Artificial Intelligence for Software Engineering: The Journey so far and the Road ahead

IFTEKHAR AHMED, University of California, Irvine, USA
ALDEIDA ALETI, Monash University, Australia
HAIPENG CAI, University at Buffalo, SUNY, USA
ALEXANDER CHATZIGEORGIOU, University of Macedonia, Greece
PINJIA HE, The Chinese University of Hong Kong, China
XING HU, Zhejiang University, China
MAURO PEZZÈ, USI Università della Svizzera Italiana, Switzerland
DENYS POSHYVANYK, William & Mary, USA
XIN XIA, Zhejiang University, China

Artificial intelligence and recent advances in deep learning architectures, including transformer networks and large language models, change the way people think and act to solve problems. Software engineering, as an increasingly complex process to design, develop, test, deploy, and maintain large-scale software systems for solving real-world challenges, is profoundly affected by many revolutionary artificial intelligence tools in general, and machine learning in particular. In this roadmap for artificial intelligence in software engineering, we highlight the recent deep impact of artificial intelligence on software engineering by discussing successful stories of applications of artificial intelligence to classic and new software development challenges. We identify the new challenges that the software engineering community has to address in the coming years to successfully apply artificial intelligence in software engineering, and we share our research roadmap towards the effective use of artificial intelligence in the software engineering profession, while still protecting fundamental human values.

We spotlight three main areas that challenge the research in software engineering: the use of generative artificial intelligence and large language models for engineering large software systems, the need of large and unbiased datasets and benchmarks for training and evaluating deep learning and large language models for software engineering, and the need of a new code of digital ethics to apply artificial intelligence in software engineering.

CCS Concepts: • **Software and its engineering → Software development techniques**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Automated Software Development, Machine Learning, Large Language Models, Artificial Intelligence, Explainable AI, Ethical AI

Authors' addresses: Iftekhar Ahmed, iftekha@uci.edu, University of California, Irvine, Irvine, California, USA; Aldeida Aleti, aldeida.aleti@monash.edu, Monash University, Melbourne, Australia; Haipeng Cai, haipengc@buffalo.edu, University at Buffalo, SUNY, Buffalo, New York, USA; Alexander Chatzigeorgiou, achat@uom.edu.gr, University of Macedonia, Greece; Pinjia He, hepinjia@cuhk.edu.cn, The Chinese University of Hong Kong, Hong Kong, China; Xing Hu, xinghu@zju.edu.cn, Zhejiang University, China; Mauro Pezzè, mauro.pezze@usi.ch, USI Università della Svizzera Italiana, Lugano, Switzerland; Denys Poshyvanyk, dposhyvanyk@gmail.com, William & Mary, USA; Xin Xia, xin.xia@acm.org, Zhejiang University, Hangzhou, China.

ACM 1557-7392/2025/4-ART
https://doi.org/10.1145/3719006

1 INTRODUCTION

Various quotes reveal that even experts have a sense of awe towards the pace of progress in artificial intelligence fueling debates about its potential to give rise to smarter-than-human intelligence and the associated legal, moral, and ethical issues that emerge. On the one hand, almost every professional domain has incorporated or experimented with artificial intelligence platforms that can be prompted by human input [27]. Almost 77% of consumers today use an artificial intelligence-powered service or device [5]. On the other hand, people's perception of systems powered by artificial intelligence as a mixture of 'software and sorcery' [97] sparks skepticism against artificial intelligence and is often driven by misconceptions and myths.

Regardless of a positive or negative position against technological progress and recent notable developments in transformer networks and large language models, artificial intelligence is expected to disrupt the nature of the software engineering profession, considering the naturalness of software [48] and the rapidly increasing maturity of machine learning approaches and tools. `OpenAI ChatGPT`, `GitHub Copilot`, `Amazon CodeWhisperer` are the tip of the iceberg consisting of intense efforts to train models on the enormous corpus of existing datasets (including source code), and we have only skimmed the surface of research and entrepreneurship in this area so far.

Wang et al.'s recent 12-year systematic literature review on 1,428 machine learning and deep learning-related software engineering papers indicates a steadily increasing interest, with defect analysis and software maintenance and evolution taking up over half of the collection [121]. Watson et al.'s review on the use of deep learning in software engineering research that spans 128 articles across 23 software engineering tasks reveals that software engineers commonly use generative artificial intelligence to synthesize, comprehend, and generate code [123]. Recent reviews discuss the application of machine learning to select and prioritize test cases [89], and to predict maintenance issues [10] and defects in software systems [73]. The many recent approaches highlight both the potential of artificial intelligence in reshaping software engineering, bringing in an era where machines assist or even co-create alongside humans, and the need for thoughtful evidence-based approaches to address both ethical considerations and the impact on the workforce.

In this roadmap paper, we identify the open technical and organizational challenges for the software engineering community. The technical challenges come from the use of artificial intelligence and large language models for engineering large software systems, and the needs of high-quality, both general and domain-specific, datasets for training and evaluating models. The organizational challenges come from the perspective of human-artificial intelligence collaboration in software engineering teams, and the inevitable changes to the software process and the engineering profession itself. The roadmap outlines the envisioned 2030 research horizon emerging from the lessons learned and the challenges that lie ahead.

2 GENERATIVE ARTIFICIAL INTELLIGENCE AND LARGE LANGUAGE MODELS FOR SOFTWARE ENGINEERING

Deep learning and large language models [45] have emerged as a transformative technology with the potential to disrupt many domains. Deep learning is already exerting a profound influence on the software development life cycle and its main activities, requirement analysis and elicitation, architecture and detailed design, implementation and integration, testing, deployment, and maintenance. The proliferation of applications of large language models opens new challenges that we need to address to scale up to mature industrial applications. For example, large language models suffer from many false positives when generating test assertions [33, 111, 124].

In this section, we discuss the main research directions to handle the limited context length of large language models and meet the efficiency requirements of software engineering tasks: *engineering prompts*, *integrating deep learning and large language models with classic software engineering*, *evaluating large language models*, *and*

explaining artificial intelligence decisions in software engineering. In the next section, we discuss the research frontier related to datasets.

2.1 Prompt Engineering

The performance and reliability of large language models depend on the prompts that we use to query the large language model. Engineering prompts that optimize large language models is far from trivial, and is still a large open challenge in many application domains. The current techniques for engineering prompts to elicit the knowledge and reasoning abilities of large language models for specific tasks, like *Chain-of-Thought*, *ReAct* and *Tree-of-Thoughts*, mainly focus on natural language processing tasks. The source code is different from the natural language [18], and it is an open question whether these techniques developed for natural language processing can be successfully applied in software engineering.

The research has achieved impressive results by augmenting the prompts with task-specific and in-context examples and demonstrations. Large language models learn from task instructions, example tasks, and a query that consists of the prompt (*in context learning*). The main advantage of in-context learning is the limited cost of model training; however, it is still an open challenge to confirm both the promising results of early studies on the composition of good in-context learning demonstrations [38] and define design patterns and principles that efficiently limit the impact of in-context learning on the performance of large language models. The successful applications of *zero-shot and one-shot learning* for summarizing code, *prompt combinations* for refining code, and *conversational prompting* for automatically repairing programs open new horizons for efficient prompting in software engineering.

Open Challenge-1 *How to engineer and combine prompts with domain specific information to efficiently and effectively address software engineering tasks?*

Appropriately combining well defined prompts with domain specific information entails addressing the following detailed questions:

(a) *How to Improve the simple instructions used in the experiments so far ("You are an expert Java programmer; please describe the functionality of the method.") by carefully designing prompts?* The recent applications of zero-shot and one-shot learning (no in-context learning examples and only an in-context-learning example in the prompt, respectively) for summarizing code [39] do not fully outperform the state-of-the-art code summarization approaches yet.

(b) *How to improve prompts with an in-depth analysis of the impact of the different components of prompts?* Recent studies on the performance of combination of different prompts for refining code (redefining code based on code review comments) [43] focus on different combinations of preset prompts.

(c) *How to exploit heterogeneous types of information and presentation styles in conversational prompting for different software engineering tasks?* Conversational ways of prompting large language models have been explored in various software engineering domains, such as automatic program repair [130, 131]. Conversational prompts interleave generation and validation to provide immediate feedback to generate new candidates (patches in the case of automatic program repair). Conversational prompting uses feedback information (such as the results of executing a test case execution and information from previous generations) as an additional context for subsequent generations. So far, the no-human-in-the-loop studies consider only information from failing test cases and previous patches as validation feedback.

2.2 Evaluating Large Language Models

Although there exist several studies on the use of deep learning for engineering software systems, the studies on evaluating the products are still very preliminary. The few metrics of deep learning products in software engineering refer to code generation and are limited to accuracy, the similarity between the code generated by the model and the code designed manually, and *efficiency*, the time a model needs to generate code, usually paired with classic readability metrics in the generated code.

Evaluating large language models needs to rely heavily on rigorous testing, but is significantly challenged by the lack of computer-readable specifications, also known as test oracles [14]. The test oracle problem refers to the difficulty in verifying the correctness of a system's outputs during testing. Inadequate evaluation of deep learning-based systems can have significant consequences, ranging from financial losses to compromised patient well-being [95] and safety [120].

To address the wide range of scenarios that require testing, new automated testing methods have been proposed [42, 112, 132]. In contrast to many aspects of testing, the challenge of automating the test oracle has received considerably less attention and still lacks effective solutions [14], in particular in the area of evaluating deep learning-based systems. In current industry practices, organizations often rely on third-party data labeling companies for manual labels [85] or employ large language models as a judge [145]. Manual data set labeling can lead to substantial costs and be error-prone, as shown by a study that found that the ten most cited artificial intelligence data sets are riddled with labeling errors [85]. These datasets are limited in scope and represent only a fraction of the diverse scenarios that might be encountered in the real world. The large-language model-as-a-judge approach, while seemingly promising for its scalability and automation potential, has been shown to suffer from inherent biases [50]. This highlights the critical need for robust and unbiased test oracles to ensure the quality and reliability of deep learning-based systems, such as large language models.

Open Challenge-2 *How to objectively evaluate deep-learning-empowered software processes and products?* Defining metrics for both the use of deep learning in software engineering and the products of engineering software with deep learning entails addressing the following detailed questions:

(a) *What systematic strategies, standards and taxonomies to evaluate large language models for both generating code and other software engineering tasks beyond the few functional metrics used so far, towards metrics for security, robustness and consistency?* It is urgent to establish an effective, objective, and complete evaluation system for large language models for code.

(b) *What suitable metrics for complex scenarios where multiple methods that depend on each other are generated?* The many *evaluation benchmark datasets*, such as `HumanEval`, `MBPP`, `DS-1000`, contain code units, such as functions and statements, that are independent of other code contexts and that correspond to only 30% methods in open source projects.

(c) *How to automate the generation of test oracles for deep learning-based systems, reducing the dependency on manual labeling and lowering associated costs and human effort to test such complex systems?* This will allow for a more effective and efficient evaluation of deep learning-based systems, enabling software engineers to build such systems in far more reliable and cost-effective ways.

2.3 Integrating Deep Learning and Large Language Models with Classic Traditional Software Engineering

Deep learning dramatically impacts all the phases of the software engineering life cycle, from requirements elicitation and validation to source code generation, testing, and maintenance. In this section, we overview the

state of the art of integrating deep learning in the different phases of the software life cycle, and discuss the main open challenges for an effective and sustainable integration of deep learning in the software process.

2.3.1 Deep Learning for Software Requirements and Design. Deep learning models, such as deep siamese networks [100], convolutional neural networks [94], and long short-term memory [61] are mainly employed for software requirement classification [56] in the context of requirement analysis [11]. Deep learning finds applications in requirements classification [116], requirement extraction [61], requirement traceability [122], requirement validation [127], text-to-requirement generation [107], requirement-to-domain model [12], and enhancing requirements completeness [68]. Applications of deep learning to software design have garnered relatively less attention in the research arena so far. Only a small number of studies [104] explore the use of deep learning techniques to support the software design process. Design pattern identification [104] and user interface design detection [19, 77] are the two most popular research directions in design-related tasks. Convolutional neural networks have emerged as the predominant deep learning technique employed in this field, as the majority of datasets in this domain consist of images.

2.3.2 Deep Learning for Producing Source Code. Deep learning models are pivotal in expediting and improving the precision of complex coding tasks. The activity of producing source code includes *classification*, *ranking* and *generation* tasks [72, 123, 138]. Generation tasks include code representation generation [54], code generation [23], code completion [26, 60, 126], code summarization [59], code comment generation [40], and method name generation [80]. The classification tasks deal with both code localization [9], focusing primarily on discerning source code within screencasts and type inference [70], code search [62] and clone detection [125]. Triet et al.'s [58] comprehensive review of deep learning techniques for source code modeling and generation identifies the most common approaches for input embeddings, architectures, and training strategies, and reveals both open research challenges and well-established datasets. The current source code models perform significantly worse on corpora different from their training set, and there exists no study yet that uses both examples and descriptions (perform synthesis and induction) to build a neural program model for generating source code that does not already exist in the dataset.

Recent empirical studies indicate that the accuracy of Copilot-generated code depends on different factors, such as the programming language and the complexity of the task [29, 90, 140], and highlight the crucial need to evaluate the functional soundness of code produced with artificial intelligence tools, a fundamental concern in software engineering. Several studies have investigated the strengths and weaknesses of code intelligence tools [17, 64, 105], suggesting that code intelligence tools perform well in simple and well-structured tasks; however, they do not yet scale up to complex tasks involving semantic nuance [105].

2.3.3 Deep Learning for Software Testing. The automation and intelligence has seen notable advancements, attributed to advances in deep learning techniques [1]. Wang et al.'s survey [119] comprehensively overviews the use of large language models in software testing. Generative artificial intelligence is used to generate unit test cases [7], test oracles [33, 111] and system test inputs [8], for analyzing bugs and predicting failure [25], for debugging and repairing code [52]. Large language models are already successfully applied to generate logging statements [133], parsing logs [134], and analyzing root causes [6, 24]. The deep learning models found applications in several software testing activities [137] such as detecting bugs [30, 117] and vulnerabilities [106], localizing bugs [143], classifying faults [81], and generating test cases [98], with a predominant attention to bug detection and localization. Convolutional neural networks and long short-term memory-based models, for instance, Bi-directional (Bi-)long short-term memory and Gate Recurrent Units, emerge as the two most frequently employed deep learning techniques in software testing.

2.3.4 Deep Learning for IT Operations. AIOps (Artificial Intelligence for IT Operations) refers to the adoption of artificial intelligence techniques to enhance IT operations. Typical AIOps tasks include anomaly detection [55, 141],

incident triage [20, 41], and root cause analysis [135, 146]. Although AIOps has been explored for decades, the gap between industry and academia still exists and is even growing [46]. Moreover, the early applications of large-language models to software engineering tasks highlight new challenges to AIOps.

(1) *How to effectively leverage the limited context length of large language models in complex operation tasks?*
The context length of large language models is rather limited. However, AIOps commonly involves a variety of large-volume system run-time data, such as metrics, logs, and traces. For example, a cloud service provider can easily produce a petabyte (PB) of log data per day. It is difficult to prompt large language models in the full context of an industry-relevant operation task.
(2) *How to meet the efficiency requirement of AIOps tasks while harnessing the powerful capabilities of large language models?*
The inference speed of large language models is limited. Many prompt engineering techniques, for instance chain-of-thought, that are widely used in practice could produce a large amount of intermediate output that further increases the inference time.

2.3.5 Deep Learning for Program Analysis. Program analysis, that is, the process of automatically analyzing the characteristics of a program (correctness, robustness, security, and more) is pervasive in the software life cycle (optimization, validation, testing, debugging, understanding, maintenance, and more). The increasing scale, complexity, and diversity of software systems challenge program analysis with important open issues:

- the large scale and complexity of software systems limits the efficiency and accuracy of program analysis, with many false positives that reduce the applicability of program analysis approaches and tools;
- the diversity of programming languages and runtime platforms limits the evolution of program analysis techniques beyond uniform platforms, with substantial impact on deployment cost of the corresponding analysis engines;
- the analysis of dynamic programming languages produces incomplete results and ultimately underreporting.

Large language models that use large-scale multidimensional data as well as massive computational resources to tune pre-trained models with strong generalization capabilities can play an important role in addressing the open issues.

Large language models must consider the specificity of programming languages to unleash the full value of large language models in the field of program analysis. Unlike natural languages, programming languages are rigorous mathematical expressions, and there is a massive gap between natural and program languages in terms of corpus mutation, error tolerance, sentence morphology, and so on.

Open Challenge-3 *How to integrate deep learning with classic software engineering approaches for scalable and generalizable approaches to efficiently complete general as well as domain specific software engineering tasks?*
Empowering classic software engineering approaches with a deep learning core entails addressing the following detailed questions:
(a) *How to combine large language models technology (pre-trained on massive datasets) with classical program analysis to overcome the space explosion and over-abstraction of the traditional search-based and abstract program analysis algorithms, and improve the efficiency and accuracy of analysis?*
(b) *How to exploit the generalization capability of large language models to adapt to multi-language and multi-platform program analysis tasks and improve platform-unity-based on multi-task learning and migration learning?*

(c) *How to combine large language models technology with generative and adversarial learning to reduce the uncertainty of program dynamics and improve the completeness of the results of program analysis?*
(d) *How to train large language models for the domain of programming languages?*

2.4 Explainable Artificial Intelligence in Software Engineering

The *interpretability* of software-engineering-related decisions driven by artificial intelligence is a largely unexplored issue. The impressive applicability and flexibility of artificial intelligence methodologies and tools based on machine learning and artificial intelligence come with a lack of transparency, due to the black-box nature of the underlying models, leading to decreased trust. For example, the recommendation to refactor a software module, especially when it involves substantial effort, is of limited value if it is not paired with an explanation that motivates the suggestion. The General Data Protection Regulation (GDPR) of the European Union establishes the right to obtain "*meaningful information about the logic involved*" in automated decision making, a maxim that is commonly interpreted as a "*right to an explanation*' [35]. *Explainable artificial intelligence* entails the many approaches that aim to make decisions based on artificial intelligence comprehensible. The most important concept in explainable artificial intelligence defines the characteristic of a model that *allows a human to understand its function* – how the model works - without the need to explain the algorithmic details by which the model processes data internally [76].

The reasons for a decision matter in many social contexts, and a model should be transparent to confer interpretability [63]. *Transparency* connotes some sense of understanding the internal mechanism by which the model works. Such algorithmic transparency might be extremely challenging for some techniques, such as deep learning models. *Post hoc interpretations* address the transparency of decisions, providing explanations (often textual or visual) by example. Such explanations resemble essentially an answer to a why-question as argued by Miller [74]: *Why is file A defective?, Why is file A defective, while file B is clean? or Why was file A not classified as defective in version 1.2, but was subsequently classified as defective in version 1.3?* [51].

The two most recognized explainable artificial intelligence (XAI) methods are Local Interpretable Model-agnostic Explanations (LIME) and Shapley Additive Explanations (SHAP). These techniques are primarily developed for general AI applications, but their direct application to software engineering tasks, such as defect prediction, vulnerability detection, and software refactoring, remains challenging due to the unique characteristics of software artifacts. Unlike structured tabular data or natural language, software code exhibits strong syntactic and semantic dependencies, which affect feature representation and interpretability [114].

Other available explainability approaches can be categorized as intrinsic or post-hoc, model-specific or model-agnostic, and local or global. Global interpretation has the potential to reveal whether and why models assign importance to specific features. For example, it can determine whether certain qualities play a significant role in identifying a software module as susceptible to defects or vulnerabilities. Local analysis, on the other hand, explains specific model decisions by examining individual cases or predictions. For example, it can identify whether a particular metric exceeding a threshold causes an ML model to have certain behavior [88, 115].

As an example of the importance of transparency, most approaches based on large language models to predict defects [69] do not explain why models produce a particular prediction and fail to uphold privacy laws that require the need to explain any decision made by an algorithm. Jiarpakdee et al.'s evaluation of three model-agnostic techniques showcases the ability to obtain instance explanations for the predictions of defect models (for instance, that a particular Java file is predicted as defective when certain conditions for specific metrics are met) as well as global explanations (for instance, identifying the top k metrics that contribute to the final probability of being defective) [51]. Despite these advances, traditional explainability techniques often remain limited in their ability to fully uncover the reasoning behind complex model decisions. A promising approach to improving transparency

is Neurosymbolic AI, which proposes to combine neural networks' pattern recognition with symbolic AI's logical reasoning, addressing the limitations of traditional explainability techniques. For instance, in code analysis, a neurosymbolic model can not only detect security vulnerabilities but also explain the logical rules behind its decision [113].

Tsoukalas et al.[109] discuss the potential of local and global explainability approaches to pinpoint the most important metrics and thresholds that can be used to identify technical debt, as well as specific opportunities for improvement. In terms of global explanation, shapley additive explanation analysis reveals that complexity, comment density, depth of nesting, and cohesion are consistently the most important metrics that make a class susceptible to having high technical debt. In fact, it is possible to define ranges of metric thresholds to serve as a guide for designing classes. In terms of local explanation, shapley additive explanation force plots can shed light into specific class metrics that have a positive or negative impact on labeling that class as problematic in terms of technical debt.

A significant limitation of current explainability methods is their difficulty in handling multicollinearity, a common characteristic of source code. In programming languages, certain elements exhibit strong correlations, such as a parenthesis consistently following a "for" loop declaration. Many existing explanation techniques struggle in such cases, often producing misleading importance scores where highly dependent features receive inflated or distorted values. As a result, these methods fail to accurately capture the true causal relationships underlying software engineering tasks. A promising solution to this issue is causal interpretability, which removes the influence of confounding factors that might distort explanations. Unlike traditional methods that rely on statistical patterns, causal interpretability focuses on cause-and-effect relationships, leading to more reliable explanations. This approach, first introduced in doCode [79], has significant potential for AI applications in software engineering, particularly for defect prediction, vulnerability detection (e.g., memory injection attacks), and code refactoring. By leveraging causality, AI-driven insights can be grounded in real software properties rather than misleading correlations, making them more trustworthy and useful. However, challenges remain, particularly in identifying potential confounders in software artifacts, which makes causal discovery a crucial area for future research.

The close interaction between users and models to incrementally train and correct the model in *interactive machine learning* and *machine teaching* methods [78] has great potential in SE. The dialogue between models and domain experts both yields highly accurate and trustworthy models and educates users by advancing their understanding of the particular domain. For example, a large language model-based recommender system that allows expert users to reject and approve recommendations for intensive testing of highly vulnerable software modules, especially modules created from generative artificial intelligence, both improves the vulnerability detection model (the human is the teacher) and trains the developers (the human is the learner).

Of particular interest to software engineering tasks is the notion of explainability for Transformer-based language models, such as `BERT`, `GPT`, and `LLaMa-2` whose internal mechanisms are notoriously complex, rendering their explainability very challenging. The interpretability of large language models is further complicated by the vastness of their training data and the sheer size of models that often have billions of parameters, necessitating significant computational resources to derive explanations [144]. Techniques whose main purpose is to improve the accuracy of large language models such as chain-of-thought (CoT) prompting, which refers to techniques that require models to produce intermediate reasoning steps before providing an answer, can also be leveraged to explain a model's decision-making process [128]. Recent research has shown the potential of large language models to serve as tools that provide post-hoc explanations for results obtained by other machine learning models [57].

Considering the meteoric rise of large language models for software engineering (229 relevant research papers are studied in Hou et al.'s systematic literature review [49]), especially for tasks such as code summarization, annotation of code artifacts and code improvement, it is imperative to study and tailor interpretability models of

language models. Consider, for example, the case of Codex, a large language model with 12 billion parameters, whose ability to solve 72.31% of complex Python programming challenges [22] makes it a reasonable alternative for junior programmers. Assistant models that fall under the category of *reinforcement learning from human feedback* that align the model output with human feedback, can be subject to local and global explanation. For the case of Codex, achieving sufficient interpretability can advance the performance of the models in the downstream tasks related to code generation.

Open Challenge-4 The open challenges associated with explainability in artificial-intelligence-driven software engineering arise from the need to build trust in software engineers when relying on highly complex deep learning models for automating everyday tasks and for effectively involving both humans and artificial intelligence in decision making:

(a) *How to incorporate human domain knowledge in Machine Learning models* [129]?
(b) *How to explain Transformer-based language models and reinforcement learning from human feedback models that align the model output with human feedback?*

3 DATASETS AND BENCHMARKS FOR SOFTWARE ENGINEERING

The quality of the inference of the *deep learning* models is highly dependent on the data sets used for training and fine-tuning those models. Although deep learning models can rely on huge datasets for natural languages, datasets for various tasks in software engineering are still limited in size. The early applications of deep learning models to software engineering highlight relevant challenges:

(1) *How to generate high quality and large datasets for training?* We need large, high quality, ethically sourced, and unbiased datasets to properly train deep learning and fine-tune *large language models* for the many applications in software engineering;
(2) *How to evaluate deep learning and large language models in software engineering tasks?* We need benchmarks that align the experiences of developers during the practical development process for source code;
(3) *How to integrate deep learning models and domain knowledge?* We need to combine the linguistic ability of general *large language models* with domain knowledge to generate comprehensive, accurate, and reliable applications that meet the requirements of software engineering.

In this section, we discuss the main research directions to address the need of high quality *datasets*, reliable *benchmarks* and effective use of *domain knowledge*.

3.1 Datasets

Several deep learning models[1] are specifically trained to facilitate software engineering tasks, such as detecting code clones, localizing faults, summarizing code, and identifying and classifying vulnerabilities. When extensively trained on large codebases, deep learning models can learn code patterns that indicate certain program properties. Deep learning models succeed in identifying patterns of the kind observed during training, that is, patterns present in the training dataset, and fail otherwise, due to testing data being out of distribution. We can mitigate out-of-distribution as well as overfitting by training the models on large and diverse datasets [32]. We need large datasets of good quality, that is, both with high label accuracy and representative of real-world data samples, to learn patterns that will occur in future testing data sets from the training datasets. We have both datasets of good quality but small size (with hundreds or at most few thousands of samples, as exemplified in Table 1) and several relatively large but merely oversimplified and unrealistic synthetic datasets. While it is quite easy to generate trivially large datasets for specific tasks, for instance, by simply applying semantics-preserving code

[1]Notably, CodeBERT, GraphCodeBERT, CodeT5, and CodeX, at the time of writing

refactoring schemes, including junk code insertion, to multiply relatively small dataset, the resulting datasets do not significantly increase the knowledge available for training the models.

Table 1. Code Generation, Repair and Security datasets

Task	Dataset	Language	# Cases	Data Types
Generation	APPS	Python	10,000	Programming problems with several unit tests, each problem in the training set includes few correct answers.
	HumanEval	Python	164	Programming questions, each with a function signature and its body, docstrings, and few unit tests.
	MTPB	Seven languages	115	Problems written by experts, each containing a natural language multi-step description
	MBPP	Python	974	Programming problems, each consisting of a task description, code solution, and three automated test cases.
	CodeXGLUE	10 languages	>2,000k	14 datasets for 10 diverse code-related tasks (code-code, text-code, code-text, and text-text)
	XLCoST	Seven languages	>2,000k	Datasets for various code generation and code retrieval tasks
	AiXBench	Java	336	175 automated evaluation cases and 161 manual evaluation cases
	DS-1000	Python	1,000	Data science problems based on 451 StackOverflow problems, covering seven widely used Python libraries.
	CoderEval	Python & Java	230	230 Python & Java methods, with docstrings, signatures, implementations, and tests.
Repair & Security	DeepFix	C	6,975	Fixes of compiler errors in C programs
	Defects4J	Java	835	835 reproducible bugs (plus 29 deprecated bugs) from 17 open-source projects
	Asleep at the Keyboard	C & Python	89	A collection of security-related scenarios prepared by experts

In fact, it is possible to generate large high-quality datasets. However, manually labeling the data does not scale. The recent results on automatically labeling data and generating labeled data to support specific software engineering tasks (including studies on the quality of data [28, 84], on the generation of realistic data samples for detecting, localizing, and repairing vulnerabilities [82, 83], assessing training data quality effects on deep learning-based code [103], and to handle noisy data to train deep learning-based code summarization [99]) indicate a viable solution for specific tasks. However, the models defined so far do not generalize, even when trained on datasets significantly augmented with generated samples [75].

Of particular interest is the use of large-language code models to generate code and test cases, repair code, and ensure code security. The need to properly evaluate code models led to the construction of several datasets, as exemplified in Table 2. Some datasets, like `CodeXGLUE`, collect only metrics on code similarities to evaluate the performance of the model and ignore the runtime results of the generated code. Other datasets collect metrics to also evaluate the functional correctness of the model-generated code, like the *Pass@K* metrics of `HumanEval`.

Table 2. Large language models-generated datasets

Code Generation	APPS [47], HumanEval [21], MBPP [13], DS-1000, CoderEval [142] with many Python and Java programs
Test Case Generation	Methods2Test [110], a supervised dataset of Test Cases and their corresponding Focal Methods from a large set of Java software repositories
Code Repair Datasets	DeepFix [44] for C Defects4J [53] for Java
Code Security Datasets	Asleep at the Keyboard [91], with code generation scenarios designed to test the ability of code generation models to generate secure code
Multi-task Datasets	CodeXGLUE, Code General Language Understanding Evaluation Benchmark [66], tasks and a platform for evaluating and comparing models on code-code, text-code, code-text, and text-text tasks HumanEval with Pass@K as metric to evaluate the functional correctness of model-generated code XLCoST [147] supports both generation and retrieval tasks, that include code translation, code summarization, code generation, natural language and cross-language code retrieval

Many code generation datasets, like `APPS`, `HumanEval` and `MBPP`, only focus on simple standalone functions rather than code in real-world projects. Some recent datasets, like `CoderEval` and `ClassEval`, evaluate the effectiveness of large language models in generating code beyond standalone functions. Many datasets support only one or at most two programming languages. Only few datasets, like `HumanEval-X`, `MultiPL-E`, `CodeXGLUE` and `XLCoST`, provide multi language support. Most datasets avoid data contamination issue by manually labeling data. The `DS-1000` dataset stands out as it performs automated data perturbation.

Large-scale and general-purpose artificial intelligence models, like large language models, that are trained on massive amounts of data, including tremendous volumes of software code, mitigate training issues. Recent applications of large language models to software engineering tasks, such as bug and vulnerability detection and repair, logging, and GUI testing [36] have produced good results, even when large language models are not fine-tuned on specific datasets. The initial results with *large language models* indicate that performance in specific software engineering tasks is still far from making *large language models* practically adoptable.

General purpose large language models (such as `GPT-4` and `LLaMA`) still fail to understand the semantics of the code, while domain-specific large language models trained on massive code datasets (like `CodeLLaMA` and `xCoder`) efficiently target only some software engineering tasks. For example, both `GPT-3.5` and `GPT-4.0` perform better in generating code from a short prompt, without code context, than in adding lines of code to a program at some given locations. When asked to inject a fault at some given locations, `GPT-3.5` often inserts a chunk of code that is entirely irrelevant to the rest of the program, similar to generating an independent code fragment to match the given prompt and then forcibly placing the fragment at the given locations. The better performance in generating code from requirements and similar code samples seen during training than in reasoning about code aspects indicates that large language models do not really understand the code semantics, at least not yet in a context-aware or context-sensitive manner.

A promising research direction in developing effective software engineering applications based on large language models is to fine-tune general-purpose *large language models* for specific software engineering tasks. Fine-tuning large language models for specific tasks is a research direction shared with many application domains and is a key motivation behind OpenAI releasing fine-tuning APIs. The core challenge in software engineering is the creation of task-specific (most likely labeled) datasets. The challenge of data-need for fine-tuning *large language models* may only loom larger because large language models have been trained on massive amounts

of data of diverse nature, making fine-tuning against a moderate scale of task-specific dataset potentially less effective than fine-tuning against pre-trained smaller models.

A dual promising research direction in the development of effective software engineering applications based on large language models is fine-tuning the prompting of general-purpose pre-trained *large language models* for specific tasks (for instance, prompting chain of thought and analogical reasoning [139]), thus bypassing the lack of task-specific training and fine-tuning datasets. Current results indicate that *large language models* are best prompted for a specific task when they are instilled with domain-specific knowledge (or at least elicitation and guidance) through a set of task-specific prompting exemplars that ultimately are a kind of labeled and task-specific dataset. Combining large language models prompting with fine-tuning for specific tasks produces models more efficient than using prompting alone.

Open Challenge-5 *How to generate large datasets with high-quality, unbiased and reliable data for training deep learning models for both general needs and specific tasks, and filter out low-quality data automatically?*
How to fine-tune general purpose large language models for specific software engineering tasks and combine large language models prompting with fine-tuning to produce models useful for specific tasks?

3.2 Benchmarks

The current datasets that are currently used to train large language models crawl open-source platforms such as GitHub and Stack Overflow, with data and code of highly different quality, sometimes without positive *educational significance* for the generated models. The data commonly used for training *large language models* for coding are: (i) *not self-contained,* the code examples often rely on external modules or files, making them difficult to understand without context, (ii) *meaningless,* such as defining constants, setting parameters, or configuring the GUI, (iii) *hidden deep within complex or poorly documented functions,* making them difficult to understand or learn, (iv) *biased* towards certain topics, thus resulting in an uneven distribution of coding concepts and skills in the dataset. Low-quality training sets affect not only the quality of model-generated code but also the fact that models generate defective or toxic code. Both the quality and quantity of pre-training data are crucial. High-quality data can greatly improve models.

The length of code (number of tokens) is significantly longer than that of natural languages, with a consequent limited number of context examples that engineers can use when applying context learning techniques in prompt engineering, and a resulting limited quality of the results obtained with the large language model. Most task data in the software engineering domain contain only inputs and outputs, while the data used to reasoning tasks in the natural language processing domain contain a large amount of intermediate data. The recent results with chain of thoughts show that step-by-step data in the problem-solving process can effectively guide the generation of suitable large language models; however, chain of thoughts data have not been yet deeply studied in the context of software engineering domain.

AIOps, as a representative subfield of software engineering, involves a wide range of tasks, including anomaly detection, root cause analysis, incident mitigation, program repair, and many more. However, most publicly available datasets only focus on two tasks (anomaly detection and root cause analysis), and there is a big gap between the open source datasets and real-world data in the industry. The open-source datasets are either outdated, too small, or too simple, reflect only a small scope of the complexity in the real world, and are not aligned with the experiences of developers during the practical development process. For example, the most widely used log-based anomaly detection datasets (`HDFS` [136] and `BGL` [87]) were collected more than ten years ago. The public availability of benchmarks introduces a bias in evaluating large-language models due to unavoidable data

leakage. This distorts academic research that mostly targets anomaly detection and root cause analysis, with little attention to many tasks that are extremely relevant in industry.

Collecting high-quality datasets in AIOps is very challenging. Practitioners are often reluctant to collect and release industrial datasets due to privacy concerns, since metrics, logs, and traces are typically considered sensitive data. Researchers are willing to collect and release datasets but often lack a complete understanding of real-world scenarios and payloads. Different companies or even different groups might have different settings.

Researchers in academia and industry practitioners own complementary assets, and collecting AIOps datasets requires cooperation between academia and industry.

Open Challenge-6 *How to develop and maintain reliable benchmarks for an unbiased evaluation of all software engineering tasks?*
Developing and maintaining reliable and unbiased benchmarks entitles to address the following questions:
(a) How to develop and update effective large language model-based software engineering applications, by fine-tuning general-purpose foundation-scale large language models that represent all the aspects of AIOps in industrial systems?
(b) How to fine-tune the prompting of general-purpose pre-trained *large language models* and generate a suitable amount of context examples for specific tasks?

3.3 Domain Knowledge

Some domain knowledge is essential for large language models to perform domain-specific tasks. Domain knowledge is essential for an accurate analysis.
(i) *help the large language models understand the context and background of domain-specific tasks.* It may be difficult for general large language models to systematically acquire the domain knowledge that is essential for domain-specific applications. For instance, *large language models* could understand medical concepts and terms, disease characteristics and treatment methods for correctly solving problems in medical software systems; *large language models* need detailed knowledge of financial and economic concepts to produce finance-related software systems.
(ii) *narrow the search space of the problem.* Suitable domain knowledge limits the scope of key information and possible answers, thus narrowing the search space, improving the efficiency and accuracy of the model, and helping large language models quickly and accurately accomplish tasks in the specific domains.
(iii) *correct model misperception.* General *large language models* do not have a perfect and accurate knowledge of all domains. Domain experts can help large language models acquire the domain knowledge required to properly solve domain-specific tasks.

A promising future direction is to combine the linguistic ability of general large language models with domain knowledge to generate comprehensive, accurate, and reliable natural language processing applications that meet the requirements of various domains. Feeding large language models with accurate domain knowledge is still a largely open challenge in software engineering.

Open Challenge-7 *What domain knowledge is needed to fine-tune, incrementally train, and appropriately prompt large language models to perform domain-specific tasks?*

4 EFFECTIVE, EFFICIENT AND ETHICAL APPLICATION OF ARTIFICIAL INTELLIGENCE IN SOFTWARE ENGINEERING

We envision a future where artificial intelligence seamlessly integrates into software engineering processes, augmenting human capabilities, and enabling transformative advancements in software development. In the

future, artificial intelligence-powered tools and technologies will play a central role in enhancing productivity, quality assurance, and decision-making throughout the software development lifecycle. Artificial intelligence-powered tools will automate routine tasks, streamline workflows, and optimize resource allocation, allowing software engineers to focus on creative problem solving and innovation. However, artificial intelligence will introduce new ethical challenges, including bias in artificial intelligence models, privacy concerns, copyright and licensing issues, and the responsible use of artificial intelligence in software systems. There is a need to specialize in developing, maintaining, and fine-tuning artificial intelligence models for software engineering tasks, as well as better understanding how artificial intelligence can be applied in various software areas and the different ethical challenges involved.

The demand for code intelligence tools, such as `ChatGPT`, `GitHub Copilot`, and `Amazon Code Whisperer`, has surged in recent years as they redefine the landscape of software development. Code intelligence tools support developers on a spectrum of tasks, including code generation, testing, and repair. However, despite their remarkable capabilities, they grapple with significant challenges that hinder their effectiveness and security:

(i) Code intelligence tools are susceptible to unauthorized exploitation, presenting risks such as data leaks in which sensitive or confidential information may be exposed or accessed inadvertently by unauthorized parties. They are vulnerable to license violations, which occur when code protected by licenses is used or distributed without proper authorization.

(ii) The opaque nature of code intelligence tools poses a substantial barrier to external audits of their training datasets. Even for open-weight models, documentation on training data and fine-tuning relationships is often incomplete [102]. This lack of transparency not only compromises the integrity of the tools, but also exacerbates the risk of unauthorized usage, further complicating efforts to ensure data security and compliance.

(iii) The code generated by these tools often exhibits vulnerabilities, leading to potential security breaches and system compromises. For example, research has revealed that a significant proportion (up to 40%) of the code generated by one of the most popular code intelligence tools, Copilot, was found to be vulnerable [92].

(iv) The impressive performance of code intelligence tools is accompanied by high operational costs and significant energy consumption. For instance, `GitHub Copilot` reportedly experiences an average loss exceeding $20 per user per month, in addition to a subscription fee of $10 [2]. Beyond financial implications, there are concerns about environmental impact. Training of `GPT-4`, the 1.8-trillion-parameter large language model that powers `GitHub Copilot`, can result in carbon emissions between 12,456 and 14,994 metric tons of CO2 [3]. Emissions persist after model deployment for inference purposes, contributing a substantial portion, ranging from 33% to 90%, of the carbon footprint in companies such as `Meta` and `Nvidia` [67]. Intense computational requirements also lead to inevitable response latency, which has been identified as an important factor influencing developers' choices with regard to these tools [118].

(v) The large generative models that produce this code are often trained on data corpora, which include open-source projects under varying software licenses, ranging from attribution only to copy-left. Models may produce verbatim or slightly modified copies of works in their training data, which could potentially violate the international copyright law if the development team does not take appropriate steps. Furthermore, it remains unclear whether the model output should be considered a derivative work of the model's training data, implicating copyright and licensing.

(vi) Questions also surround the copyrightability/ownership of generated code. A recent survey of GitHub developers found that their views on ownership varied widely [101]. Regulatory bodies across the globe have already started to make decisions regarding generative AI outputs. For example, the US Copyright Office recently proposed guidance that work solely produced by a generative AI, regardless of the initiation effort required, is not copyrightable under US law [86]. Beyond just a theoretical question, determining the ownership/copyright

requirements that surround code written by generative AI is paramount if that code is to be used effectively in the industry.

Our vision encompasses the ethical and responsible use of artificial intelligence in software engineering. By 2030, artificial intelligence systems in software engineering will be designed and deployed with robust safeguards to ensure fairness, transparency, and accountability. These systems will uphold the principles of privacy, data protection, and security, thereby fostering trust and confidence among stakeholders.

Our vision extends to the efficient utilization of artificial intelligence in software engineering, where advanced algorithms and machine learning techniques will be used to optimize software development processes, improve resource allocation, and improve project management practices. Through continuous learning and adaptation, artificial intelligence systems will evolve to meet the evolving needs and challenges of the software engineering domain. In realizing our vision, collaboration and interdisciplinary research will be essential. We envision a future where academia, industry, and policymakers work together to advance the frontier of artificial intelligence in software engineering, sharing knowledge, best practices, and standards to ensure the responsible and beneficial integration of artificial intelligence technologies.

Our vision for the effective, efficient and ethical application of artificial intelligence in software engineering by 2030 is one in which artificial intelligence serves as a catalyst for innovation, empowerment, and sustainable growth in the software engineering industry, enriching the lives of individuals and communities around the world.

Artificial intelligence will play an important role in the management of the supply chain of software construction, delivery, and maintenance [65]. The new role of artificial intelligence in software engineering raises important questions about the profession of software engineering: Will artificial intelligence take over all technical jobs, including software development [71]? Will artificial intelligence engineering be the privilege of data science experts, or will artificial intelligence just open a new programming paradigm where software professionals will care more about the specifications and will have to adapt in order to keep their jobs?

Artificial intelligence technologies streamline and automate repetitive tasks in software development, such as code generation, testing, debugging, and maintenance. Artificial intelligence-powered tools can help identify bugs, vulnerabilities, and inconsistencies in code, leading to higher quality software products. Automated testing and code review systems driven by artificial intelligence can detect issues early in the development lifecycle, reducing the likelihood of errors and enhancing overall software reliability. An Amazon report indicates that `CodeWhisperer` [31] can speed up developers' tasks by 57% and users' tasks by 27% [4]. A recent report indicates that the `Codex` model is effective in writing code for programming tasks framed in simple English by successfully completing 29% of the challenges in Python, and the success rate increases to 70% when employing repeated sampling techniques [22].

Artificial intelligence enables software engineers to explore new approaches and solutions that were previously impractical or impossible [34]. Artificial intelligence can uncover patterns and insights from vast amounts of data, leading to innovative software designs, advanced algorithms, and novel applications in various domains. Artificial intelligence-powered development environments can adapt to individual developers' preferences, skill levels, and coding styles. The personalized integrated development environment provides tailored recommendations, suggestions, and code snippets, enhancing developers' productivity and learning experience. Software engineering teams can allocate resources more effectively, optimize project timelines, and improve overall project management, leveraging predictive analytics and data-driven decision making.

The benefits of artificial intelligence come with adverse effects, which include overreliance on artificial intelligence, job displacement, income inequality, and bias. Artificial intelligence algorithms are susceptible to bias [93], which can lead to unfair or discriminatory results in software engineering processes. Biases present

in training data or algorithmic decision making can perpetuate existing societal inequalities, leading to unfair treatment in software development projects or products.

Research has explored the potential disruptive effects of artificial intelligence and automation on the job market [37]. Innovations perceived as threatening job security often encounter resistance [96]. To address this challenge, it is important that organizations proactively communicate the benefits of generative artificial intelligence and prioritize initiatives for the upgrading and reskilling of employees [16]. By fostering a culture of adaptation and showcasing the advantages of artificial intelligence technologies, organizations can navigate the transition to automation more effectively while ensuring the readiness and resilience of the workforce.

Automation of repetitive tasks through artificial intelligence can lead to job displacement or restructuring within the software engineering profession [108]. Certain roles that primarily involve repetitive or mundane tasks, such as manual testing or basic coding, could be automated, leading to a shift in job responsibilities or potential job loss for some professionals. Basic coding skills are poised to become commoditized as automation allows for the generation of the majority of code, thereby diminishing the intrinsic value of traditional coding proficiency. This shift may result in a decline in wages across various software-related roles and could even lead to the elimination of certain positions altogether. In contrast, the demand for people skilled in architecting and designing solutions will skyrocket. Those who are proficient in the design of fundamental problem-solving strategies and novel solutions will experience exponential growth in their value. Such individuals possess the ability to perform tasks that would typically require the effort of an entire team, emphasizing the increasing importance of strategic thinking and solution-oriented approaches in the evolving landscape of technology.

As a result, the impact of artificial intelligence on income inequality within the software engineering profession is a concern. Automation driven by artificial intelligence can lead to increased demand for specialized skills in artificial intelligence development and machine learning, potentially widening the income gap between professionals with artificial intelligence expertise and those without it. This could exacerbate existing income inequalities within the software engineering workforce.

The integration of artificial intelligence into the software engineering profession is likely to have a significant impact on how software engineering is taught in universities [15]. Universities are likely to adapt their software engineering curricula to incorporate technologies and topics related to artificial intelligence. This may include the introduction of courses on machine learning, data science, and artificial intelligence ethics, as well as the integration of artificial intelligence concepts into existing software engineering courses. Universities may offer specialized streams within their software engineering programs focused specifically on artificial intelligence and machine learning. These streams would provide students with in-depth knowledge and practical skills in artificial intelligence development and applications within the software engineering context. Given the importance of artificial intelligence ethics, Software Engineering degrees may incorporate ethics and responsible artificial intelligence considerations into their software engineering curricula. This would involve educating students about the ethical implications of artificial intelligence technologies, including issues such as bias, fairness, transparency, accountability, and privacy.

Open Challenge-8 *What ethic code and software engineering profession is in the era of artificial intelligence-powered software engineering?*

Addressing the disruptive role of artificial intelligence in software engineering entails answering the following questions:

(a) How to prevent unauthorized exploitation, data leaks, license violations, vulnerabilities, security breaches, and system compromises, and enable independent and transparent audits of the training datasets of large language models and of the generated code large language models?

(b) How to prevent overreliance on artificial intelligence and reduce operational costs and energy consumption of large language models?

5 2030 RESEARCH HORIZON

In this last section, we outline the 2030 research horizon that emerges from the study of past practices and the challenges that lie ahead. We identify the key challenges shaping the new software engineering skyline. (i) *effectively using generative artificial intelligence and large language models for engineering software systems* by both tailoring large language models to the specific needs of software engineering tasks and integrating models with domain knowledge and classic software engineering approaches, (ii) *creating high quality and large volume datasets and benchmarks for software engineering tasks*, while overcoming the challenges of data labeling, (iii) *shaping human-artificial-intelligence collaboration in software engineering teams* toward a fruitful collaboration among humans and artificial intelligence developers, by suitably interpreting artificial-intelligence-based decisions and recommendations, and defining the artificial intelligence and ethics code for software engineers to address the ethical challenges introduced by the transformative potential of artificial intelligence for SE. We summarize the main research directions as open questions that detail the four main challenges.

Artificial Intelligence and Large Language Models for Engineering Large Software Systems

A main challenge that we face when using artificial intelligence for engineering large software systems is the limited context length and inference speed of large language models that badly adapt to the large-volume system runtime data of AIOps. The most recent studies highlight important research directions; however, they focus on small problems and simple solutions, and do not scale to large problems and complex tasks.

We need to *effectively engineer prompts* and *combine them with domain specific knowledge* to address the huge variety of software engineering tasks. We need approaches to design prompts, study the impact of the different components of the prompts, and effectively integrate information from processes and artifacts to efficiently address the different phases of the software life cycle (*Open Challenge-1* in Section 2.1).

We need new *metrics* to objectively evaluate deep-learning enabled processes and products. We need strategies, taxonomies, and new metrics to evaluate the impact of both deep learning and large language models on the software process and products (*Open Challenge-2* in Section 2.2).

We need to *integrate deep learning and large language models with classic software engineering* for scalable and generalizable approaches to efficiently complete general and domain-specific software engineering tasks (*Open Challenge-3 in Section 2.3*).

We need *transparent and explainable approaches* to build trustworthy processes and artifacts, with humans and artificial intelligence jointly involved in decision making, to convey meaningful explanations to software developers for adopting artificial intelligence-generated code and artifacts, and incorporate human domain knowledge in ML models (*Open Challenge-4 in Section 2.4*).

The main challenges leave many open research questions:

RQ_A1. *What large language models will we see in software engineering?*

Deep learning, generative artificial intelligence and large language models dramatically upset software engineering research and practice, and impact on all software engineering activities, from requirements engineering to software production, software testing, software validation, software maintenance, and more.

What (generative) artificial intelligence will best fit the different software engineering activities, and how will artificial-intelligence-powered approaches differ within the software engineering activities and life cycle?

RQ_A2. *How will we integrate domain knowledge and mature software engineering techniques in large language models to handle software engineering tasks?*
Large language models perform well in general-purpose tasks. However, specialized knowledge is crucial in domain-specific applications to help large language models understand the context and background of the task, narrow the search space, and reduce misinterpretations.
How will we incorporate human domain knowledge and knowledge from different software engineering tasks and applications to define effective artificial-intelligence-powered approaches for software engineering?

Datasets and Benchmarks for Software Engineering

A main challenge that we face when training and fine-tuning artificial intelligence and large language models for software engineering tasks is the lack of unbiased datasets and benchmarks for software engineering tasks. Current studies rely on either small size or unrealistic synthetic datasets, which have been available on the Web for a long time now.

We need *large,reliable and unbiased datasets and benchmarks* and enough *domain knowledge* for training deep learning models and fine-tuning large language models for specific software engineering tasks (*Open Challenge-5 in Section 3.1*, *Open Challenge-6 in Section 3.2*, and *Open Challenge-7 in Section 3.3*).

The main challenges leave many open research questions:

RQ_B1. *What ultra-large-scaled dataset and benchmarks with domain and human knowledge will we create and use for artificial-intelligence-powered software engineering tasks?*
Large language models require huge and suitably labeled benchmarks that are representative of industrial-scale applications, are publicly available, and are not used for training general purpose *large language models*.
How will we address the contradicting confidentiality and open access requirements?

Human-artificial-intelligence Collaboration in Software Engineering Teams

A main challenge that we face when using generative artificial intelligence and large language models is trust in both the processes that involve artificial intelligence and large language models and the generated products. We need a new *code of ethics* to regulate unauthorized exploitation of data, allow independent and transparent audits of training datasets of large language models, and define the new software engineering profession.

The main challenges leave many open research questions:

RQ_C1. *How will humans collaborate withing hybrid human-artificial-intelligence teams?*
artificial intelligence will assist humans by accomplishing tedious, repetitive, effort-intensive and error prone activities. Artificial intelligence will collaborate with humans in teams, but will never completely throw humans out of software engineering teams.
How will we interpret artificial-intelligence-based decisions and recommendations to help humans understand the decision process and validate the artificial intelligence decisions?

RQ_C2. *What code of ethics should be applied/enforced for artificial-intelligence-based decisions and recommendations?*
The use of artificial intelligence for software engineering raises ethical challenges that involve privacy, responsible use of AI, confidentiality, trust and security.
How will we enforce a responsible use of AI, protect the confidentiality of data and personal information, and enhance trust in artificial-intelligence-powered processes and products? How will we prevent data

leaks, enforce transparency, and protect artificial intelligence-powered processes and products from security breaches?

RQ_C3. *How will generative artificial intelligence and large language models reshape the software engineering profession?*

Artificial intelligence will dramatically change the software engineering profession. It will improve the efficiency of software engineering processes, allowing engineers to focus more on creative problem solving and innovation.

How will pedatrsonalized artificial intelligence-powered development environments improve developer productivity and learning experience?

ACKNOWLEDGMENTS

This work is partially supported by the Swiss SNF project A-Test Autonomic Software Testing (200021_215487).

REFERENCES

[1] 2023. 7-reasons-why-software-testing-has-brighter-future-than-development. https://www.linkedin.com/pulse/7-reasons-why-software-testing-has-brighter-future-than-development-mz12f?trk=article-ssr-frontend-pulse_more-articles_related-content-card.

[2] 2023. Big Tech Struggles to Turn AI Hype Into Profits Microsoft, Google and others experiment with how to produce, market and charge for new tools. https://www.wsj.com/tech/ai/ais-costly-buildup-could-make-early-products-a-hard-sell-bdd29b9f

[3] 2023. The carbon footprint of GPT-4. https://towardsdatascience.com/the-carbon-footprint-of-gpt-4-d6c676eb21ae

[4] 2023. Impactful work: Helping developers around the world improve productivity with AI. https://aws.amazon.com/careers/life-at-aws-impactful-work-helping-developers-around-the-world-improve-productivity/

[5] 2023. *What Consumers Really Think About AI: A Global Study.* Retrieved November 18, 2023 from https://www.pega.com/ai-survey

[6] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1737–1749.

[7] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. arXiv:2302.10352

[8] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology* 176 (2024).

[9] Mohammad Alahmadi, Abdulkarim Khormi, Biswas Parajuli, Jonathan Hassel, Sonia Haiduc, and Piyush Kumar. 2020. Code localization in programming screencasts. *Empirical Software Engineering* 25 (2020), 1536–1572.

[10] Hadeel Alsolai and Marc Roper. 2020. A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology* 119 (2020), 106214. https://doi.org/10.1016/j.infsof.2019.106214

[11] Chetan Arora, John Grundy, and Mohamed Abdelrazek. 2023. Advancing requirements engineering through generative ai: Assessing the role of llms. *arXiv preprint arXiv:2310.13976* (2023).

[12] Sathurshan Arulmohan, Marie-Jean Meurs, and Sébastien Mosser. 2023. Extracting domain models from textual requirements in the era of large language models. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 580–587.

[13] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[14] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[15] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (, Toronto ON, Canada,) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 500–506. https://doi.org/10.1145/3545945.3569759

[16] James Bessen. 2019. Automation and jobs: When technology boosts employment. *Economic Policy* 34, 100 (2019), 589–626.

[17] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* 22, 3 (2023), 781–793.

[18] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. 2019. Studying the difference between natural and programming language corpora. *Empirical Softw. Eng.* 24, 4 (aug 2019), 1823–1868. https://doi.org/10.1007/s10664-018-9669-7

[19] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.

[20] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. Continuous incident triage for large-scale online service systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 364–375.

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

[22] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[23] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

[24] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2023. Empowering practical root cause analysis by large language models for cloud incidents. *arXiv preprint arXiv:2305.15778* (2023).

[25] Satyendra Singh Chouhan and Santosh Singh Rathore. 2021. Generative adversarial networks-based imbalance learning in software aging-related bug prediction. *IEEE Transactions on Reliability* 70, 2 (2021), 626–642.

[26] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. (2021). https://doi.org/10.1109/TSE.2021.3128234 arXiv:arXiv:2108.01585

[27] J. Constantz. 2023. Nearly a Third of White-Collar Workers Have Tried ChatGPT or Other AI Programs, According to a New Survey. *Time Magazine* (Jan. 2023). https://time.com/6248707/survey-chatgpt-ai-use-at-work/

[28] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 121–133. https://doi.org/10.1109/ICSE48619.2023.00022

[29] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[30] Giovanni Denaro, Rahim Heydarov, Ali Mohebbi, and Mauro Pezzè. 2023. Prevent: An Unsupervised Approach to Predict Software Failures in Production. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5139–5153. https://doi.org/10.1109/TSE.2023.3327583

[31] Ankur Desai and Atul Deo. 2022. Introducing Amazon CodeWhisperer, the ML-powered coding companion. https://aws.amazon.com/blogs/machine-learning/introducing-amazon-codewhisperer-the-ml-powered-coding-companion/

[32] Prem Devanbu, Matthew Dwyer, Sebastian Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Baishakhi Ray, Rishabh Singh, and Xiangyu Zhang. 2020. Deep Learning & Software Engineering: State of Research and Future Directions. arXiv:2009.08525 [cs.SE]

[33] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *ICSE 2022*. ACM. https://www.microsoft.com/en-us/research/publication/toga-a-neural-method-for-test-oracle-generation/

[34] Christof Ebert and Panos Louridas. 2023. Generative AI for Software Practitioners. *IEEE Software* 40, 4 (2023), 30–38. https://doi.org/10.1109/MS.2023.3265877

[35] European Commission. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). https://eur-lex.europa.eu/eli/reg/2016/679/oj

[36] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. https://doi.org/10.48550/ARXIV.2310.03533

[37] Carl Benedikt Frey and Michael A Osborne. 2017. The future of employment: How susceptible are jobs to computerisation? *Technological forecasting and social change* 114 (2017), 254–280.

[38] S. Gao, X. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 761–773. https://doi.org/10.1109/ASE56229.2023.00109

[39] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 39, 13 pages. https://doi.org/10.1145/3597503.3608134

[40] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to comment" translation" data, metrics, baselining & evaluation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 746–757.

[41] Jiazhen Gu, Jiaqi Wen, Zijian Wang, Pu Zhao, Chuan Luo, Yu Kang, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, et al. 2020. Efficient customer incident triage via linking with system incidents. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1296–1307.

[42] Guoxiang Guo, Aldeida Aleti, Neelofar Neelofar, and Chakkrit Tantithamthavorn. 2024. MORTAR: Metamorphic Multi-turn Testing for LLM-based Dialogue Systems. *arXiv preprint arXiv:2412.15557* (2024).

[43] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 34, 13 pages. https://doi.org/10.1145/3597503.3623306

[44] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 31.

[45] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. A survey on large language models: Applications, challenges, limitations, and practical usage. *Authorea Preprints* (2023).

[46] Shilin He, Xu Zhang, Pinjia He, Yong Xu, Liqun Li, Yu Kang, Minghua Ma, Yining Wei, Yingnong Dang, Saravanakumar Rajmohan, et al. 2022. An empirical study of log analysis at Microsoft. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1465–1476.

[47] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

[48] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[49] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. https://doi.org/10.48550/ARXIV.2308.10620

[50] Hui Huang, Yingqi Qu, Hongli Zhou, Jing Liu, Muyun Yang, Bing Xu, and Tiejun Zhao. [n. d.]. On the limitations of fine-tuned judge models for llm evaluation, 2024. *URL https://arxiv. org/abs/2403.02839* 3, 4 ([n. d.]).

[51] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2022. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering* 48, 1 (2022), 166–185. https://doi.org/10.1109/TSE.2020.2982385

[52] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[53] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[54] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*. PMLR, 5110–5121.

[55] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel C Briand. 2024. Impact of log parsing on deep learning-based anomaly detection. *Empirical Software Engineering* 29, 6 (2024), 139.

[56] Fatemeh Khayashi, Behnaz Jamasb, Reza Akbari, and Pirooz Shamsinejadbabaki. 2022. Deep Learning Methods for Software Requirement Classification: A Performance Study on the PURE dataset. *arXiv preprint arXiv:2211.05286* (2022).

[57] Nicholas Kroeger, Dan Ley, Satyapriya Krishna, Chirag Agarwal, and Himabindu Lakkaraju. 2023. Are Large Language Models Post Hoc Explainers? https://doi.org/10.48550/ARXIV.2310.05797

[58] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (jun 2020), 38 pages. https://doi.org/10.1145/3383458

[59] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*. 184–195.

[60] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[61] Mingyang Li, Ye Yang, Lin Shi, Qing Wang, Jun Hu, Xinhua Peng, Weimin Liao, and Guizhen Pi. 2020. Automated extraction of requirement entities by leveraging LSTM-CRF and transfer learning. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 208–219.

[62] Wei Li, Haozhe Qin, Shuhan Yan, Beijun Shen, and Yuting Chen. 2020. Learning code-query interaction for enhancing code searches. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 115–126.

[63] Zachary C. Lipton. 2018. The Mythos of Model Interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue* 16, 3 (jun 2018), 31–57. https://doi.org/10.1145/3236386.3241340

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

[65] Matthew Lodge. 2021. Software Testing Is Tedious. AI Can Help. *Harvard Business Review* (Feb. 2021). https://hbr.org/2021/02/software-testing-is-tedious-ai-can-help

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

[67] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. 2023. Estimating the carbon footprint of bloom, a 176b parameter language model. *Journal of Machine Learning Research* 24, 253 (2023), 1–15.

[68] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. 2023. Improving requirements completeness: Automated assistance through large language models. *arXiv preprint arXiv:2308.03784* (2023).

[69] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518. https://doi.org/10.1016/j.asoc.2014.11.023

[70] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

[71] Farhad Manjoo. 2023. It's the End of Computer Programming as We Know It. (And I Feel Fine.). *The New York Times* (June 2023). https://www.nytimes.com/2023/06/02/opinion/ai-coding.html

[72] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using Transfer Learning for Code-Related Tasks. arXiv:2206.08574 [cs.SE]

[73] Faseeha Matloob, Taher M. Ghazal, Nasser Taleb, Shabib Aftab, Munir Ahmad, Muhammad Adnan Khan, Sagheer Abbas, and Tariq Rahim Soomro. 2021. Software Defect Prediction Using Ensemble Learning: A Systematic Literature Review. *IEEE Access* 9 (2021), 98754–98771.

[74] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence* 267 (2019), 1–38. https://doi.org/10.1016/j.artint.2018.07.007

[75] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: graph-based vulnerability localization in source code. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 367, 18 pages.

[76] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. 2018. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing* 73 (2018), 1–15. https://doi.org/10.1016/j.dsp.2017.10.011

[77] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* 46, 2 (2020), 196–221. https://doi.org/10.1109/TSE.2018.2844788

[78] Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and Ángel Fernández-Leal. 2022. Human-in-the-loop machine learning: a state of the art. *Artificial Intelligence Review* 56, 4 (Aug. 2022), 3005–3054. https://doi.org/10.1007/s10462-022-10246-w

[79] David Nader Palacio, Alejandro Velasco, Nathan Cooper, Alvaro Rodriguez, Kevin Moran, and Denys Poshyvanyk. 2024. Toward a Theory of Causation for Interpreting Neural Code Models . *IEEE Transactions on Software Engineering* 50, 05 (May 2024), 1215–1243. https://doi.org/10.1109/TSE.2024.3379943

[80] Son Nguyen, Hung Phan, Trinh Le, and Tien N Nguyen. 2020. Suggesting natural method names to check name consistencies. In *Proceedings of the acm/ieee 42nd international conference on software engineering*. 1372–1384.

[81] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. 2020. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software* 163 (2020), 110538.

[82] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*.

[83] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VULGEN: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2527–2539. https://doi.org/10.1109/ICSE48619.2023.00211

[84] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2023. Open Science in Software Engineering: A Study on Deep Learning-Based Vulnerability Detection. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1983–2005. https://doi.org/10.1109/TSE.2022.3207149

[85] Curtis G Northcutt, Anish Athalye, and Jonas Mueller. 2021. Pervasive label errors in test sets destabilize machine learning benchmarks. *arXiv preprint arXiv:2103.14749* (2021).

[86] United States Copyright Office. 2023. Copyright Registration Guidance: Works Containing Material Generated by Artificial Intelligence. 16190 Federal Register, Vol. 88, No. 51.

[87] Adam Oliner and Jon Stearley. 2007. What Supercomputers Say: A Study of Five System Logs. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 575–594.

[88] David N. Palacio, Daniel Rodriguez-Cardenas, Alejandro Velasco, Dipin Khati, Kevin Moran, and Denys Poshyvanyk. 2024. Towards More Trustworthy and Interpretable LLMs for Code through Syntax-Grounded Explanations. https://doi.org/10.48550/arXiv.2407.08983 arXiv:2407.08983 [cs].

[89] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. 2021. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering* 27, 2 (Dec. 2021). https://doi.org/10.1007/s10664-021-10066-6

[90] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[91] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[92] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[93] Anjana Perera, Aldeida Aleti, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Burak Turhan, Lisa Kuhn, and Katie Walker. 2022. Search-based fairness testing for regression-based machine learning systems. *Empirical Software Engineering* 27, 3 (2022), 79.

[94] Florian Pudlitz, Florian Brokhausen, and Andreas Vogelsang. 2019. Extraction of system states from natural language requirements. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 211–222.

[95] Jonathan G Richens, Ciarán M Lee, and Saurabh Johri. 2020. Improving the accuracy of medical diagnosis with causal machine learning. *Nature communications* 11, 1 (2020), 3923.

[96] Everett M Rogers, Arvind Singhal, and Margaret M Quinlan. 2014. Diffusion of innovations. In *An integrated approach to communication theory and research*. Routledge, 432–448.

[97] K. Roose. 2022. The Brilliance and Weirdness of ChatGPT. *The New York Times* (Dec. 2022). https://www.nytimes.com/2022/12/05/technology/chatgpt-ai-twitter.htm

[98] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 287–298.

[99] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, Singapore, Singapore,) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 107–119. https://doi.org/10.1145/3540250.3549145

[100] Lin Shi, Mingzhe Xing, Mingyang Li, Yawen Wang, Shoubin Li, and Qing Wang. 2020. Detection of hidden feature requests from massive chat messages via deep siamese network. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 641–653.

[101] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Laura A Heymann, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. 2024. Developer Perspectives on Licensing and Copyright Issues Arising from Generative AI for Coding. *arXiv preprint arXiv:2411.10877* (2024).

[102] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Laura A Heymann, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. 2025. The ML Supply Chain in the Era of Software 2.0: Lessons Learned from Hugging Face. *arXiv preprint arXiv:2502.04484* (2025).

[103] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1609–1620. https://doi.org/10.1145/3510003.3510160

[104] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. 2019. Feature maps: A comprehensible software representation for design pattern detection. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 207–217.

[105] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938* (2023).

[106] Junfeng Tian, Wenjing Xing, and Zhen Li. 2020. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *Information and Software Technology* 123 (2020), 106289.

[107] Archana Tikayat Ray, Bjorn F Cole, Olivia J Pinon Fischer, Anirudh Prabhakara Bhat, Ryan T White, and Dimitri N Mavris. 2023. Agile Methodology for the Standardization of Engineering Requirements Using Large Language Models. *Systems* 11, 7 (2023), 352.

[108] Feichin Ted Tschang and Esteve Almirall. 2021. Artificial Intelligence as Augmenting Automation: Implications for Employment. *Academy of Management Perspectives* 35, 4 (Nov. 2021), 642–659. https://doi.org/10.5465/amp.2019.0062

[109] Dimitrios Tsoukalas, Nikolaos Mittas, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Dionysios Kechagias. 2024. Local and Global Explainability for Technical Debt Identification. *IEEE Transactions on Software Engineering* (2024), 1–15. https://doi.org/10.1109/TSE.2024.3422427

[110] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2Test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 299–303.

[111] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating Accurate Assert Statements for Unit Test Cases Using Pretrained Transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test* (Pittsburgh, Pennsylvania) *(AST '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3524481.3527220

[112] Lam Nguyen Tung, Steven Cho, Xiaoning Du, Neelofar Neelofar, Valerio Terragni, Stefano Ruberto, and Aldeida Aleti. 2024. Automated Trustworthiness Oracle Generation for Machine Learning Text Classifiers. *arXiv preprint arXiv:2410.22663* (2024).

[113] Alejandro Velasco, Aya Garryyeva, David N. Palacio, Antonio Mastropaolo, and Denys Poshyvanyk. 2025. Toward Neurosymbolic Program Comprehension. https://doi.org/10.48550/arXiv.2502.01806 arXiv:2502.01806 [cs].

[114] Alejandro Velasco, David N. Palacio, Daniel Rodriguez-Cardenas, and Denys Poshyvanyk. 2024. Which Syntactic Capabilities Are Statistically Learned by Masked Language Models for Code? *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 72–76. https://doi.org/10.1145/3639476.3639768

[115] Alejandro Velasco, Daniel Rodriguez-Cardenas, Luftar Rahman Alif, David N. Palacio, and Denys Poshyvanyk. 2025. How Propense Are Large Language Models at Producing Code Smells? A Benchmarking Study. https://doi.org/10.48550/arXiv.2412.18989 arXiv:2412.18989 [cs].

[116] Sanidhya Vijayvargiya, Lov Kumar, Lalita Bhanu Murthy, and Sanjay Misra. 2022. Software Requirements Classification using Deep-learning Approach with Various Hidden Layers. In *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE, 895–904.

[117] Xiaohui Wan, Zheng Zheng, Fangyun Qin, Yu Qiao, and Kishor S Trivedi. 2019. Supervised representation learning approach for cross-project aging-related bug prediction. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 163–172.

[118] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. Practitioners' Expectations on Code Completion. *arXiv e-prints* (2023), arXiv–2301.

[119] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* (2024), 1–27. https://doi.org/10.1109/TSE.2024.3368208

[120] Jun Wang, Li Zhang, Yanjun Huang, and Jian Zhao. 2020. Safety of autonomous vehicles. *Journal of advanced transportation* 2020, 1 (2020), 8867757.

[121] Simin Wang, Liguo Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2023. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 49, 3 (2023), 1188–1231. https://doi.org/10.1109/TSE.2022.3173346

[122] Wentao Wang, Nan Niu, Hui Liu, and Zhendong Niu. 2018. Enhancing automated requirements traceability by resolving polysemy. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 40–51.

[123] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2021. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. arXiv:2009.06520 [cs.SE]

[124] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3377811.3380429

[125] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

[126] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 334–345. https://doi.org/10.1109/MSR.2015.38

[127] Jonas Paul Winkler, Jannis Grönberg, and Andreas Vogelsang. 2019. Predicting How to Test Requirements: An Automated Approach. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 120–130.

[128] Skyler Wu, Eric Meng Shen, Charumathi Badrinath, Jiaqi Ma, and Himabindu Lakkaraju. 2023. Analyzing Chain-of-Thought Prompting in Large Language Models via Gradient-based Feature Attributions. https://doi.org/10.48550/ARXIV.2307.13339

[129] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. 2022. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems* 135 (2022), 364–381. https://doi.org/10.1016/j.future.2022.05.014

[130] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[131] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. https://doi.org/10.48550/ARXIV.2301.13246

[132] Mingxuan Xiao, Yan Xiao, Shunhui Ji, Yunhe Li, Lei Xue, and Pengcheng Zhang. 2025. ABFS: Natural Robustness Testing for LLM-based NLP Software. *arXiv preprint arXiv:2503.01319* (2025).

[133] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[134] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[135] Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei, Pinjia He, Dongmei Zhang, and Qi Zhang. 2025. OpenRCA: Can Large Language Models Locate the Root Cause of Software Failures?. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

[136] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 117–132.

[137] Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2022. Predictive models in software engineering: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–72.

[138] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–73.

[139] Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H. Chi, and Denny Zhou. 2024. Large Language Models as Analogical Reasoners. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=AgDICX1h50

[140] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th international conference on predictive models and data analytics in software engineering*. 62–71.

[141] Boxi Yu, Jiayi Yao, Qiuai Fu, Zhiqing Zhong, Haotian Xie, Yaoliang Wu, Yuchi Ma, and Pinjia He. 2024. Deep learning or classical machine learning? an empirical study on log-based anomaly detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–13.

[142] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 37, 12 pages. https://doi.org/10.1145/3597503.3623316

[143] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*. 219–229.

[144] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. 2024. Explainability for Large Language Models: A Survey. *ACM Trans. Intell. Syst. Technol.* 15, 2, Article 20 (feb 2024), 38 pages. https://doi.org/10.1145/3639372

[145] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2023), 46595–46623.

[146] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

[147] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).

Received 17 December 2024; revised 17 December 2024; accepted 17 February 2025