA

Shin Yoo
School of Computing
KAIST
Daejeon, Korea

Jie M. Zhang
Department of Informatics
King's College London
London, UK

Abstract—This paper provides a survey of the emerging area of Large Language Models (LLMs) for Software Engineering (SE). It also sets out open research challenges for the application of LLMs to technical problems faced by software engineers. LLMs' emergent properties bring novelty and creativity with applications right across the spectrum of Software Engineering activities including coding, design, requirements, repair, refactoring, performance improvement, documentation and analytics. However, these very same emergent properties also pose significant technical challenges; we need techniques that can reliably weed out incorrect solutions, such as hallucinations. Our survey reveals the pivotal role that hybrid techniques (traditional SE plus LLMs) have to play in the development and deployment of reliable, efficient and effective LLM-based SE.

Index Terms—Automated Program Repair, Documentation generation, Generative AI, Genetic Improvement, Human-Computer Interaction, Large Language Models, Refactoring, Requirements engineering, Search Based Software Engineering (SBSE), Software Analytics, Software Engineering Education, Software Processes, Software Maintenance and Evolution, Software Testing.

I. INTRODUCTION

This paper surveys the recent developments, advances and empirical results on LLM-based SE; the application of Large Language Models (LLMs) to Software Engineering (SE) applications. We use the survey to highlight gaps in this rapidly developing, but as yet embryonic, research literature. Based on gaps in the literature and technical opportunities, we also identify open problems and challenges for the software engineering research community.

While any survey of such a rapidly expanding area can neither aspire nor claim to be comprehensive, we hope that this survey will provide a useful and relatively complete snapshot of the early universe of this exciting new subdiscipline of Software Engineering: LLM-based Software Engineering. Although the scientific and technical structure of the field is still emerging, it is already possible to identify trends, productive avenues for future research, and important technical challenges that need to be addressed.

In particular, we are already able to discern important connections to (and resonance with) existing trends and well-established approaches and subdisciplines within Software Engineering. Furthermore, although we find considerable grounds for optimism, there remain important technical challenges, which are likely to inform the research agenda for several years. Many authors have highlighted, both scientifically and anecdotally, that hallucination is a pervasive problem for LLMs [1] and also that it poses specific problems for LLM-based SE [2]. As with human intelligence, hallucination means that the LLM can create fictitious output. In the context of software engineering, it means that the engineering artefacts created could be incorrect, yet appear plausible; LLMs may introduce bugs.

However, unlike many other applications of LLMs, software engineers are typically blessed with automatable ground truth (software execution), against which most software engineering artefacts can be evaluated. Also, the software engineering research community has already devoted a great deal of time to producing automated and semi-automated techniques for checking the potentially incorrect results produced by humans. This means that, for the discipline and the research community, there is a great deal of experience and expertise on which to draw, when tackling the challenges posed by issues like hallucination.

Clearly, automated testing techniques [3]–[5] will have a central role to play in ensuring correctness, just as they already do for human-engineered artefacts. When generating entirely new features and systems, automated test data generation suffers from the lack of an automatable oracle [6] (an automated technique for determining whether output behaviour is correct for a given input stimulus). Given LLMs' propensity to hallucinate, the Oracle Problem will remain highly relevant, and solutions to it will become all the more impactful [7].

However, some SE applications concern adaption, improvement and development of *existing* software systems, for which there *is* a readily-available automatable oracle: the functional behaviour of the original system.

In this paper, we call this the 'Automated Regression Oracle', an approach that has already proved advantageous in the field of Genetic Improvement [8]. The Automated Regression Oracle simply uses the existing version of the software system as a reference against which to benchmark output from any subsequent adaptions and changes.

Of course, there is a risk of 'baking in' functional incorrectness, since the Automated Regression Oracle cannot detect what the system *should* do, but only capture what it currently does. Therefore, the Automated Regression Oracle can test only for functional regressions so it is best suited to use cases where the existing functionality is to be maintained. For example, for non-functional improvements such as performance optimisation and for semantics-preserving refactoring.

The input provided to an LLM will be a natural focus of growing research, and we can expect a rapid development of the literature on prompt engineering and prompt optimisation [9]. In this survey, we highlight existing work and open challenges for prompt engineering with regard to several specific aspects of software engineering.

The output from an LLM need not be confined purely to code, but can also include other software engineering artefacts, such as requirements, test cases, design diagrams, and documentation. In general, the language-based nature of an LLM, allows it to generate any linguistically-defined software engineering artefact.

We typically think of the software engineering artefact as the primary output of the LLM, but it is not the only output. The *explanation* provided with the primary output is also an important output of any LLM. Our survey highlights the need for much more research, not only into optimising prompt engineering (which focuses on the input to the LLM) but also the need for work on the optimisation of explanations provided with the primary output.

LLMs are inherently nondeterministic: the same prompt produces different answers on different inference executions (unless the temperature is set to zero, which has often been found to be suboptimal over multiple executions) [10]. Furthermore, irrespective of the temperature setting, subtle changes in the prompt can lead to very different outputs [10]. As well as motivating 'prompt engineering' and output processing, this nondeterministic behaviour raises challenges for the scientific evaluation of LLM-based Software Engineering:

> If results can vary each time we run the process, how can we determine whether a proposed technique achieves an advance over the state of the art?

This is a problem that has already been well studied in the context of Empirical Software Engineering [11] and Search Based Software Engineering (SBSE) [12]. In particular, SBSE bears many similarities to LLM-based Software Engineering, sharing with it the need to achieve robust scientific evaluation in the presence of noisy, non-deterministic, and incomplete results [13], [14]. There is, therefore, already a mature software engineering literature on just the kind of robust scientific evaluation techniques needed to cater for LLM-based scientific evaluation.

For example, well-studied techniques, such as parametric and non-parametric inferential statistics, are now routinely used to provide robust scientific conclusions in the presence of highly non-deterministic algorithms in the SBSE discipline.

TABLE I
A (ALL) DENOTES ALL PREPRINTS THAT ARE CATEGORISED UNDER CS (COMPUTER SCIENCE). L (LLM) DENOTES PREPRINTS WHOSE TITLE OR ABSTRACT INCLUDES "LLM", "LARGE LANGUAGE MODEL", OR "GPT". $L \cap S$ DENOTES PREPRINTS IN CS.SE OR CS.PL CATEGORY WHOSE TITLE OR ABSTRACT INCLUDES THE SAME KEYWORDS. NOTE THAT THE YEAR 2023 ONLY INCLUDES DATA UP TO 27 JULY 2023.

Year	$|A|$	$|L|$	$|L \cap S|$	$\frac{|L|}{|A|}(\%)$	$\frac{|L \cap S|}{|L|}(\%)$
2007	2,238	0	0	0.00	0.00
2008	3,645	0	0	0.00	0.00
2009	4,873	0	0	0.00	0.00
2010	7,543	0	0	0.00	0.00
2011	9,114	0	0	0.00	0.00
2012	12,316	0	0	0.00	0.00
2013	14,933	0	0	0.00	0.00
2014	16,320	0	0	0.00	0.00
2015	18,818	0	0	0.00	0.00
2016	23,707	0	0	0.00	0.00
2017	30,746	0	0	0.00	0.00
2018	41,927	0	0	0.00	0.00
2019	55,325	36	0	0.00	0.00
2020	71,431	99	5	0.00	5.05
2021	77,520	192	13	0.25	6.77
2022	81,964	434	45	0.53	10.36
2023	52,547	1,665	181	3.17	10.87

In order to understand the growth trends within LLM-based Software Engineering, we performed a manual analysis of data on the number of publications on specific topics from arXiv. Table I contains the raw data[1], which was manually extracted from the arXiv metadata dump made publicly available via Kaggle (https://www.kaggle.com/datasets/Cornell-University/arxiv), accessed on the $27^{th.}$ July 2023. We first filtered out publications for which the classification code does not start with the cs prefix (i.e., Computer Science), resulting in column A.

To identify Computer Science papers that are relevant to LLMs, we filtered the publications into subcategories on artificial intelligence (cs.AI), machine learning (cs.LG), neural and evolutionary computation (cs.NE), software engineering (cs.SE), and programming language (cs.PL) using the queries "Large Language Model", "LLM", and "GPT" in either the title or the abstract (we manually excluded instances of overloaded acronyms such as GPT for General Planning Tool), resulting in column L. Finally, we used the same queries to identify LLM-based Software Engineering papers in software engineering (cs.SE) and programming language (cs.PL). These queries are inherently approximate, so we confine ourselves only to conclusions based on overall trends for which there is strong evidence rather than specific details of the numbers observed. Nevertheless, we report the raw numbers observed to support replication by others.

---

[1]The numbers for 2023 are underestimated since the data was accessed in July 2023.
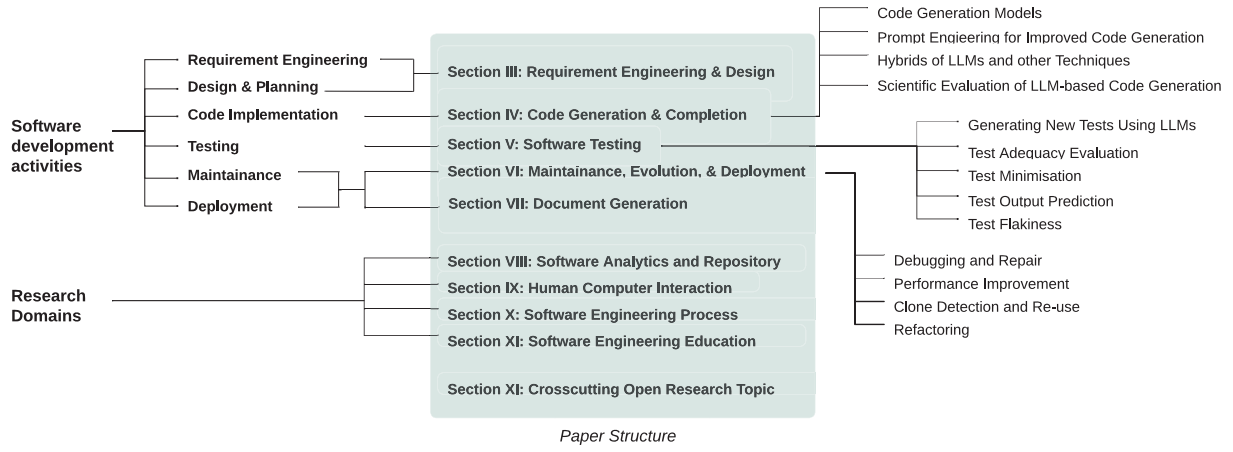
Fig. 1. A mapping between software development activities, research domains, and the paper structure
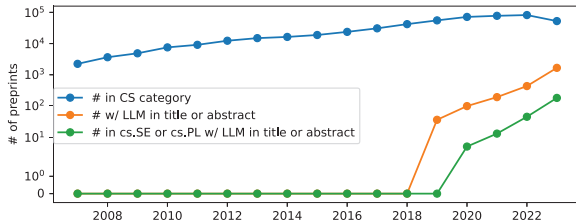

Fig. 2. Trends in number of arXiv preprints. The blue line denotes the number of preprints categorised under "CS". The orange line denotes the number of preprints in AI (cs.AI), Machine Learning (cs.LG), Neural and Evolutionary Computing (cs.NE), Software Engineering (cs.SE), and Programming Language (cs.PL) whose title or abstract contains either "Large Language Model", "LLM", or "GPT". The green line denotes the number of preprints in SE and PL categories whose title or abstract contains either "Large Language Model", "LLM", or "GPT"
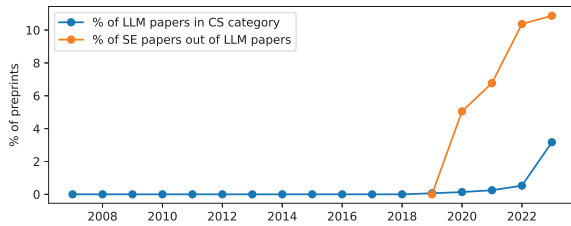

Fig. 3. Proportions of LLM papers and SE papers about LLMs. By "about LLMs", we mean that either the title or the abstract of a preprint contains "LLM", "Large Language Model", or "GPT". The blue line denotes the percentage of the number of preprints about LLMs out of the number of all preprints in the CS category. The orange line denotes the percentage of the number of preprints about LLMs in cs.SE and cs.PL categories out of all preprints about LLMs

Figure 2, shows the growth in the number of arXiv-published papers on Computer Science ($|A|$, in Blue), and on LLMs ($|L|$, in orange). Those papers specifically on Software Engineering *and* LLMs are depicted in Green ($|L \cap S|$). Given the rapid rise in overall publication volumes, we use a logarithmic scale for the vertical axis. Unsurprisingly, we see an overall rise in the number of CS publications.

Also, given the recent upsurge in attention for LLMs, the exponential rise in the number of papers on LLMs is relatively unsurprising.

Perhaps more interesting is the rapid uptake of Software Engineering applications of LLMs, as revealed by the growth trend, pictured in green on this figure. In order to examine this trend in more detail, we plot the proportion of LLM publications (L) to all CS publications (A) in blue, as well as the proportions of LLM-based software engineering publications ($L \cap S$) to all LLM publications in orange in Figure 3. As can be seen, the proportion of LLM papers on LLM-based Software Engineering has been rising dramatically since 2019. Already, more than 10% of all papers on LLMs are concerned with LLM-based Software Engineering.

As a result of this growth, we can expect many other surveys of LLM-Based SE. The rapid expansion of the literature makes it unlikely that further comprehensive SE-wide studies will fit the space constraints of a single paper, but we can expect many specific comprehensive surveys of sub-areas of interest, and also Systematic Literature Reviews (SLRs) that tackle SE-wide crosscutting issues by asking specific research questions of the primary literature in the systematic review. Already, such SLRs are appearing. For example, Hou et al. [15] provided an excellent recent SLR covering 229 research papers from 2017 to 2023 reporting SE tasks tackled, data collection and preprocessing techniques, and strategies for optimising LLM performance (such as prompt engineering).

The remainder of this paper is organised to follow the top-level software development activities and research domains as depicted in Figure 1.

II. Preliminaries

A. Large Language Models

A Large Language Model (LLM) refers to an Artificial Intelligence (AI) model that has been trained on large amounts of data and is able to generate text in a human-like fashion. Table III provides a glossary of LLM terminology to make the paper self-contained.

LLMs are typically based on deep learning techniques, such as transformers, and have the capability to generate useful language output. As a result, they have been found capable of performing a wide range of language-related tasks, including text generation [16], answering questions [17], translation [18], summarization [19], and sentiment analysis [20].

Rumelhart et al. [21] introduced the concept of Recurrent Neural Network, opening up the possibility of processing sequential data. Long Short Term Memory (LSTM) architecture, an extension of the RNN architecture introduced by Hochreiter and Schmidhuber [22], significantly improved their performance in many applications.

In 2017, Vaswani et al. [23] introduced the Transformer architecture, which captures word relationships with the self-attention mechanism. The transformer architecture had a profound impact on language modelling and triggered an explosion of activity on LLMs.

In 2018, OpenAI released the Generative Pre-trained Transformer (GPT) model, followed by subsequent iterations (GPT-2, GPT-3, GPT-3.5, and GPT-4). With GPT-3 and 3.5, many observers noticed a significant step change in generative performance, thereby attracting a great deal of interest in GPT (and ChatGPT) in particular, and also in LLMs more generally.

LLMs achieve this performance, in part, due to the large corpora on which they are trained: For example, GPT-3 is trained on 45TB of text data and has 175 billion parameters. Meta launched open-sourced LLaMA in February 2023, which is trained on 1.4 trillion tokens with a variety of model sizes ranging from 7 billion to 65 billion parameters [24].

B. Categories of Large Language Models

There are three categories of large language models:

1) **Encoder-only model**: also known as an autoencoder, consists of an encoder network but does not have a separate decoder network. It takes an input sequence and maps it to a lower-dimensional representation. The purpose of an autoencoder is to learn an encoding of the input data. Examples of Encoder-only LLMs are BERT from Google, RoBERTa from Meta, and DeBERTa from Microsoft [1].

2) **Encoder-decoder model**: in addition to the encoder network, there is a decoder network that generates an output sequence by iteratively generating tokens or symbols based on the context vector and previously generated tokens. It can be adopted for tasks like machine translation or text generation. Examples of Encoder-decoder LLMs are T5 from Google and BART from Meta [1].

3) **Decoder-only model**: Unlike the previous two types of LLMs, decoder-only LLMs do not have an encoder component to process the input data, but only a decoder component that directly generates an output sequence based on a given context or input. Decoder-only models are often based on architectures such as autoregressive models, where the output is generated token-by-token. Each token generated by the decoder is conditioned on the previous tokens generated and the context.

Popular examples of decoder-only models are the GPT (Generative Pre-trained Transformer) series developed by OpenAI, LLaMA from Meta, Claude from Anthropic, and PaLM from Google [1].

C. Large Language Models for Software Engineering

While LLMs have been widely applied to tasks involving natural languages, their application to software development tasks, involving programming languages, has also gained significant recent attention.

In 2021, OpenAI introduced CodeX, a fined-tuned descendant of GPT-3. CodeX is used by GitHub's Copilot, which provides users of Visual Studio Code, Visual Studio, Neovim, and JetBrains with code completion. The new version of Copilot, GitHub Copilot X[2], is based on GPT-4. In February 2023, GitHub reported that, on average, 46%[3] of the developers' code was written by Copilot [25]. For Java only, that number is 62%. Thomas Dohmke, CEO of GitHub, said Copilot will write 80% of code "sooner than later" in June 2023 [26].

In 2022, DeepMind introduced AlphaCode [27], trained with 40B parameters on selected public GitHub repositories. It achieved on average a ranking in the top 54% in competitions with more than 5,000 participants in simulated evaluations.

The most recent GPT model, GPT-4, also performs code generation. According to GPT-4's technical report [28], the zero-shot pass@1 accuracy is 67% with GPT-4 on HumanEval, an open-source dataset from OpenAI consisting of 164 programming problems.

On a benchmark of 100 LeetCode problems, GPT-4 has comparable performance with human developers [29]. On the $24^{th.}$ August 2023, Meta released open-sourced Code Llama [30], a state-of-the-art for publicly available LLMs on coding tasks.

Table II lists the representative LLMs that are designed for code generation/completion based on natural language descriptions.

III. REQUIREMENTS ENGINEERING AND DESIGN

Requirements engineering is an important discipline in software engineering. It forms the fundamental link between the technical attributes of the system software engineers build, and the purpose for which the systems are built. There is a mature literature, and a large research community concerned specifically with problems associated with requirements engineering problems [31].

There has also been previous work on artificial intelligence approaches to support requirements engineering, notably in the form of computational search for requirements engineering [32]. However, hitherto, the discipline of requirements engineering has received less attention from the emerging literature on LLM-based software engineering.

---

[2]GitHub Copilot X is under technical preview at the time we accessed it on July 17th 2023.

[3]In this paper, all percentages are reported with a precision of 2 significant digits.

TABLE II
EXISTING LARGE LANGUAGE MODELS FOR CODE GENERATION

Name	Release date	Produced by	Parameters	Open-sourced	Price	Supported languages	Type
CodeBERT	February 2020	Microsoft	125M	YES	free	6	Encoder-decoder
InCoder	April 2022	Meta	6.7B, 1.3B	YES	free	30	Decoder-only
AlphaCode	February 2022	DeepMind	300M, 1B, 3B, 9B, and 41B	NO	free	Python or C++	Encoder-decoder
CodeX	August 2021	OpenAI	12B	NO	free	>11	Decoder-only
Copilot	October 2021	Github and OpenAI	12B	NO	free for individual developers and organisations	>11	Decoder-only
CodeT5	Nov 2021	Salesforce Research	60M, 220M, 770M	YES	free	6	Encoder-decoder
CodeT5+	May 2023	Salesforce Research	2B, 6B, 16B	YES	free	9	Encoder-decoder
PolyCoder	Oct 2022	Carnegie Mellon University	160M, 400M, 2.7B	YES	free	>11	Decoder-only
CodeWhisperer	April 2023	Amazon	unknown	NO	free for individual developers	15	unknown
WizardCoder	June 2023	Microsoft	15B	YES	free	unknown	Encoder-only
CodeGeeX	Sep 2022	Tsinghua University et al.	13B	YES	free	23	Decoder-only
CodeGen	March 2022	Salesforce Research	350M, 1B, 3B, 7B, 16B	YES	free	Python	Decoder-only
StarCoder	May 2023	BigCode	15B	YES	free	>80	Encoder-only
phi-1	June 2023	Microsoft	1.3B	NOT YET	free	Python	Decoder-only
Code Llama	August 2023	Meta	7B, 13B, 34B	YES	free	>7	Decoder-only

TABLE III
KEY TERMINOLOGY RELATED TO LARGE LANGUAGE MODELS

Term	Explanation
Chain of Thoughts (CoT)	In the context of LLMs, chain of thought represents the logical flow of ideas and reasoning within the text generated by LLMs.
Encoder & Decoder	Encoders are components of LLMs that map any given input of a specific type (such as image, audio, text) into a latent vector space. Decoders perform the reversal: they can take an input from a latent vector space, and (re)constructdata of the original type.
Few-shot learning	A machine learning technique that aims to train models to perform well on new tasks or classes with only a few new items of labelled training data. It is also known as in-context learning. With LLMs, few-shot examples are typically included in the prompt.
Fine-tuning	A process by which a model, trained on a large dataset or a related task, is further trained on a smaller or more specific dataset to improve its performance on the target task or domain.
Generative AI	A category of artificial intelligence that focuses on generating or creating new content, such as images, text, music, and videos.
Parameters	Parameters are the numerical values inside LLMs that are adjusted during training, and primarily include weights and biases. Weights dictate the strength of connections between neurons and serve as coefficients to the input values or activation thresholds for preceding neurons. Biases shift the weighted sum of inputs, before this sum is fed into the activation function. The number of parameters is often used as a measure of the size of an LLM.
Prompt	The input provided to the LLM to stimulate the generation of a response.
Prompt engineering	The process of designing and optimising prompts to achieve desired outcomes.
ReAct	The ReAct (Reasoning and Acting) prompting framework allows LLMs to generate reasoning traces as well as actions specific to the given task. Once an LLM generates an action, it can be carried out externally, and the observation of the output of the action can be included in the next prompt, providing information to the LLM. This enables LLMs to use external tools.
Temperature	A parameter that affects the randomness of the generated content. A higher value (e.g., 1.0) yields more diverse and creative content, while a lower value (e.g., 0.2) yields more deterministic content.
Token	A token is the atomic unit with which an LLM represents its input and output. Tokens are enumerations, and can represent words, characters, subwords or other segments of text and/or code.
Top-N, Pass@N	For many applications, LLMs will typically generate a number of candidate solutions in a ranking. Top-N metrics count the number of problems correctly solved by an LLM with an answer among its Top N candidates. Similarly, Pass@N counts the number of programming questions for which a candidate program within the Top N rank has passed the test case.
Zero-shot learning	A machine learning technique that enables models to generalize and make predictions on classes or tasks that were not seen during the training phase. There is no labelled data available for these new classes.

Zhang et al. [33] conducted a preliminary evaluation of ChatGPT's zero-shot requirement retrieval performance on two requirements analysis tasks over four data sets. Although these results are only preliminary, they provide optimism that LLMs can be used as a support for efficient and effective requirements engineering. Luo et al. [34] conducted prompt engineering with BERT for automatic requirement classification. Luitel et al. [35] focused on requirements completeness and used BERT to generate predictions for filling masked slots in requirements.

A. Open Problems in LLMs for Requirement Engineering

Unlike other software development activities, we did not find much work on LLM-based requirements engineering or on LLM-based design. Indeed, there was even evidence that practising engineers are reluctant to rely on LLMs for higher-level design goals [36]. There is thus a great opportunity to expand on this open field of research.

The majority of LLM applications are focused on tasks such as code generation, testing, and repair. These tasks benefit from LLM's capability to generate code. Nevertheless, LLMs also have significant potential to support requirements engineering activities, thanks to their natural language processing capabilities.

For example, traceability is a long-standing, cross-cutting concern in software engineering. In particular, identifying traceability links between requirements and other engineering artefacts, such as code and tests, are especially challenging because requirements are often written in natural language; a natural fit for LLMs.

IV. CODE GENERATION AND COMPLETION

Of all the Software Engineering application areas for LLMs, code completion is the area that has been most thoroughly explored hitherto. Even prior to the advent of LLMs, it was suggested that learning from existing code repositories is the key to successful and intelligent code completion [37]: pre-trained LLMs deliver on these early aspirations for code completion. While hallucination has been pointed out as the weakness of LLMs more generally, the specific task of code completion sidesteps hallucination problems by acting as a recommender system to the developer. The developer thus bears the responsibility to weed out any hallucinated LLM output before it leaks into the code base.

Of course, a high degree of hallucination would have made code completion recommender systems ineffective. The widespread and rapid adoption, and the positive results already reported for code completion, provide early indications that this has not happened. For example, Murali et al. [38] reported the experience of deploying CodeCompose, a code completion tool based on the Incoder LLM [39], at Meta. During 15 days, 4.5 million code completion suggestions were made by CodeCompose, and the acceptance rate from developers was 22%. The qualitative feedback was highly positive, with 92% positive. Similarly, Peng et al. [40] reported that programmers could complete a non-trivial task (implementing an HTTP server in JavaScript) 56% faster when paired with GitHub Copilot, compared to the control group that did not receive any such support.

Many software engineers already appear to have decided that benefits outweigh any necessary human filtration effort, with enthusiastic levels and rates of adoption already being reported. Once LLM-based code completion is fully adopted, there are expectations that programmers will spend more time reviewing rather than writing code [41].

A. Code Generation Models

Automated code generation has a long history, tracing its origins back to early visions of automated program synthesis [42], which have continued to develop and have generated impressive results [43].

From the pioneering work of Hindle et al. on the naturalness of software [44], we know that programmers write code (and languages enforce code writing styles), that make code highly predictable. Furthermore, Barr et al. [45] found that 43% of commits to a large repository of Java projects could be reconstituted from existing code. They called this 'The Plastic Surgery Hypothesis' because of the way automated repair proceeds by scavenging for existing code to patch up issues elsewhere [46].

Their empirical study provided evidence for the efficacy of this scavenging approach, but also underlined the repetitive and predictable nature of software. In a larger repository (sourceforge), Gabel and Su [47] found that a programmer would have to write more than six lines of code in order to create a novel code fragment.

These research findings on code naturalness, reusability and predictability, make it unsurprising that LLMs have been able to exploit that same predictable reusability to produce effective recommendations for code generation. These observations have underpinned the growth of generate-and-test approaches to repair and genetic improvement [8], [46]. The generate-and-test approach offers greater code transformation freedom (compared to more traditional correct-by-construction approaches [48]), precisely because the generated code may not preserve strict, mathematically-defined (and not always appropriate, nor useful) interpretations of correctness.

This freedom to explore a wider space of "semantic near neighbours" allows Genetic Improvement to find dramatic optimisations (see Section VI-C). The Genetic Improvement approach, nomenclature, and evaluation methodologies also provide a scientific framework within which to understand and evaluate LLM-based code generation. Both technologies share the 'generate-and-test' approach to program transformation and code generation, potentially making much of the existing work on genetic improvement directly applicable to LLM-based code generation.

In 2021, Chen et al. [49] introduced CodeX, a GPT language model fine-tuned on publicly available code from GitHub, and evaluated its Python code-writing capabilities. They released a new evaluation set called 'HumanEval' to measure functional correctness for synthesizing programs from docstrings, and found that CodeX outperformed GPT-3 and GPT-J when tackling these problems. Since then there has been an explosion in research on LLM-based code generation and the HumanEval dataset has been used in many subsequent studies.

In 2022, Li et al. [27] introduced AlphaCode, a system for code generation that creates novel solutions to competitive programming problems. They found that three key components were critical to achieving reliable performance:

1) An extensive programming dataset for training and evaluation.
2) Large and efficient-to-sample transformer-based architectures.
3) Large-scale model sampling to explore the search space, followed by behaviour-based filtering.

In simulated evaluations on programming competitions on the Codeforces platform, AlphaCode achieved, on average, a ranking of the top 54% in competitions with more than 5,000 participants.

Several papers also introduced code synthesis LLMs [50]–[53], based on large data sets with little pre-filtering of the training data. However, in 2023, Gunasekar et al. [54] reported that, by training with only a textbook-quality code corpus, LLMs with lower parameter counts could achieve performance comparable to much larger models.

They classified an existing Python code corpus with the GPT-4 model, by prompting it to determine the educational value of the given code for a student who wants to learn programming. Second, they used GPT-3.5 to create synthetic textbooks about Python. Specific code generation use cases have also been tackled, such as numerical algorithm code generation [55], and generation of code from behavioural descriptions [56]. More examples of the existing LLMs for code generation and the code generation leaderboard can be found in Table II and Figure 4.

B. Prompt Engineering for Improved Code Generation

Prompt engineering has been extensively used as a way to improve code generation. For example, Li et al. [57] reported pass@1 improvements of between approximately 50% and 80% on CodeX, CodeGeeX, CodeGen, and InCoder on several benchmarks (MBPP for Python, MBJP for Java, and MBJSP for JavaScript). Döderlein et al. [58] reported the prompt-engineered improvement of Copilot and CodeX success rates from approximately 1/4 to 3/4 on HumanEval and LeetCode. He and Vechev [59] used prompt engineering to improve the security of LLM-generated code, reporting an improvement in security from 59% (of cases considered) to 92%. White et al. [60] provided a catalogue of prompt engineering design patterns for various software engineering tasks, including code generation. Denny et al. [61] argued that prompt engineering is a useful learning activity that fosters software engineering students' computational thinking.

Other authors have considered ways to decompose prompt engineering into iterative and multiphase conversations with the LLM, moving it closer to Chain of Thought reasoning. For example, Li et al. [62], [63] reported an 18% increase in ChatGPT Pass@1 using a two-stage sketch-based approach, SkCoder, in which the LLM first creates a sketch and then subsequently implements these sketches. Jiang et. al. [64] and Zhang et al. [65] also sought to deploy Chain-of-Thought-style reasoning by prompting LLMs to reflect and self-edit.

Existing software engineering analysis techniques can also provide additional information for fine-tuning and prompt engineering. For example, Ahmed et al. [66] show how simple static analysis can be used in the prompt to improve the performance of code generation with few-shot learning.

Shin et al. [67] compared prompt engineering and fine tuning with GPT-4 for code generation tasks, demonstrating that fine-tuning works better than prompt engineering.

C. Hybrids of LLMs and other Techniques

Throughout our survey of the literature, we found strong evidence that some of the most promising results can be achieved by hybridising; combining LLMs with other existing software engineering techniques. This section surveys work on hybrid LLMs for code generation.

Several authors have developed hybrids of LLMs combined with planning and search. For example, Zhang et al. [68], [69] reported improvements over baselines of between approximately 11% and 27%, while Zhang et al. [70] hybridized code generation with API search techniques.

Hybrid approaches have also used existing software engineering and/or AI techniques to select the best candidate from an LLM's top-n outputs. For example, Chen et al. [71] use test generation to choose candidates and reported improvement of approximately 20% on five pre-trained LLMs; Inala et al. [72] use a neural network-based ranker to predict code correctness and potential faults. Jain et al. [73] proposed Jigsaw, which post-processes the generated code based on program analysis and synthesis techniques.

Dong et al. [74] treated LLMs as agents, letting multiple LLMs play distinct roles in addressing code generation tasks collaboratively and interactively. They reported improvements of approximately 30%-47%.

D. Scientific Evaluation of LLM-based Code Generation

There is a pressing need for more thorough scientific evaluation. Many authors have anecdotally reported on cases where LLMs failed to generate correct, secure, and reliable code. Poldrack et al. [75] also highlight the need for substantial human validation. In this section, we survey the literature on the empirical evaluation of LLM-based code generation in terms of correctness, robustness, explainability, determinism, and security.

1) Correctness Evaluation: The GPT-4 Technical Report [28] evaluated the correctness of GPT-4's code generation on the HumanEval dataset, reporting a zero-shot accuracy of 67%, a modest improvement on the (earlier ChatGPT) results reported by Yetistiren et al. [76].

Borji [77] presented a rigorous, categorised and systematic analysis of LLM code generation failures for ChatGPT. Eleven categories of failures, including reasoning, factual errors, mathematics, coding, and bias, are presented and discussed in their work.

Figure 4 shows the leaderboard of code generation correctness in terms of the pass@1 (i.e., the test pass rate for the top-1 code candidate) on the HumanEval dataset according to Papers With Code, a platform that highlights trending AI research and the code behind the method and models.[4] The LLM models behind each method are shown in brackets. At the time of writing, the best code generation model, Reflexion [78], can generate correct code for over 90% of the generation tasks. However, these numbers and the relative rankings of different language models are inherently subject to change in such a rapidly developing field. For example, the figure given for correct code on HumanEval in the original GPT-4 Report [28] was only 67%, so the updated figure of 80% (at the time of writing, which is five months later) retrieved from the Papers-With-Code website presumably represents the evolution of GPT4 since then.

Despite the promising results in the literature on code generation and completion, Din et al. [79] reported that the performance of code completion dropped by more than 50% on HumanEval when the context contains bugs.

---

[4]The actual leaderboard can be found at https://paperswithcode.com/sota/code-generation-on-humaneval/; results in Figure 4 accessed on 24th August 2023.
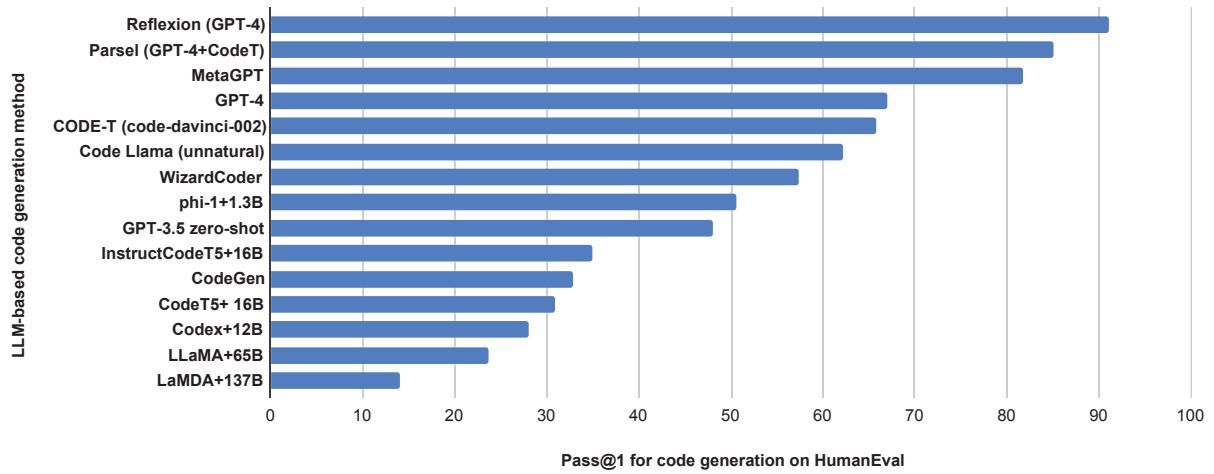
Fig. 4. Code generation leaderboard for the HumanEval benchmark. These methods are either based on LLMs or LLMs themselves.

2) Robustness Evaluation: LLM code generation robustness is the degree to which similar prompts elicit semantically and syntactically similar code generation. Treude [80] introduced GPTCOMPARE, a prototype tool for visually highlighting similarities and differences between LLM code outputs. Yan et al. [81] introduced COCO to test the robustness and consistency of LLM-based code generation systems.

3) Explainability Evaluation: One considerable advantage of LLMs, over previous machine learning techniques, is the way in which the code generation artefacts are accompanied by explanations. Such explanations have the potential to increase adoption, by providing additional confidence and faster understanding. More work is needed to evaluate and optimise explanations that accompany generated code and other software engineering artefacts.

Initial evaluation by MacNeil et al. [82] on their interactive Web development e-book, suggested that a majority of students perceived LLM-generated code explanations to be helpful. Noever and Williams [83] also showed the potential for explanations to help human engineers, particularly where code is obfuscated or lacks sufficient existing documentation. In this way, the ability to produce insight and explanation may go beyond simply justifying the code generated by the LLM itself, but may become a valuable source of education and documentation (See Section XI).

Sun et al. [84] focus on users' explainability needs for generative AI in three software engineering use cases: code generation based on natural language description (with Copilot), translation between different programming languages (with Transcoder), and code autocompletion (with Copilot). Their investigation was conducted as 9 workshops with 43 software engineers and identified 11 categories of explainability needs in the context of Generative AI (GenAI) for code. It also proposed 4 types of features for generative AI: AI documentation, model uncertainty, attention distribution, and social transparency (i.e., making visible the socio-organizational factors that govern the use of AI).

Mohammadkhani et al. [85]used the attention mechanism to study CodeBERT and GraphCodeBERT on tasks including code documentation generation, code refinement, and code translation.

4) Determinism Evaluation: LLMs are nondeterministic. Ouyang et al. [10] empirically studied the non-determinism of ChatGPT in code generation, founding that over 60% of tasks had zero equal test output across different requests. Nevertheless, their study of the literature in LLM-based code generation demonstrate that only 21.1% of these papers consider the non-determinism threat in their experiments.

5) Security Evaluation: Hajipour et al. [86] proposed a few-shot prompting approach to detecting security vulnerabilities, reporting that their approach automatically finds thousands of security vulnerabilities in several models. Khoury et al. [87] found that the code generated by ChatGPT often fell way below even minimal standards of secure coding. Risse and Böme [88] reported results that indicated vulnerability detection accuracy may be over-reported, due to the model overfitting to unrelated training set features .

In addition, Yetistiren et al. [76] presented a comprehensive evaluation of the performance of Copilot, CodeWhisperer, and ChatGPT, covering different aspects including code validity, code correctness, code security, code reliability, and Their results show a wide degree of divergence in performance, motivating the need for further research and investigation. For example, they found 65%, 46%, and 31% of the programs generated by ChatGPT, Copilot, and CodeWhisperer (respectively) were correct.

6) Benchmarks: As with other scientific evaluations, software engineering evaluation relies on publicly available and representative benchmark suites. A number of these have already emerged and can support software engineering evaluation of LLM-based applications. The Papers-With-Code platform[5] provides a summary of 15 benchmarks for evaluating code generation.

[5]https://paperswithcode.com/task/code-generation

Authorized licensed use limited to: Sheffield University. Downloaded on November 13,2024 at 12:31:12 UTC from IEEE Xplore. Restrictions apply.

Evaluations have often relied on small programming problems from programming courses [89], synthetically generated problem sets [90], and online judging platforms such as Leetcode [29], [65], [91]. Although results reported naturally vary by LLM in training sets, the overall conclusions of these evaluations indicate success rates of between 20% and 80%.

Nevertheless, existing code generation benchmarks tend to rely on test suites to automatically judge code correctness, which can be inadequate, leading to false judgements [92]. This highlights the need for more work on evaluation benchmarks that are specifically tailored to LLM-based code generation evaluation. Liu et al. [93] draw attention to the problem, showing how existing test suites can lead to high degrees of false positive conclusions (also a serious problem for online judge platforms [92]). To alleviate this problem, they propose EvalPlus – a code synthesis benchmarking framework that automatically generates test inputs and rigorously evaluates the functional correctness of LLM-generated code. Their evaluation of 14 popular LLMs (including GPT-4 and ChatGPT) demonstrated that with the newly generated tests for HumanEval, the assessment of pass@k drops by up to 15%, averaged over problems considered.

Jimenez et al. [94] introduced SWE-bench with the aim of evaluating LLMs on code generation problems in a realistic software engineering setting. SWE-bench contains 2,294 software engineering problems, drawn from real GitHub issues. The results suggest that Claude 2 and GPT-4 solve only 4.8% and 1.7% of the coding tasks, respectively.

E. Open Problems in Code Generation and Completion

Assessing the generated code remains a critical problem for LLM-based code generation and completion: while much work already started applying existing software testing knowledge to this problem, we expect closer integration of automated testing techniques with code generation and completion techniques.

Fortunately, there is a large body of existing work on automated test data generation [3]–[5], much of which will have an important role to play in ensuring the correctness of the engineering artefacts generated by LLMs. A recurring theme of the challenges covered in this paper, is that code execution provides precisely the 'ground truth' needed to filter hallucinated responses. It can also provide guidance as part of interactive reasoning/action ('ReAct') dialogue [95], both with and within LLMs.

Automated test data generation allows the software engineer to target the exploration of the most relevant regions of this run-time ground truth. This test-based targeting can help filter, fine-tune and to optimise prompts, thereby minimising problems posed by hallucination. LLMs also have considerable potential for automating the process of constructing effective and efficient software test suites.

Another important problem is how to efficiently fine-tune pre-trained LLMs so that they perform better for a specific programming language, codebase, or domain: this is especially important because training an LLM from scratch requires significant computational resources.

For example, transfer learning has been proposed as a way to improve code completion performance when the volume of training examples for a specific programming language is inadequate [96].

The current focus of research is on the code produced by LLMs. However, the explanations produced by LLMs may turn out to be at least as important. One could imagine many scenarios in which an engineer would prefer to accept a (possibly) suboptimal software engineering artefact that comes with a compelling explanation, over a potentially more performant solution with a less compelling explanation. After all, engineers regularly make the same judgement call for human-designed engineering artefacts, so why would we expect it to be any different for those produced by machines? As with prompt engineering, which focuses on optimising the input to the LLM, explanation engineering is also likely to become an area of study in its own right.

V. SOFTWARE TESTING

Software testing is a well-established research discipline, the origins of which can be traced back to Turing's pioneering work in the late 1940s [97]. Much of the focus of this research has been on the automated generation of test suites, able to achieve high fault revelation potential at low computational cost [3]–[5]. This provides us with, not only techniques able to weed out incorrect LLM-generated code, but also a mature baseline against which to compare novel LLM-based and hybrid techniques for test suite generation.

There is already a sufficiently large body of work to warrant a survey specifically on LLM-based Software Testing: Wang et al. [98] presented a survey of papers primarily on testing, but also including debugging and repair. They reported on 52 papers (33 published since 2022) of which approximately one-third concerned test-based LLM fine-tuning, while the remainder relied upon prompt engineering.

A. Generating New Tests Using LLMs

In this section, we review existing work on LLMs for test data generation, before highlighting open problems and challenges for the development of this emerging field. The tests generated may not be executable because the LLM is not guaranteed to generate compilable code. Nie et al. [99] report 29% of tests generated using TeCo are executable, while Yuan et al. [100] found that approximately one-quarter of the tests generated by ChatGPT were executable, rising to one-third with suitable prompt engineering.

Of those tests that do compile, several authors have reported on the code coverage achieved. For example, Bareiß et al. [101] reported an increase from the 10% achieved using Randoop [102] to 14% with CodeX. Hashtroudi et al. [103] reported a 50% increase in line coverage for the tests they generated by fune-tuning CodeT5. Siddiq et al. [104] reported 80% coverage on the HumanEval dataset using CodeX, but also found that neither the studied LLMs could achieve more than 2% coverage on the EvoSuite SF110 dataset.

Hybrid approaches that combine existing test generation and evaluation techniques, such as fuzz-based testing and search-based testing, with LLMs have already demonstrated promising results. For example, Lemieux et al. [105] introduced CODAMOSA, an algorithm that combines Search-Based Software Testing (SBST) [5] and CodeX to generate high-coverage test cases for programs under test. When SBST's coverage improvements stall, CODAMOSA asks CodeX to provide example test cases for under-covered functions. This helps SBST redirect its search to more useful areas of the search space. In an evaluation of 486 benchmarks, CODAMOSA achieved significantly higher coverage compared to SBST and LLM-only baselines. Hu et al. [106] introduced Chat-Fuzz, which augments the widely studied fuzzer, AFL, with ChatGPT, in order to get more format-conforming mutants. In an evaluation of 12 target programs chosen from three benchmarks, ChatFuzz achieved higher branch coverage than AFL by 13%. Dakhel et al. [107] used mutation testing to help LLMs to generate tests. In particular, they augmented prompts for Codex and Llama-2-chat with surviving mutants. They report that their approach detects 28% more human-written faults. Xia et al. [108] recently demonstrate that LLMs can serve as a universal fuzzer for systems across different application domains and programming languages, including C/C++ compilers, JDK, SMT solvers, and even quantum computing systems.

Deng et al. [109] propose TitanFuzz, which uses LLMs (i.e., Codex) to generate valid input DL programs to test DL libraries. The results on PyTorch and TensorFlow reveal that TitanFuzz can achieve 30%/51% higher code coverage than state-of-the-art fuzzers. Later on, they further introduced FuzzGPT [110], which synthesizes unusual programs for fuzzing DL libraries. Their results indicated that CodeX and CodeGen could outperform TitanFuzz on PyTorch and TensorFlow when re-targeted for fuzz-based testing.

Li et al. [111] used a hybrid of differential testing and ChatGPT to elevate the latter's ability to generate failure-inducing test cases of buggy programs. They report a test effectiveness improvement from 29% to 78%.

A promising area for LLM-based test generation is GUI testing, because the manipulation of the application state via GUI often requires a semantic understanding of both the user interface as well as the application domain. Sun et al. [112] described user interface via text, and asked ChatGPT which action it would like to perform next based on the text, then convert the answer into actual GUI interaction. This resulted in 32% higher activity coverage compared to the state-of-the-art.

One particularly important problem that is challenging for classical techniques is the construction of test cases from user reports. The user reports are written in natural language. This has presented considerable challenges for existing techniques, but is ideally suited to LLMs. Kang et al. [113] introduced Libro, a few-shot learning failure reproduction technique that automatically generates tests from general bug reports, based on CodeX. Libro successfully reproduced approximately one third of the failures.

Feng and Chen [114] demonstrated a replicability rate of 80% on bug reports with natural-language-defined steps to reproduce, using an LLM out of the box (ChatGPT) with Chain of Thought prompt engineering alone.

Several authors have considered prompt engineering to improve the results of test generation [115], [116]. For example, Schafer et al. [116] proposed TESTPILOT, which re-prompts with failing tests and associated error messages, achieving reported average statement coverage of 68%. Xie et al. [117] create prompts for test generation by parsing the project and creating an adaptive focal context that includes the focal method and its dependencies. They further used rule-based repair to fix syntactic and simple compile errors in the tests.

Although the outcomes of LLM-based testing may be uncertain, researchers have explored cross reference or majority of votes [118], [119] methods to estimate the confidence of LLMs, based on the notion of 'self-consistency' [120]. For example, the Libro introduced by Kang et al. [113] uses CodeX to generate tests from bug reports that can reproduce failures. If multiple tests show similar failure behavior, Libro estimates that LLM is "confident" in its predictions. Furthermore, where there is partial oracle information, this can also be used to augment confidence estimates. Such partial oracle information is often available when the goal of the overall processes to improve on existing code. For example, when improving the efficiency of an existing test, automated partial oracle information can be gathered from observing whether the test behaves similarly to the original (passing and failing in the same situations), and is also faster to execute.

B. Test Adequacy Evaluation

Test effectiveness is typically measured in terms of 'adequacy criteria' [121], [122]. Since testing cannot exhaustively explore every possibility, adequacy criteria provide a form of lower bound on the effectiveness achieved by a suite of tests. Mutation testing is a widely-studied technique for assessing the adequacy of software test suites [123], [124], in which synthetic faults (called 'mutants'), are deliberately injected in order to assess test adequacy. Mutation testing has been shown to provide more stringent adequacy criteria than other structural coverage-based criteria such as statement and branch coverage [125].

One of the challenging open problems for mutation testing is to generate mutants that faithfully model important classes of real-world faults. Khanfir et al. [126] used CodeBert to generate developer-like mutants and found that their approach has better fault revelation ability than PiTest. Garg et al. [127] applied CodeBERT to generate mutants that faithfully capture vulnerabilities. They evaluation found that 17% of the mutants fail the tests that are failed by 89% of the respective vulnerabilities. Brownlee [128] used GPT-3.5 to generate mutants for genetic improvemnt and observed that randomly sampled LLM-based edits compiled and passed unit tests more often compared to standard GI edits.

C. Test Minimisation

Test minimisation improves the efficiency of software testing by removing redundant test cases. Pan et al. [129] applied CodeBERT, GraphCodeBERT, and UniXcoder to extract embeddings of test code to conduct test minimisation. Their approach achieves a 0.84 fault detection rate and runs much faster (26.73 minutes on average) than the baseline.

D. Test Output Prediction

Liu et al. [130] proposed CodeExecutor, a pre-trained Transformer model, to predict the program's whole execution trace. The purpose is to imitate the real-world arbitrary program execution behaviour. Their evaluation compares CodeExecutor with CodeX, and shows that CodeExecutor significantly outperforms Codex in execution trace prediction (e.g., 76% vs. 13% output accuracy for the Tutorial dataset).

E. Test Flakiness

A test is flaky if it can pass on some occasions and fail on others without any apparent (tester-controllable) change in the execution context. Test flakiness is one of the most pressing and impactful problems that inhibit test effectiveness in industry [131]. LLMs have been used to predict flakiness with high accuracy (with 73% F1 score [132], [133] and 97% accuracy [134] reported).

F. Open problems in LLMs for Software Testing

There are many open problems in LLM-based software test data generation, most of which lie well within the grasp of existing software testing techniques. We can thus expect an exciting explosion in LLM-based software test generation in the coming years. This section outlines some directions for this research agenda.

1) Prompt Engineering: There are many aspects of a good software test that could be favoured by suitable prompt engineering. For example, we need to understand how to engineer prompts that

- Predict and reduce generated test flakiness;
- Reveal likely faults, for example via training on historic fault data;
- Optimise the balance between mocking and integration testing;
- Make realistic data builders, mock objects, parameters and inputs;
- Predict tests that are most likely to elicit tests that cover corner cases;
- Tailor test generation to focus behaviour that is prevalent in production.

2) Augmenting Existing Tests: Work on LLM-based test generation has focused on the automated generation of novel test suites. However, given the array of existing test generation techniques, there remains an important (and comparatively less well-studied) open problem of augmentation and regeneration based on existing test suites [135], [136].

Test augmentation and regeneration can exploit few-shot learning and/or can fine-tune (on an existing suite of test data and historical faults), to generate augmented test suites.

More work is needed on LLMs for generating additional test assertions that capture corner cases, historical faults, and likely programmer errors, drawing on the training data available. Hybridization between LLMs and existing automated test generation techniques is also a productive theme [105].

3) Test Correctness: Traditional software test generation has suffered from the Oracle Problem [6], i.e., they are inhibited by the lack of an automated oracle that determines whether a test outcome is correct. Two cases pertain to AI-generated tests:

1) **The generated test passes on the current release**: We might assume that the functionality is correctly tested and that the generated test thus acts as a regression test, against which future changes can be checked.

2) **The generated test fails on the current release**: We need to know whether the assertion is wrong or whether the generated test has found a bug.

Both cases can have pernicious consequences when they are not imbued with self-regulation. A test case that passes may merely reflect coincidental correctness [137], [138]. Worse, it might be the case that the code is, in fact, *incorrect* (and that the test is equally incorrect yet captures, and thereby enforces, the incorrect behaviour). In such cases, the generation of the test will tend to *inhibit* fault remediation, by failing on future fixes. This problem also affects LLM-generated test cases, and may be more pernicious in cases where such tests hallucinate oracle properties, baking into the generated tests these incorrect oracle assertions.

On the other hand, when a generated test case fails, this may indicate a bug. This bug revelation would denote a 'win' for LLM-based testing. However, should it turn out that the ratio of false positives to true positives are high, then the cost (e.g., in human assessment) may make the technique impractical, even when it does reveal true positive bugs [131]. More work is needed on self-assessment of confidence, self-checking for correctness, consistency, and robustness of generated tests. We need to develop techniques for automatically assessing, augmenting and filtering raw outcomes from execution of LLM-based tests, before presenting the 'test signal' to the developer.

The interaction between LLM hallucination and test correctness is an important topic in its own right. Since LLM-based code generation is generally driven by what is most likely, rather than what is most correct, hallucination poses threats to any questions of correctness. However, interestingly, Feldt et al. [139] reported a case of hallucination being *helpful* for software testing, because it may reveal discrepancies between the actual program semantics and the programmer's perception of the semantics. They suggested a form of conversational testing agents (i.e., any generated tests are filtered by the programmer via the conversation) to harness this capability without posing any threats to overall test correctness.

More work is also required on the scientific foundations on which evaluations of LLM-based software testing rest. More care and attention are clearly needed to heed the 'best practice' advice for the scientific analysis and reporting from previous work on foundations of Empirical and Search Based Software Engineering [11], [13], [14].

4) Mutation Testing: More work is needed to explore the adequacy achievable with LLM-based test generation, and also to use LLM-based techniques to support and enhance test adequacy investigation and assessment. LLMs can be fine-tuned on a fault model, and thereby used to suggest mutants that are highly coupled to real faults, and can thus be used to assess test adequacy.

VI. Maintenance, Evolution and Deployment

Software maintenance and evolution have been important topics of study for many decades. They are concerned with existing code bases from which we seek understanding and business logic extraction, and for which we seek to re-engineer, repair and refactor. Maintenance problems, such as these, all reside within language-rich problem domains. It is therefore unsurprising that this area finds many applications of LLM-based techniques, as we review in this section.

A. Debugging

Kang et al. [140] studied GPT-3.5's fault localisation ability, and found that LLM could often identify the faulty method on the first try. Wu et al. [141] present a comprehensive investigation into the capability of GPT-3.5 and GPT-4 for fault localisation accuracy, stability, and explainability. The results demonstrate that GPT-4 achieves 47% higher fault localisation accuracy over the state-of-the-art, but the performance declines dramatically with a longer code context.

Feng and Chen [142] proposed AdbGPT, which reproduces Android bugs from bug reports through prompt engineering with ChatGPT. With a dataset of 88 bug reports, AdbGPT was able to successfully reproduce 81%, outperforming the baselines and ablations. Joshi et al. [143] focused on multilingual debugging and proposed RING, which proposes a prompt-based strategy that conceptualizes repair as localization, transformation, and candidate ranking.

To address the data leakeage threat in fault localisation and program repair, Wu et al. [144] introduced ConDefects with 1,254 Java bugs and 1,625 Python bugs that were produced between October 2021 and September 2023. Researchers are allowed to select code samples based on their creation period, thereby allowing them to evaluate the effectiveness of different LLMs according to their training data cut-off date. In addition, there has been work on predicting bug severity with LLMs [145].

B. Program Repair

Repairing bugs has been a topic of much interest for over a decade in the software engineering research community [146], [147], and has already found its way into initial industrial deployment [148].

Much of the work on automated repair has used the generate-and-test approach widely adopted in the field of Genetic Improvement and readily applicable to LLM-based techniques. As a result, LLMs are certain to have a positive impact on automated software repair, but there remain technical challenges in taming the hallucination problem and managing scalability, as we report in this section.

In order to achieve scalability, all generate-and-test approaches need to address the build time problem [149]. LLM-based repair is no exception; the propensity to hallucinate makes it all the more important that the test phase can be executed regularly. It is likely that using ReAct deployment models [95] will help to find efficient and effective engineering trade-offs. When ReAct is applied to repair, the overall approach would alternate between the 'Reason' phase (generating candidate fixes) and the 'Action' phase (evaluating fixes through testing, which involves the build problem).

To address this issue, we can refer to the well-established literature on software repair [46], [150], grounded in over two decades of the development of search-based approaches to software engineering [12], [151]. This literature provides the research community with a firm foundation of experience and expertise, making it very well-placed to develop LLM-based generate-and-test approaches to the problem.

Recent work on repair has started to use neural AI models, such as the seminal work of Tufano et al. [152]. More recently, since 2022, there has been a rapid development of emergent embryonic research literature on LLM-based repair. For example, Xia et al. [153] proposed AlphaRepair. It redefines the APR problem as a cloze (or infilling) task, where the LLMs are leveraged to directly fill-in correct code based on the bi-directional context of the potential buggy code portions. AlphaRepair also demonstrates for the first time that LLMs can outperform all prior APR techniques.

They further conducted an empirical study [154] on nine LLMs across five datasets in three different languages. Their findings not only affirmed the superiority of LLM-based APR (especially the cloze-style approach) but also offered a number of practical guidelines. Wei et al. [155] synthesize a patch through the interaction between an LLM and a Completion Engine, and found that the approach surpasses the best-performing baseline by 14 and 16 bugs fixed.

Program repair naturally fits a conversational model of prompt engineering. Xia et al. [156] proposed conversational APR, which alternates between patch generation and validation in a conversational manner. Their evaluation on ten LLMs demonstrated that their approach has superiority in both effectiveness and efficiency.

They further proposed ChatRepair [157], showing that the conversational approach fixes 162 out of 337 bugs for only $0.42 per bug, thereby also addressing potential concerns about the computational resources required. Chen et al. [158] introduced SELF-DEBUGGING, which teaches an LLM to debug its predicted code via few-shot learning, SELF-DEBUGGING reports baseline accuracy improvements of up to 12%.

Studies have also reported results for particular classes of bugs, for example, Pearce et al. [159] reported repair results from five commercial LLMs on security bugs, Charalambous et al. [160] combined ChatGPT with with formal verification strategies to verify and automatically repair software vulnerabilities. Cao et al. [161] report ChatGPT results for Deep Learning (DL) program repair.

Repair does not always start with an existing failing test case, but can start with a natural language description of a failure in production. Automation opens the door to faster responses to user-generated bug reports. This is a route to repair that has also been explored for LLMs in the work of Fakhoury et al. [162], who generated functionally correct code edits from natural language issue descriptions. They propose Defects4J-Nl2fix, a dataset of 283 Java programs from the Defects4J dataset with high-level descriptions of bug fixes. The state-of-the-art LLMs evaluated on this benchmark achieve up to 21% Top-1 and 36% Top-5 accuracy.

Automated repair can also reduce the burden on engineers, managing DevOps-style on-call for production systems. For example, Ahmed et al. [163] studied the use of LLM-based root causing and remediation of 40,000 incidents on Microsoft cloud services. The authors evaluated multiple LLMs using semantic and lexical metrics in zero-shot, fine-tuned, and multitask settings, showing that fine-tuning significantly improves incident response effectiveness.

The ability to perform fine-tuning for a specific task or domain can significantly improve the model performance in program repair. Jiang et al. [164] empirically evaluated the performance of 10 different Code Language Models (CLMs) and 4 fault benchmarks, and showed that repair-specific fine-tuning could significantly improve success rates. On average, the 10 CLMs already successfully repaired 72% more faults than state-of-the-art DL-based APR techniques. After fine-tuning, the number increased to 160%. Jin et al. [165] proposed InferFix, which contains a LLM (Codex Cushman) finetuned on supervised bug-fix data. InferFix achieves a 76% Top-1 repair accuracy on Java, and over 65% on C# using the InferredBugs dataset. Berabi et al. [166] introduced TFix, a T5 model fine-tuned on bug-fixing data, reporting that it outperformed existing learning-based approaches. Xia et al. [167] combines LLM fine-tuning and prompting to automate the plastic surgery hypothesis and demonstrated that their approach fixes 89 and 44 bugs (outperforming the baseline by 15 and 8).

LLMs can also help to explain the patches that they generate. Kang et al. [168] proposed AutoSD to provide debugging explanation with LLMs to help developers judge the correctness of patches. They found that AutoSD produced comparable results to existing baselines with high-quality repair explanations. Sobania [169] studied the capability of GPT 3.5 in explaining the patches generated a search-based repair tool, ARJA-e, on 30 bugs from Defects4J. 84% of the LLM explanations are found to be correct.

C. Performance Improvement

Since the inception of computer programming, the paramount importance of performance optimisation has been recognised. Indeed, performance optimisation is even mentioned by Ada Lovelace in her nineteenth-century notes on the analytical engine [170]. Much initial practical deployment of optimisation took place in compiler development, through work on optimising compilers [171]. This is the bedrock on which current practical and efficient computation rests, but it is necessarily a one-size-fits-all approach; widely applicable due to its generality, yet suboptimal for bespoke problem domains for the same reason. There has, therefore, also been much work on specific source-to-source transformations to improve optimisation, dating back to the 1970s [172], [173].

For a long time, the focus of this work was on finding suitable sets of meaning-preserving transformations, the motivation being that a correct program can be transformed into a more efficient version of itself, while retaining its correctness. However, more recently, research on program synthesis took a different turn: Inspired by Genetic Programming [174], and early results from Automated Program Repair [146], [175], it considered a wider set of transformations in an approach that has come to be known as 'Genetic Improvement' [8], [176].

The wider set of transformations may produce incorrect code, but automated testing can filter these, to ensure sufficient faithfulness to the intended semantics. Furthermore, the freedom to treat existing code as a kind of 'genetic material' produced dramatic improvements in non-functional properties, such as execution time, memory and power consumption (e.g., 70x speed up of a non-trivial gene sequencing system [177]).

Although the potential for artificial intelligence techniques, such as evolutionary algorithms, to improve performance has been well studied, researchers have only just begun to consider the potential for LLM-based performance improvement. In the work by Madaan et al. [178], the authors use CODEGEN and CodeX to suggest functionally correct, Performance-Improving Edits (PIEs), improving execution time of Python and C++ (already pre-optimised with the maximally optimising compiler option -O3). Similarly, Garg et al. [179] proposed DeepDev-PERF, a performance improvement suggestion approach for C# applications that. DeepDev-PERF took the English-pretrained BART-large model and further pretrained it on Source code. Kang and Yoo [180] proposed the use of LLMs to suggest objective-specific mutation operators for genetic improvement, and provided demonstrations on improving efficiency and decreasing memory consumption. Garg et al. [181] proposed RAPGen, which generates zero-shot prompts for LLMs to improve performance. The prompts are generated via retrieving a prompt instruction from a pre-constructed knowledge base of previous performance improvements. Chen et al. [182] used GPT models as baselines for their source code optimisation method, Supersonic, and found that Supersonic improves running time for 26.0% of the programs, compared to only 12.0% for GPT-3.5-Turbo and 4.0% for GPT-4.

Cummins et al. [183] focused on the performance of compilers and presented results on LLMs for optimising compiler instructions. Their results demonstrate that a relatively small (7B-parameter) LLM, trained to generate instruction counts and optimized compiler LLVM code, can generate 3% improvements in reducing compiler instruction counts, outperforming the state-of-the-art. Their results are also promising in terms of correctness, with 91% compilable and 70% functionally correct wrt the original compiler output.
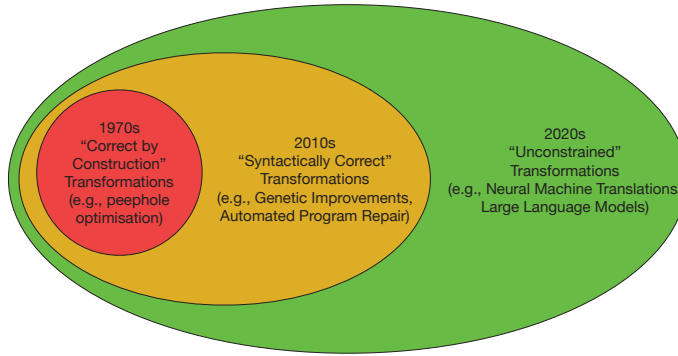

Fig. 5. The Widening Scope of Program Transformation

Over a period of some 50 years, the software engineering community has evolved its conception of what it means to transform an existing software system into an equivalent system that improves performance while retaining functional behaviour. In the 1970s, the strongest concern was correctness, so transformation palettes were defined to consist solely of transformation steps that were (functionally) correct by construction.

However, by 2010 the community was already exploring the application of considerably more relaxed notions of equivalence that merely retain sufficient operational faithfulness to the behaviour of the original. The tight semantic straitjacket of the 1970s was thereby considerably relaxed to allow transformations that might even *fail* some test cases. During the same period, operational performance became increasingly important. A key underlying principle of this research agenda is that no overall software system can be deemed functionally correct, when it is executed on a system in which inefficiency has left insufficient remaining resources. This principle applies even in the (comparatively rare) cases where the software has been fully proven to be functionally correct. As the more pithy slogan has it:

"There is nothing correct about a flat battery" [8].

This evolution of the community's approach to code transformation and synthesis is depicted in Figure 5 (red and yellow regions).

In the context of this increasing relaxation of semantic constraints, we can view LLM-based code optimisation as a further development of this overall direction of travel: Code optimised by LLMs may not be even *syntactically* correct, let alone semantically correct (depicted by the green region of Figure 5).

Despite these correctness challenges, inherent in LLM-Based SE, there is a large pool of training data, and LLMs have a propensity to exhibit emergent behaviour. These observations combine to yield surprising results that, although not guaranteed to be correct, can potentially dramatically change performance characteristics in useful ways.

Of course, as we increasingly allow more permissive transformation pallets in the hope of optimising multiple non-functional properties, we simultaneously place far greater reliance upon the ability of testing to provide reassurance of functional faithfulness. Testing is also vital to check for regressions in those non-functional properties that are not targeted by the improvement process. As a result, software testing in general (and automated high coverage test generation in particular), will become ever more important.

D. Clone Detection and Re-use

There has been much previous work on managed software reuse [184] in order to extract value and avoid duplication, a topic also tackled using LLMs [185]. Software typically contains large numbers of clones, arising from ad hoc re-use, resulting in much work on automated clone detection [186], a topic for which fuzz-based fine-tuned LLMs have also been applied [187].

E. Refactoring

When we refactor code, we generally expect its behaviour to remain unchanged. This is particularly attractive for automated approaches (such as search-based refactoring [188]) because it means that we can simply rely on the Automated Regression Oracle. This 'automatable oracle for free' advantage is significant and will also apply to LLM-based refactoring.

Poldrack et al. [75] show that GPT-4 refactoring of existing code can significantly improve code quality according to long-established structural metrics such as Halstead [189] and McCabe [190] complexity. Noever and Williams [83] emphasize the value of AI-driven code assistants in refactoring legacy code and simplifying the explanation or functionality of high-value repositories.

F. Open Problems in Maintenance and Evolution

Since so many of the subdomains of software maintenance and evolution are concerned with existing legacy system source code, we can expect rapid growth in the application of LLMs. This section outlines some existing open problems in this nascent sub-area of research.

1) Open Problems in Performance Improvement: Much more work is needed on the development of LLM-based techniques for automatically finding performance improvements. As with Genetic Improvement, these need not be confined merely to execution time, but can also consider other non-functional attributes such as power consumption [191]–[193] and memory footprint [194] as well as multi-objective, trade-offs between sets of non-functional properties [195]. We expect more work on Genetic Improvement-style LLM-based code optimisation techniques, with the potential for many dramatic advances and breakthroughs.

44

2) Open Problems in Refactoring: By definition, refactoring does not change semantics, so LLM-based refactoring can rely on the Automated Regression Oracle. It is therefore surprising that there is not already more work on LLM-based refactoring. In this subsection, we outline possible directions.

Design patterns have played a critical role in practical software engineering for three decades [196]. LLMs may help engineers to refactor existing code to use design patterns, while providing developer-friendly explanations and documentation.

Refactoring also becomes necessary whenever new technologies emerge. For example, when an API is updated or a new API becomes available. Although they can be (sometimes automatically [197]) repaired, API misuse remains a common source of software engineering bugs. Automating the process of refactoring for new APIs is less challenging than other code transformations, because of the presence of the Automated Regression Oracle.

Finally, the few-shot learning capabilities of LLMs may enable more bespoke refactoring. The emergent work on LLM-based refactoring has focused on global refactoring according to well-known refactoring patterns. However, programmers often have project-specific refactoring requirements. Up to a third of software engineering effort is spent on largely repetitive, tedious, and potentially error-prone refactoring activities that implement these project-specific refactoring needs. The few-shot learning potential of LLMs may automatically generalise from specific examples, automating what we call 'bespoke' refactoring. More work is needed to develop techniques for reliable few-shot-learnt bespoke refactorings.

VII. DOCUMENTATION GENERATION

Most of the work on LLM-based software engineering has focused on the generation of code, but there is also considerable potential for LLM-based documentation generation.

Sun et al. [198] explored how ChatGPT performs on code summarisation of Python code. They used CSN-Python and compared ChatGPT with NCS, CodeBERT, and CodeT5. They adopted three widely-used metrics: BLEU, METEOR, and ROUGE-L. Surprisingly, the results show that ChatGPT's performance is significantly worse than the baseline models in terms of BLEU and ROUGE-L.

Ahmed et al. [66] conducted prompt engineering for code summarisation on GPT-3.5, while Geng et al. [199] performed experiments on two Java language datasets, Funcom and TLC, using Codex: to generate *multiple-intent* comments. Gent et al. [200] demonstrate that pre-trained LLMs already have sufficient context to generate multiple different code summaries from different technical perspectives.

A. Open Problems in Documentation Generation and Code Summarization

Many existing code summarization techniques are retrieval-based: the given code is represented in a vector format using a neural representation, which is subsequently used to retrieve the most relevant textual summarization from the corpus.

There is a clear limitation to this approach due to the fact that the set of summaries that can be generated are constrained by the training corpus. LLMs may enable automated code summarization that is not restricted to this training corpus, assisted by their natural language processing capabilities.

While this may result in richer and more semantically relevant summaries, we also note that existing evaluation metrics are often lexical in nature, hindering our ability to compare and evaluate richer summaries generated by LLMs [198]. Recent advances in ReAct-based approaches [95] may open up other avenues for greater assurance in the documentation generated, even when it cannot be executed.

VIII. SOFTWARE ANALYTICS AND REPOSITORY MINING

There is a well-established field of software analytics; how to yield insight for human engineers from existing software artefacts [201]. The large amount of software artefact information publicly available online has stimulated the growth of scientific insights gained by Mining Software Repositories (MSR) [202], [203]. While MSR tends to focus on scientific research insights from such mining, software analytics tends to focus on opportunities for organisations to gain insight from the analysis of their own repositories, which can also benefit AI understandability [204].

Hitherto, in both cases, much of the collection, curation and analysis of data has relied upon labour-intensive human analysis. We found no work on the use of LLMs to support this activity. Nevertheless, because many LLMs have already ingested this software artefact data, and are capable of providing reasoning and insight, it seems natural to expect them to play a significant role.

For example, LLMs may identify interesting new MSR research questions, based on their ability to ingest large amounts of data, including research questions and hypotheses that have previously proved interesting to researchers. They may also assist with traceability, which software engineers have great difficulty maintaining [205], [206].

IX. HUMAN COMPUTER INTERACTION

Finding productive interfaces between human engineers and software infrastructure has remained a recurring theme throughout the lifetime of the development of software engineering [207], [208], dating back to the inception of the discipline in the 1960s [209].

We found evidence of many interesting research questions. For example, Vaithilingam et al. [210] reported on the difficulties 24 participants had in understanding, editing, and debugging the Copilot-generated code, while Feldt et al. [139] proposed a hierarchy of design architecture for LLM-based software testing agents. Liang et al. [36] surveyed 410 practising software engineers, finding widespread use of LLMs to facilitate low-level programming tasks, but also resistance to using LLMs for more design-level software engineering activities. Feng et al. [211] collected 316K tweets and 3.2K Reddit posts about ChatGPT's code generation to understand social media's attitudes toward AI-assisted coding tools.

They found that fear is the dominant emotion associated with ChatGPT's code generation, overshadowing other emotions such as happiness and surprise. Ahmad et al. [212] explore the way in which a novice software architect could interact with ChatGPT.

X. SOFTWARE ENGINEERING PROCESS

Software engineering concerns, not only software products, but also the process by which they are constructed [213]. Previous research on software assistants [207], [214]–[217] is clearly of particular relevance to LLM-based software engineering, a topic some authors have already started to consider. For example, Ross et al. [218], introduced an LLM-based programmers' assistant, evaluating its deployment with 42 participants while Tian et al. [219] highlighted the attention span limitations of ChatGPT.

XI. SOFTWARE ENGINEERING EDUCATION

Teachers have expressed concern at the difficulties of identifying cases where students have relied on LLMs to construct their assignments [220], while other authors have argued that the long-term impact of LLMs on education will be beneficial [221]. However, our present focus rests more narrowly on the specific impact of LLMs on the field of *software engineering* education, where the current literature focuses on LLM-based tutorial support.

For example, Jalil et al. [222] explored opportunities for (and issues with) ChatGPT in software testing education. Savelka et al. [223] analysed the effectiveness of three models in answering multiple-choice questions from introductory and intermediate programming courses at the postsecondary level. Several other authors [82], [83], [224] explored the capabilities of CodeX for generating programming exercises and code explanations. Their general finding was that the majority of the generated content was novel, sensible, and useful (see also Section IV-D3).

XII. CROSSCUTTING OPEN RESEARCH TOPICS

A number of patterns emerge from the embryonic literature on LLM-based software engineering. In this section, we outline those that raise open research questions that cut across all software engineering applications

A. Building and Tuning LLMs for SE

Most of the previous work has treated LLMs as atomic components, with a focus on incorporating these in wider software engineering workflows. While there have been attempts to tailor the behaviour, these have tended to focus on prompt engineering, with a few examples of fine-tuning.

A more challenging but potentially impactful problem lies in training and fine-tuning models, specifically for software engineering tasks. Ding et al. [225] train a BERT-like LLM with execution inputs and dynamic execution traces. They show how this dynamic information improves (up to 25%) the accuracy of the model for downstream software engineering predictive tasks: vulnerability and clone detection and coverage prediction (full execution path and branch coverage).

More work is needed on new forms of LLMs, specifically tailored for software engineering that take advantage of software's unique properties and distinguish it from natural language. Dynamic information is one such key differentiator currently missing from most of the work. We expect the next generation of SE-specific LLMs to address this.

An important aspect of building and training LLMs is their energy consumption. LLM capabilities have been associated with their size [226], resulting in rapid growth of model size [227], [228]. The training and developing of larger models may have direct environmental impact [229]. While it has been suggested that the model performance depends not only on model size but also on the volume of training data [230], the question of the right model size required to achieve the desired performance remains unclear.

Lighter models may also widen adoption, thereby leading to enhanced deployability. Recently, techniques such as low-rank adaptation (lora) [231] and model quantization [232] have shown potential, but they remain to be empirically evaluated with respect to specific applications.

B. The Need for Dynamic Adaptive Prompt Engineering and Parameter Tuning

Initial work on prompt engineering has demonstrated its potential to considerably improve the software engineering artefacts generated by LLMs. However, as already found [58], the results are highly problem-specific, so a one-size-fits-all approach is unrealistic. Furthermore, very few papers report model parameter settings, yet we know that many of these, such as the temperature setting, play a crucial role in determining the nature of the generated LLM output.

As an immediate starting point, it is imperative that authors make a point of conspicuously reporting these parameter settings to support replication. However, we also need more research on dynamic adaptive prompt engineering and model parameter tuning. This research agenda may draw inspiration from existing work on parameter tuning for other dynamic adaptive tasks, such as fuzzing [233]. Dynamic prompt optimisation may also exploit techniques associated with SBSE [12], reformulating prompt optimisation as a multi-objective computational search process.

C. Hybridisation

LLMs are seldom most effective when used in isolation, but can be highly effective as part of an overall SE process. More work is needed to understand the design patterns for SE workflows into which LLMs can safely, efficiently and effectively reside. We believe that existing SE theory and practice associated with generate-and-test approaches, such as Automated Repair and Genetic Improvement, are already highly amenable to LLMs.

We expect to see much more work incorporating LLMs into these existing software engineering frameworks. However, more work is required to tailor and extend these frameworks, to best take advantage of the opportunities offered by LLM-based software engineering.

In particular, we expect to see a rapid development of work on static and dynamic analyses for prompt engineering and post-processing of LLM responses. We also expect to see hybrid software engineering processes, adapting Continuous Integration pipelines to incorporate LLMs.

D. Harnessing Hallucination

While hallucination has widely been regarded as a problem, as reported in this survey, it may also prove to provide benefits when applied to software engineering domains. LLM hallucinations are seldom entirely *random* incorrect responses. Rather, because of their inherent statistical properties, they would be better characterised as 'plausible futures', and this may often make them useful when set in the right context.

Hallucination can be repurposed to provide potentially *useful suggestions* for software enhancement. For example, when hallucinating a test case, the LLM may be repurposed to suggest new features, while a hallucinated code summarisation might indicate potential for (human) code misunderstanding; if the LLM 'misunderstood' the code, might not a human also misunderstand it? When the LLM hallucinates an non-existent API, it may be repurposed as a way to *suggest* refactoring to simplify or extend existing APIs. More work is needed to exploit this positive potential, and to harness hallucination for software improvement.

E. Robust, Reliable, and Stable Evaluation

Hort et al. [234] conducted a review of 293 papers on LLMs for code generation, to determine the degree to which sufficient information was shared to support replication. They found that only 33% shared source code and 27% shared trained artefacts. They also evaluated the papers from the perspective of energy consumption, assessing the degree to which it was possible for an independent researcher to assess the energy consumed during training. They report that approximately 38% (30 out of 79 publications which involve model training) shared sufficient information to estimate their energy consumption during training.

Further evidence that there may be a growing issue with scientific evaluation quality in the literature on LLM-Based Software Engineering can be found in the survey of LLM-Based Testing by Wang et al. [98]. In their survey, they filtered an initial pool of papers on LLM-Based Testing to remove those that did not meet standard evaluation quality constraints. These constraints required papers to include a clear, defined, repeatable evaluation methodology that includes a control or baseline against which to measure effectiveness. This filtration criterion removed more than 90% of the papers that initially met keyword search criteria.

As these analyses of the literature demonstrate, more work is clearly needed to establish firm scientific foundations for the emerging discipline of LLM-based Software Engineering. Such foundations may draw on existing foundations for Empirical Software Engineering in general and, more specifically, on AI-based Software Engineering, such as SBSE (where there is a natural similarity [105], [235]).

Nevertheless, LLMs have their own unique properties, such as the ability to provide explanations, which will require domain-specific theoretical and empirical scientific foundations.

LLMs inherently exhibit non-deterministic behaviour. Researchers need to carefully design their experiments, configure their LLMs (e.g., evaluating the effects of different distribution sampling strategies), and take into account non-determinism when drawing their conclusions on LLMs. The SBSE literature provides advice on the inferential statistics required to support such evaluation [13], [14].

We will witness a rapid growth in the number and diversity of language models for software engineers in the coming years. Both practitioners and practising software engineers will need reliable, efficient and comprehensive benchmarking systems. Benchmarking platforms such as TESTPILOT [116] and platforms such as Papers With Code (https://paperswithcode.com/sota/code-generation-on-humaneval/) will become increasingly important.

As well as generic scientific foundations, benchmarks and evaluation platforms, we also expect to see longitudinal studies of developer behaviour when programming with LLM assistance, so that we can understand the programmer-LLM interaction better and design more effective use case scenarios.

F. Thorough Testing

The problem of hallucination has already been widely studied. It will continue to be a topic of great interest, both within the software engineering community and in the wider computer science community. While it is likely great progress will be made, the inherent risk of hallucination is unlikely to be completely eradicated, since it is as germane to the LLM technology, as it is to human intelligence. Fortunately, over more than six decades, software engineers have developed robust automated verification and testing technologies that help to reduce the impact of human mistakes. We expect that such technologies will also carry over to artificial intelligence mistakes.

G. Handling Longer Textual Inputs

The performance of LLMs on large-sized input prompts is likely to be a topic of great interest in the artificial intelligence community [236]. Advances in this area will have a strong impact on software engineering, because of the considerable size of software systems and the consequent opportunities that additionally open when larger prompts are to be effectively handled.

H. Less Well-covered Subdomains of Software Engineering

As our survey reveals, some subdomains of software engineering are notably under-represented in the literature; some surprisingly so. For example, Requirements Engineering and Design (Section III), and Refactoring (Section VI-E) enjoy very little coverage, yet they are surely ripe for consideration, since they rely heavily upon linguistic forms of analysis and the recognition and prediction of patterns.

References

[1] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, B. Yin, and X. Hu, "Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond," Apr. 2023, arXiv:2304.13712.

[2] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu, "The scope of ChatGPT in software engineering: A thorough investigation," 2023, arXiv:2305.12138.

[3] S. Anand, A. Bertolino, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, J. Li, P. McMinn, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, August 2013.

[4] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[5] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing (keynote paper)," in 8th *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, Graz, Austria, April 2015.

[6] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[7] J. Shin, H. Hemmati, M. Wei, and S. Wang, "Assessing evaluation metrics for neural test oracle generation," *arXiv preprint arXiv:2310.07856*, 2023.

[8] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, Jun. 2018.

[9] A. Cantino, "Prompt engineering tips and tricks with GPT-3," 2021. [Online]. Available: https://blog.andrewcantino.com/blog/2021/04/21/prompt-engineering-tips-and-tricks/

[10] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv prnote arXiv:2308.02828*, 2023.

[11] S. Easterbrook, "Empirical research methods for software engineering," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 574.

[12] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 11:1–11:61, November 2012.

[13] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in 33rd *International Conference on Software Engineering (ICSE'11)*. New York, NY, USA: ACM, 2011, pp. 1–10.

[14] M. Harman, P. McMinn, J. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification: LASER 2009-2010*, B. Meyer and M. Nordio, Eds. Springer, 2012, pp. 1–59, LNCS 7007.

[15] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv prnote arXiv:2308.10620*, 2023.

[16] T. Goyal, J. J. Li, and G. Durrett, "News summarization and evaluation in the era of gpt-3," 2023, arXiv:2209.12356.

[17] R. Nakano, J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders, X. Jiang, K. Cobbe, T. Eloundou, G. Krueger, K. Button, M. Knight, B. Chess, and J. Schulman, "Webgpt: Browser-assisted question-answering with human feedback," 2022, arXiv:2112.09332.

[18] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[19] Q. Xie, Z. Luo, B. Wang, and S. Ananiadou, "A survey for biomedical text summarization: From pre-trained to large language models," 2023, arXiv:2304.08763.

[20] K. Kheiri and H. Karimi, "Sentimentgpt: Exploiting gpt for advanced sentiment analysis and its departure from current machine learning," 2023, arXiv:2307.10234.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017, arXiv:1706.03762.

[24] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023, arXiv:2302.13971.

[25] S. Zhao, "Github Copilot now has a better AI model and new capabilities." [Online]. Available: https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/

[26] "GitHub CEO says Copilot will write 80% of code sooner than later," https://www.freethink.com/robots-ai/github-copilot, accessed: 2023-07-27.

[27] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-Level Code Generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, arXiv:2203.07814.

[28] OpenAI, "GPT-4 Technical report," 2023, arXiv:2303.08774.

[29] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of Artificial General Intelligence: Early experiments with GPT-4," Apr. 2023, arXiv:2303.12712.

[30] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2023, arXiv:2308.12950.

[31] C. J. Neill and P. A. Laplante, "Requirements engineering: the state of the practice," *IEEE software*, vol. 20, no. 6, pp. 40–45, 2003.

[32] Y. Zhang, A. Finkelstein, and M. Harman, "Search based requirements optimisation: Existing work and challenges," in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'08)*, vol. 5025. Montpellier, France: Springer LNCS, 2008, pp. 88–94.

[33] J. Zhang, Y. Chen, N. Niu, and C. Liu, "A Preliminary Evaluation of ChatGPT in Requirements Information Retrieval," Apr. 2023, arXiv:2304.12562.

[34] X. Luo, Y. Xue, Z. Xing, and J. Sun, "Prcbert: Prompt learning for requirement classification using bert-based pretrained language models," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[35] D. Luitel, S. Hassani, and M. Sabetzadeh, "Improving requirements completeness: Automated assistance through large language models," *arXiv prnote arXiv:2308.03784*, 2023.

[36] "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in 46th *International Conference on Software Engineering (ICSE 2024)*, April 2024, to appear.

[37] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 213–222.

[38] V. Murali, C. Maddila, I. Ahmad, M. Bolin, D. Cheng, N. Ghorbani, R. Fernandez, and N. Nagappan, "Codecompose: A large-scale industrial deployment of ai-assisted code authoring," 2023, arXiv:2305.12050.

[39] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2023, arXiv:2204.05999.

48

[40] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub Copilot," 2023, arXiv:2302.06590.

[41] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.

[42] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–164, 1971.

[43] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[44] A. Hindle, E. Barr, Z. Su, P. Devanbu, and M. Gabel, "On the naturalness of software," in *International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012.

[45] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in 22nd *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, Hong Kong, China, November 2014, pp. 306–317.

[46] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[47] M. Gabel and Z. Su, "A study of the uniqueness of source code," in 18th *ACM SIGSOFT international symposium on foundations of software engineering (FSE 2010)*. Santa Fe, New Mexico, USA: ACM, 7-11 Nov. 2010, pp. 147–156.

[48] J. Darlington and R. M. Burstall, "A transformation system for developing recursive programs," *J. ACM*, vol. 24, no. 1, pp. 44–67, 1977.

[49] Mark Chen et al., "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374.

[50] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2022, accepted at ICLR 2023.

[51] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," 2023, arXiv:2305.02309.

[52] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. San Diego CA USA: ACM, Jun. 2022, pp. 1–10.

[53] E. Jiang, E. Toh, A. Molina, K. Olson, C. Kayacik, A. Donsbach, C. J. Cai, and M. Terry, "Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models," in *CHI Conference on Human Factors in Computing Systems*. New Orleans LA USA: ACM, Apr. 2022, pp. 1–19.

[54] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. D. Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, H. S. Behl, X. Wang, S. Bubeck, R. Eldan, A. T. Kalai, Y. T. Lee, and Y. Li, "Textbooks are all you need," 2023, arXiv:2306.11644.

[55] A. Kashefi and T. Mukerji, "ChatGPT for Programming Numerical Methods," Apr. 2023, arXiv:2303.12093.

[56] L. Chemnitz, D. Reichenbach, H. Aldebes, M. Naveed, K. Narasimhan, and M. Mezini, "Towards code generation from bdd test case specifications: A vision," 2023, arXiv:2305.11619.

[57] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "Towards Enhancing In-Context Learning for Code Generation," Mar. 2023, arXiv:2303.17780.

[58] J.-B. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic?" Feb. 2023, arXiv:2210.14699.

[59] J. He and M. Vechev, "Controlling Large Language Models to Generate Secure and Vulnerable Code," Feb. 2023, arXiv:2302.05319.

[60] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design," Mar. 2023, arXiv:2303.07839.

[61] P. Denny, V. Kumar, and N. Giacaman, "Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 1136–1142.

[62] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "Skcoder: A sketch-based approach for automatic code generation," *arXiv prnote arXiv:2302.06144*, 2023.

[63] J. Li, G. Li, Y. Li, and Z. Jin, "Enabling programming thinking in large language models toward code generation," 2023, arXiv:2305.06599.

[64] S. Jiang, Y. Wang, and Y. Wang, "Selfevolve: A code evolution framework via large language models," *arXiv prnote arXiv:2306.02907*, 2023.

[65] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," 2023, arXiv:2305.04087.

[66] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Improving Few-Shot Prompts with Relevant Static Analysis Products," Apr. 2023, arXiv:2304.06815.

[67] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks," *arXiv preprint arXiv:2310.10508*, 2023.

[68] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," *arXiv prnote arXiv:2303.05510*, 2023, to appear, ICLR 2023.

[69] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning Code Generation with Large Language Model," Mar. 2023, arXiv:2303.06689.

[70] K. Zhang, G. Li, J. Li, Z. Li, and Z. Jin, "Toolcoder: Teach code generation models to use api search tools," 2023, arXiv:2305.04032.

[71] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "CODET: Code Generation With Generated Tests," 2023.

[72] J. P. Inala, C. Wang, M. Yang, A. Codas, M. Encarnación, S. K. Lahiri, M. Musuvathi, and J. Gao, "Fault-aware neural code rankers," 2022, arXiv:2206.03865.

[73] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 1219–1231.

[74] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration Code Generation via ChatGPT," Apr. 2023, arXiv:2304.07590.

[75] R. A. Poldrack, T. Lu, and G. Beguš, "AI-assisted coding: Experiments with GPT-4," Apr. 2023, arXiv:2304.13187.

[76] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT," Apr. 2023, arXiv:2304.10778.

[77] A. Borji, "A Categorical Archive of ChatGPT Failures," Apr. 2023, arXiv:2302.03494.

[78] N. Shinn, F. Cassano, B. Labash, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," May 2023, arXiv:2303.11366.

[79] T. Dinh, J. Zhao, S. Tan, R. Negrinho, L. Lausen, S. Zha, and G. Karypis, "Large language models of code fail at completing code with potential bugs," 2023, arXiv:2306.03438.

[80] C. Treude, "Navigating Complexity in Software Engineering: A Prototype for Comparing GPT-n Solutions," Jan. 2023, arXiv:2301.12169.

[81] M. Yan, J. Chen, J. M. Zhang, X. Cao, C. Yang, and M. Harman, "Coco: Testing code generation systems via concretized instructions," 2023, arXiv:2308.13319.

[82] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. Toronto ON Canada: ACM, Mar. 2023, pp. 931–937.

[83] D. Noever and K. Williams, "Chatbots as fluent polyglots: Revisiting breakthrough code snippets," 2023, arXiv:2301.03373.

[84] J. Sun, Q. V. Liao, M. Muller, M. Agarwal, S. Houde, K. Talamadupula, and J. D. Weisz, "Investigating Explainability of Generative AI for Code through Scenario-based Design," in *27th International Conference on Intelligent User Interfaces*. Helsinki Finland: ACM, Mar. 2022, pp. 212–228.

[85] A. H. Mohammadkhani, C. Tantithamthavorn, and H. Hemmati, "Explainable ai for pre-trained code models: What do they learn? when they do not work?" *arXiv preprint arXiv:2211.12821*, 2022.

[86] H. Hajipour, T. Holz, L. Schönherr, and M. Fritz, "Systematically Finding Security Vulnerabilities in Black-Box Code Generation Models," Feb. 2023, arXiv:2302.04012.

[87] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How Secure is Code Generated by ChatGPT?" Apr. 2023, arXiv:2304.09655.

[88] N. Risse and M. Böhme, "Limits of machine learning for automatic vulnerability detection," 2023, arXiv:2306.17193.

[89] J. Savelka, A. Agarwal, C. Bogart, Y. Song, and M. Sakr, "Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?" Mar. 2023, arXiv:2303.09325.

[90] A. Liu, X. Hu, L. Wen, and P. S. Yu, "A comprehensive evaluation of ChatGPT's zero-shot Text-to-SQL capability," Mar. 2023, arXiv:2303.13547.

[91] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–5.

[92] K. Liu, Y. Han, J. Zhang, Z. Chen, F. Sarro, M. Harman, G. Huang, and Y. Ma, "Who judges the judge: An empirical study on online judge tests," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*, Jan. 2023.

[93] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," May 2023, arXiv:2305.01210.

[94] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[95] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," 2023, arXiv:2210.03629.

[96] W. Zhou, S. Kim, V. Murali, and G. A. Aye, "Improving code autocompletion with transfer learning," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 161–162.

[97] A. M. Turing, "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*. Cambridge, England: University Mathematical Laboratory, Jun. 1949, pp. 67–69.

[98] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language model: Survey, landscape, and vision," 2023, arXiv:2307.07221.

[99] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," 2023, arXiv:2302.10166.

[100] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation," May 2023, arXiv:2305.04207.

[101] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code," *arXiv prnote arXiv:2206.01335*, 2022.

[102] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[103] S. Hashtroudi, J. Shin, H. Hemmati, and S. Wang, "Automated test case generation using code models and domain adaptation," *arXiv preprint arXiv:2308.08033*, 2023.

[104] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Exploring the Effectiveness of Large Language Models in Generating Unit Tests," Apr. 2023, arXiv:2305.00418.

[105] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models," 2023.

[106] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative ai," 2023, arXiv:2306.06782.

[107] A. Moradi Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *arXiv e-prints*, pp. arXiv–2308, 2023.

[108] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," *arXiv preprint arXiv:2308.04748*, 2023.

[109] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models," Mar. 2023, arXiv:2212.14834.

[110] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT," Apr. 2023, arXiv:2304.02014.

[111] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, and S.-C. Cheung, "Finding Failure-Inducing Test Cases with ChatGPT," Apr. 2023, arXiv:2304.11686.

[112] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, H. Qian, Y. Liu, and Z. Chen, "Automatic code summarization via ChatGPT: How far are we?" 2023, arXiv:2305.12865.

[113] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2312–2323.

[114] "Prompting is all you need: Automated Android bug replay with Large Language Models," in 46th *International Conference on Software Engineering (ICSE 2024)*, April 2024, to appear.

[115] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," 2023, arXiv:2305.04207.

[116] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive Test Generation Using a Large Language Model," Feb. 2023, arXiv:2302.06527.

[117] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "ChatUniTest: a ChatGPT-based automated unit test generation tool," May 2023, arXiv:2305.04764.

[118] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 974–985. [Online]. Available: https://doi.org/10.1145/3377811.3380420

[119] Z. Sun, J. M. Zhang, Y. Xiong, M. Harman, M. Papadakis, and L. Zhang, "Improving machine translation systems via isotopic replacement," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1181–1192. [Online]. Available: https://doi.org/10.1145/3510003.3510206

[120] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2023.

[121] T. J. Ostrand and E. J. Weyuker, "Data flow-based test adequacy analysis for languages with pointers," in *Symposium on Testing, Analysis, and Verification (TAV4)*, Victoria, BC, Canada, 1991, pp. 74–86.

[122] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 85–103.

[123] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649 – 678, September–October 2011.

[124] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019.

[125] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 597–608.

[126] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Efficient mutation testing via pre-trained language models," *arXiv preprint arXiv:2301.03543*, 2023.

[127] A. Garg, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Vulnerability Mimicking Mutants," Mar. 2023, arXiv:2303.04247.

[128] A. E. I. Brownlee, J. Callan, K. Even-Mendoza, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Enhancing genetic improvement mutations using large language models," in *SSBSE 2023: Challenge Track*. San Francisco, USA: Springer, 8 Dec 2023, to appear.

[129] R. Pan, T. A. Ghaleb, and L. Briand, "LTM: Scalable and Black-box Similarity-based Test Suite Minimization based on Language Models," *arXiv prnote arXiv:2304.01397*, 2023.

[130] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan, "Code Execution with Pre-trained Language Models," May 2023, arXiv:2305.05383.

[131] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis (keynote paper)," in 18th *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*, Madrid, Spain, September 23rd-24th 2018, pp. 1–23.

[132] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, "FlakyCat: Predicting flaky tests categories using few-shot learning," in

2023 IEEE/ACM International Conference on Automation of Software Test (AST), 2023, pp. 140–151.

[133] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *IEEE Transactions on Software Engineering*, 2022.

[134] S. Fatima, H. Hemmati, and L. Briand, "Black-box prediction of flaky test fix categories using language models," 2023, arXiv:2307.00012.

[135] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in 23rd *Automated Software Engineering (ASE '08)*. L'Aquila, Italy: IEEE, 2008, pp. 218–227.

[136] S. Yoo and M. Harman, "Test data regeneration: Generating new test data from existing test data," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 3, pp. 171–201, May 2012.

[137] R. Abou Assi, C. Trad, M. Maalouf, and W. Masri, "Coincidental correctness in the defects4j benchmark," *Software Testing, Verification and Reliability*, vol. 29, no. 3, p. e1696, 2019.

[138] K. Androutsopoulos, D. Clark, H. Dan, M. Harman, and R. Hierons, "An analysis of the relationship between conditional entropy and failed error propagation in software testing," in 36th *International Conference on Software Engineering (ICSE 2014)*, Hyderabad, India, June 2014, pp. 573–583.

[139] R. Feldt, S. Kang, J. Yoon, and S. Yoo, "Towards autonomous testing agents via conversational large language models," 2023, arXiv:2306.05152.

[140] S. Kang, G. An, and S. Yoo, "A preliminary evaluation of llm-based fault localization," *arXiv preprint arXiv:2308.05487*, 2023.

[141] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," 2023, arXiv:2308.15276.

[142] S. Feng and C. Chen, "Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models," Jun. 2023, arXiv:2306.01987.

[143] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with LLMs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5131–5140.

[144] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, "Condefects: A new dataset to address the data leakage concern for llm-based fault localization and program repair," *arXiv preprint arXiv:2310.16253*, 2023.

[145] E. Mashhadi, H. Ahmadvand, and H. Hemmati, "Method-level bug severity prediction using source code metrics and llms," 2023.

[146] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[147] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," 2023, arXiv:2303.18184.

[148] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*, Montreal, Canada, 2019.

[149] M. Harman, "Scaling genetic improvement and automated program repair (keynote paper)," in *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 2022, pp. 1–7.

[150] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[151] M. Harman and B. F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[152] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," 2019, arXiv:1812.08693.

[153] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[154] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," 2022, arXiv:2210.14179.

[155] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair," in *FSE 2023*, 2023.

[156] C. S. Xia and L. Zhang, "Conversational Automated Program Repair," Jan. 2023, arXiv:2301.13246.

[157] ——, "Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," Apr. 2023, arXiv:2304.00385.

[158] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching Large Language Models to Self-Debug," Apr. 2023, arXiv:2304.05128.

[159] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," 2022, arXiv:2112.02125.

[160] Y. Charalambous, N. Tihanyi, R. Jain, Y. Sun, M. A. Ferrag, and L. C. Cordeiro, "A new era in software security: Towards self-healing software via large language models and formal verification," *arXiv preprint arXiv:2305.14752*, 2023.

[161] J. Cao, M. Li, M. Wen, and S.-c. Cheung, "A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair," Apr. 2023, arXiv:2304.08191.

[162] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions," Apr. 2023, arXiv:2304.03816.

[163] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models," Feb. 2023, arXiv:2301.03797.

[164] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," 2023, arXiv:2302.05020.

[165] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-End Program Repair with LLMs," Mar. 2023, arXiv:2303.07263.

[166] B. Berabi, J. He, V. Raychev, and M. T. Vechev, "TFix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 780–791.

[167] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," in *ASE 2023*, 2023.

[168] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," 2023, arXiv:2304.02195.

[169] D. Sobania, A. Geiger, J. Callan, A. Brownlee, C. Hanna, R. Moussa, M. Zamorano Lopez, J. Petke, and F. Sarro, "Evaluating explanations for software patches generated by large language models," in *SSBSE 2023: Challenge Track*, ser. LNCS. San Francisco, USA: Springer, 8 Dec 2023, to appear.

[170] A. A. Lovelace, "Sketch of the analytical engine invented by Charles Babbage by L. F. Menabrea of Turin, officer of the military engineers, with notes by the translator," 1843, translation with notes on article in italian in Bibliothèque Universelle de Genève, October, 1842, Number 82.

[171] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, techniques and tools*, 1986.

[172] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica*, vol. 6, pp. 41–60, 1976.

[173] H. Partsch, *The CIP Transformation System*, 1984, pp. 305–322, peter Pepper (ed.).

[174] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[175] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP'09), Operating Systems Review (OSR)*, October 2009, pp. 87–102.

[176] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu, "Genetic improvement for adaptive software engineering (keynote paper)," in 9th *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014)*. New York, NY, USA: ACM, 2014, pp. 1–4.

[177] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Transactions on Evolutionary Computation (TEVC)*, vol. 19, no. 1, pp. 118–135, Feb 2015.

[178] A. Madaan, A. Shypula, U. Alon, M. Hashemi, P. Ranganathan, Y. Yang, G. Neubig, and A. Yazdanbakhsh, "Learning Performance-Improving Code Edits," Feb. 2023, arXiv:2302.07867.

[179] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, "Deepdev-perf: a deep learning-based approach for improving

software performance," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 948–958.

[180] S. Kang and S. Yoo, "Towards objective-tailored genetic improvement through large language models," 2023, arXiv:2304.09386.

[181] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "Rapgen: An approach for fixing code inefficiencies in zero-shot," *arXiv preprint arXiv:2306.17077*, 2023.

[182] Z. Chen, S. Fang, and M. Monperrus, "Supersonic: Learning to generate source code optimisations in c/c++," *arXiv preprint arXiv:2309.14846*, 2023.

[183] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather, "Large language models for compiler optimization," 2023.

[184] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[185] Q. Huang, J. Zhu, Z. Li, Z. Xing, C. Wang, and X. Xu, "PCR-Chain: Partial code reuse assisted by hierarchical chaining of prompts on frozen copilot," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, pp. 1–5.

[186] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, p. 111796, 2023.

[187] J. Zhao, Y. Rong, Y. Guo, Y. He, and H. Chen, "Understanding programs by exploiting (fuzzing) test cases," *arXiv prnote arXiv:2305.13592*, 2023.

[188] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14–34, 2017.

[189] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[190] T. J. McCabe, "A complexity measure," vol. 2, pp. 308–320, 1976.

[191] B. R. Bruce, J. Petke, M. Harman, and E. T. Barr, "Approximate oracles and synergy in software energy search spaces," *IEEE Transactions on Software Engineering*, vol. 45, no. 11, pp. 1150–1169, 2019.

[192] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for OLED smartphones," in *36th International Conference on Software Engineering (ICSE 2014)*. New York, NY, USA: ACM, 2014, pp. 527–538.

[193] D. R. White, J. Clark, J. Jacob, and S. Poulding, "Searching for resource-efficient programs: Low-power pseudorandom number generators," in *2008 Genetic and Evolutionary Computation Conference (GECCO 2008)*, Atlanta, USA, Jul. 2008, pp. 1775–1782.

[194] F. Wu, M. Harman, Y. Jia, J. Krinke, and W. Weimer, "Deep parameter optimisation," in *Genetic and evolutionary computation conference (GECCO 2015)*, Madrid, Spain, July 2015, pp. 1375–1382.

[195] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs (keynote paper)," in 27th *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, September 2012, pp. 1–14.

[196] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1995.

[197] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman, "Evaluating automatic program repair capabilities to repair API misuses," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2658–2679, 2022.

[198] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, H. Qian, Y. Liu, and Z. Chen, "Automatic Code Summarization via ChatGPT: How Far Are We?" May 2023, arXiv:2305.12865.

[199] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "An Empirical Study on Using Large Language Models for Multi-Intent Comment Generation," Apr. 2023, arXiv:2304.11384.

[200] "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in 46th *International Conference on Software Engineering (ICSE 2024)*, April 2024, to appear.

[201] T. Menzies and T. Zimmermann, "Software analytics: so what?" *IEEE Software*, vol. 30, no. 4, pp. 31–37, 2013.

[202] A. E. Hassan, "The road ahead for mining software repositories," in *2008 Frontiers of Software Maintenance*. IEEE, 2008, pp. 48–57.

[203] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, 2017.

[204] T. Menzies and T. Zimmermann, "Software analytics: What's next?" *IEEE Software*, vol. 35, no. 5, pp. 64–70, 2018.

[205] G. Spanoudakis and A. Zisman, "Software traceability: a roadmap," in *Handbook of software engineering and knowledge engineering: vol 3: recent advances*. World Scientific, 2005, pp. 395–428.

[206] J. Cleland-Huang, O. Gotel, A. Zisman *et al.*, *Software and systems traceability*. Springer, 2012, vol. 2, no. 3.

[207] C. Lebeuf, M.-A. Storey, and A. Zagalsky, "Software bots," *IEEE Software*, vol. 35, no. 1, pp. 18–23, 2017.

[208] M. Lehman and W. Turski, "Essential properties of ipses," *ACM SIGSOFT Software Engineering Notes*, vol. 12, no. 1, pp. 52–55, 1987.

[209] M. H. Hamilton and W. R. Hackler, "Universal systems language: lessons learned from apollo," *Computer*, vol. 41, no. 12, pp. 34–43, 2008.

[210] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. New Orleans LA USA: ACM, Apr. 2022, pp. 1–7.

[211] Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, "Investigating code generation performance of ChatGPT with crowdsourcing social data," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2023, pp. 876–885.

[212] A. Ahmad, M. Waseem, P. Liang, M. Fehmideh, M. S. Aktar, and T. Mikkonen, "Towards Human-Bot Collaborative Software Architecting with ChatGPT," Feb. 2023, arXiv:2302.14600.

[213] R. Pressman, *Software Engineering: A Practitioner's Approach*, 3rd ed. Maidenhead, Berkshire, England, UK.: McGraw-Hill Book Company Europe, 1992, european adaptation (1994). Adapted by Darrel Ince. ISBN 0-07-707936-1.

[214] K. Bojarczuk, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, S. Lucas, E. Meijer, R. Rojas, and S. Sapora, "Measurement challenges for cyber cyber digital twins: Experiences from the deployment of Facebook's WW simulation system (keynote paper)," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '21)*, October 2021, keynote talk given jointly by Maria Lomeli and Mark Harman.

[215] C. Cornes, J. Courant, J.-C. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, C. Paulin-Mohring, A. Saibi, and B. Werner, "The coq proof assistant, reference manual, version 5.10," Inria, Institut National de Recherche en Informatique et en Automatique, Technical Report RT-0177, Jul. 1995.

[216] K. B. Gallagher, "Evaluating the surgeon's assistant: Results of a pilot study," in *International Conference on Software Maintenance (ICSE '92)*, Los Alamitos, California, USA, Nov. 1992, pp. 236–244.

[217] M. Ward, F. W. Calliss, and M. Munro, "The maintainer's assistant," in *International Conference on Software Maintenance (ICSM 1989)*, Los Alamitos, California, USA, 1989, p. 307.

[218] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*. Sydney NSW Australia: ACM, Mar. 2023, pp. 491–514.

[219] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is ChatGPT the Ultimate Programming Assistant – How far is it?" Apr. 2023, arXiv:2304.11938.

[220] L. Meckler and P. Verma, "Teachers are on alert for inevitable cheating after release of ChatGPT," *The Washington post*, December 2022.

[221] W. Heaven, "ChatGPT is going to change education, not destroy it," *MIT Technology review*, April 2023.

[222] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "ChatGPT and Software Testing Education: Promises & Perils," Mar. 2023, arXiv:2302.03287.

[223] J. Savelka, A. Agarwal, C. Bogart, and M. Sakr, "Large Language Models (GPT) Struggle to Answer Multiple-Choice Questions about Code," Mar. 2023, arXiv:2303.08033.

[224] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research V.1*. Lugano and Virtual Event Switzerland: ACM, Aug. 2022, pp. 27–43.

[225] "TRACED: Execution-aware pre-training for source code," in 46th *International Conference on Software Engineering (ICSE 2024)*, April 2024, to appear.

[226] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020, arXiv:2001.08361.

[227] Aakanksha Chowdhery et al., "PaLM: Scaling language modeling with pathways," 2022, arXiv:2204.02311.

[228] Jack W. Rae et al., "Scaling language models: Methods, analysis & insights from training gopher," 2022, arXiv:2112.11446.

[229] P. Henderson, J. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau, "Towards the systematic reporting of the energy and carbon footprints of machine learning," *J. Mach. Learn. Res.*, vol. 21, no. 1, jan 2020.

[230] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," 2022, arXiv:2203.15556.

[231] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021, arXiv:2106.09685.

[232] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," 2018, arXiv:1802.05668.

[233] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *CoRR*, vol. abs/1812.00140, 2018, 1812.00140.

[234] M. Hort, A. Grishina, and L. Moonen, "An exploratory literature study on sharing and energy use of language models for source code," 2023, arXiv:2307.02443.

[235] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "ChatGPT vs SBST: A comparative assessment of unit test suite generation," 2023, arXiv:2307.00588.

[236] U. Shaham, E. Segal, M. Ivgi, A. Efrat, O. Yoran, A. Haviv, A. Gupta, W. Xiong, M. Geva, J. Berant, and O. Levy, "SCROLLS: Standardized CompaRison Over Long Language Sequences," Oct. 2022, arXiv:2201.03533.