A 2030 Roadmap for Software Engineering

MAURO PEZZÈ, Università della Svizzera italiana, Lugano, Switzerland, Università degli Studi di Milano-Bicocca, Milan, Italy, and Constructor University, Schaffhausen, Switzerland
SILVIA ABRAHÃO, Universitat Politecnica de Valencia, Valencia, Spain
BIRGIT PENZENSTADLER, Chalmers University of Technology, Gothenburg, Sweden
DENYS POSHYVANYK, William & Mary, Williamsburg, VA, USA
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore, Singapore
TAO YUE, Beihang University, Beijing, China

The landscape of software engineering has dramatically changed in recent years. The impressive advances of artificial intelligence are just the latest and most disruptive innovation that has remarkably changed the software engineering research and practice. This special issue shares a roadmap to guide the software engineering community in this confused era. This roadmap is the outcome of a 2-day intensive discussion at the *2030 Software Engineering* workshop. The roadmap spotlights and discusses seven main landmarks in the new software engineering landscape: artificial intelligence for software engineering, human aspects of software engineering, software security, verification and validation, sustainable software engineering, automatic programming, and quantum software engineering. This editorial summarizes the core aspects discussed in the 37 papers that comprise the seven sections of the special issue and guides the interested readers throughout the issue. This roadmap is a living body that we will refine with follow-up workshops that will update the roadmap for a series of forthcoming ACM TOSEM special issues.

CCS Concepts: • **Software and its engineering → Requirements analysis**; **Software design engineering**; **Automatic programming**; **Software defect analysis**; **Formal software verification**; **Programming teams**; **Documentation**; **Software evolution**;

Additional Key Words and Phrases: A roadmap for software engineering, AI and software engineering, Human factor in software engineering, Automatic Programming, Sustainable software engineering, Quantum software engineering, AI for verification and validation, security and software engineering, generative AI for software engineering, Large language models for software engineering

Mauro Pezzè was supported by the Swiss National Foundation under grant SNF 200021_215487 (A-Test); Silvia Abrahão was supported by the State Research Agency under grant PID2022-140106NB-I00 (UCI-Adapt) and Generalitat Valenciana under grant CIAICO/2021/303 (AKILA); Birgit Penzenstadler thanks the Software Center for project #60 *Towards a sustainable future*; and Denys Poshyvanyk's research has been supported in part by the NSF CCF-234635, CCF-2311469, CNS-2132281, and CCF-1955853 grants. Abhik Roychoudhury's work was partially supported by a Singapore Ministry of Education (MoE) Tier3 grant MOE-MOET32021-0001; Tao Yue was supported by the State Key Laboratory of Complex & Critical Software Environment (SKLCCSE, grant No. CCSE-2024ZX-01) and the Fundamental Research Funds for the Central Universities.
Authors' Contact Information: Mauro Pezzè (corresponding author), Università della Svizzera italiana, Lugano, Switzerland, Università degli Studi di Milano-Bicocca, Milan, Italy, and Constructor University, Schaffhausen, Switzerland; e-mail: mauro.pezze@usi.ch; Silvia Abrahão, Universitat Politecnica de Valencia, Valencia, Spain; e-mail: sabrahao@dsic.upv.es; Birgit Penzenstadler, Chalmers University of Technology, Gothenburg, Sweden; e-mail: birgitp@chalmers.se; Denys Poshyvanyk, William & Mary, Williamsburg, VA, USA; e-mail: dposhyvanyk@wm.edu; Abhik Roychoudhury, National University of Singapore, Singapore, Singapore; e-mail: abhik@comp.nus.edu.sg; Tao Yue, Beihang University, Beijing, China; e-mail: taoyue@gmail.com.

This work is licensed under Creative Commons Attribution International 4.0.

© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/5-ART118
https://doi.org/10.1145/3731559

ACM Reference format:
Mauro Pezzè, Silvia Abrahão, Birgit Penzenstadler, Denys Poshyvanyk, Abhik Roychoudhury, and Tao Yue. 2025. A 2030 Roadmap for Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 118 (May 2025), 55 pages.
https://doi.org/10.1145/3731559

1 Introduction

The landscape of software engineering has undergone profound changes in recent years. Impressive advances in generative artificial intelligence are the most recent and dramatic innovations that redefine the world of software engineering. Generative artificial intelligence disrupts the research landscape with dramatic effects on software engineering research, education, and practice, as never before.

This special issue is the result of intensive 2-day discussions at the *2030 Software Engineering Workshop*, co-located with ACM SIGSOFT FSE (Foundations of Software Engineering) held on 15–16 July 2024, in Porto De Galinhas, Brazil. We invited the authors of 58 of the 76 papers submitted to the workshop to further discuss their ideas and extend their submissions for this special issue. The *liberating structure*[1] of the workshop fostered in-depth discussions among 62 participants through a carefully curated series of activities, including *impromptu networking*, *flash keynotes*, *1-2-4-All*, *shift and share*, *world café*, *fishbowl*, *open space technology*, *25/10 crowd sourcing*, and *conversation café* sessions[2] (see Figure 1[3]).

Following the ACM TOSEM peer review process, the authors thoroughly reviewed their papers based on the new ideas that emerged from the discussions at the workshop and the reviewers' comments on the workshop submissions. Ultimately, the ACM TOSEM Editorial Board selected 33 papers for this special issue. We completed the special issue with four *editors' papers* written by a subset of ACM TOSEM editors and peer reviewed by the ACM TOSEM editorial board. These papers address four hot topics: *Artificial Intelligence for Software Engineering* [33], *Software Engineering by and for Humans* [1], *Automatic Programming* [117], and *Software Security Analysis* [26].

Overview of the Editorial and the Special Issue

This editorial reflects the key discussions at the workshop that emphasize the disruptive impact of generative artificial intelligence on software engineering. Seven core open research areas emerged from the workshop, forming the structure of this special issue: artificial intelligence for software engineering, software engineering by and for humans, sustainable software engineering, automatic programming, security and software engineering, verification and validation, and quantum software engineering.

Artificial Intelligence for Software Engineering. The recent breakthrough in generative artificial intelligence triggers the deepest change in the skyline of software engineering research and practice since the Internet revolution in the second half of the last century. The software engineering community has never seen such a fast and predominant growth of new research threads, such as the application of artificial intelligence and machine learning in software engineering and

[1] https://www.liberatingstructures.com/.
[2] A special thank to *Daniel Russo*, who designed the program, and to *Matteo Ciniselli*, *Luca Di Grazia*, *Niccolò Puccinelli*, and *Ket artificial intelligence Qiu*, who coordinated the program on-site (Niccolò and Ketai) and off-site from Lugano (Luca and Matteo).
[3] More pictures at https://www.inf.usi.ch/faculty/pezze/se2030.html.

Fig. 1. Participants and discussions at the 2030 Software Engineering Workshop (the upper and lower half of the picture, respectively).

the challenges of engineering machine learning-driven systems, both of which have become the dominant themes of the main software engineering conferences and journals.

Section 2 of this editorial presents the main opportunities of generative artificial intelligence in software engineering and emphasizes the challenges of defining unbiased datasets and benchmarks shared between industry and academia, engineering reliable and stable prompts, integrating artificial intelligence tools with classic software engineering techniques, explaining the process and results of the artificial intelligence tool, and balancing effectiveness, efficiency, and ethics.

The 10 papers that comprise Section *Artificial Intelligence for Software Engineering*, the largest section of the special issue, offer a fish-eye view of opportunities and challenges. The *editors' paper* [33] presents a comprehensive overview of opportunities and challenges.

Terragni et al.'s [189], Qiu et al.'s [160], Burgueño et al.'s [25], and Kessel and Atkinson's [88] papers discuss the impact of artificial intelligence on the software engineering process. Terragni et al. [189] address the opportunities and challenges of integrating artificial intelligence into the software development process, emphasizing the risks of overreliance on artificial intelligence, the centrality of human judgment, the need to train the next generation of software engineers, and the importance of multidisciplinary collaborations between communities. Qiu et al. [160] present the current state and future trends of artificial intelligence-assisted programming, focusing on how artificial intelligence alters the roles of developers from manual coders to orchestrators of artificial intelligence-driven development ecosystems. Burgueo et al. [25] discuss the challenges of model-based software engineering with deep learning and a large language model. Kessel and Atkinson [88] propose semantic-aware training of generative artificial intelligence to check, synthesize, and modify software engineering artifacts.

He et al.'s [73], Zhao et al.'s [243], and Chen et al.'s [35] papers deal with the integration of artificial intelligence within multi-agent systems, large language model app stores, and large language models for mobile systems, respectively. He et al. [73] highlight the need of integrating large language models into multi-agent systems to enable autonomous problem-solving, improve

robustness, and provide scalable solutions for managing the complexity of real-world software projects. Zhao et al. [243] highlight the issues of large language model app stores, focusing on data mining, security risk identification, development assistance, and market dynamics, and discusses the relationships between stakeholders and technological advancements, with the focus on ethics and impacts on human society. Chen et al. [35] present challenges in improving mobile computing with large language models.

Lin et al.'s [245] and Gao et al.'s [62] papers complete the horizon with access issues and overall challenges. Lin et al. [245] discuss obstacles, opportunities, and challenges of the open source artificial intelligence-based software engineering solution to facilitate access to diverse organizational resources for open source artificial intelligence models, while ensuring privacy. Gao et al. [62] identify seven aspects of software engineering in the era of large language models and 25 related challenges.

Software Engineering by and for Humans. Machine learning, artificial intelligence, and autonomous systems are shaping a new landscape for software engineering by and for humans by radically changing even the basic concept of a software artifact. These evolving systems present new ethical, fairness, and technical challenges for software engineers. Humans are becoming an integral part of large software ecosystems, and the new role of humans in software systems calls for a shift in software engineering research from a narrow focus on users of software systems to a broader vision where humans are an integral part of cyber-physical ecosystems.

Section 3 of this editorial explores the impact of generative artificial intelligence on developers, software engineering teams, and *human–artificial intelligent agent* collaboration. The six papers that comprise the *Software Engineering by and for Humans* section of this special issue examine the roles of humans in the new landscape of software engineering in the era of generative artificial intelligence and highlight the challenges posed by hybrid *human–artificial intelligent agent teams*.

The *editors' paper* [1] explores various dimensions of *human–artificial intelligent agent* interactions and discusses the disruptive effects of artificial intelligence on software development, developer productivity, team dynamics, development tools, and the broader profession and education of software engineering. It also outlines the transition from **Graphical User Interfaces (GUIs)** to intelligent **Adaptive User Interfaces (AUIs)** and from No Operations to artificial intelligence for IT Operations, while anticipating new challenges in developing reliable, human-centric smart ecosystems, such as the next generations of smart cities.

Autili et al.'s [9], Mastropaolo et al.'s [123], and Souza et al.'s [51] papers discuss the challenges of engineering smart systems for humans. Autili et al. [9] address human and societal challenges in the design of smart digital systems, define proactive, reactive, and passive roles for human interaction, and explore the duality of trust and trustworthiness. Mastropaolo et al. [123] discuss the interplay between artificial intelligence-driven automation and human innovation. Souza et al. [51] introduce the concept and challenges of *fairness debt* in the development of smart digital systems, investigate the causes of fairness deficiencies in software development, and highlight their effects on individuals and communities.

Jackson et al. [79] and Hyrynsalmi et al. [78] focus on the impact of artificial intelligence on software engineers. While Jackson et al. [79] emphasize the enduring importance of human creativity in the era of generative artificial intelligence for software engineering, Hyrynsalmi et al. [78] discuss the risks and impact of generative artificial intelligence on diversity and inclusion in the field.

In summary, this special issue establishes a foundation for understanding and addressing the disruptive shift in human-centered software engineering in the coming decade.

Sustainable Software Engineering. The concept of sustainable development [24] extends beyond classic environmental concerns and spreads over software systems in cyber-physical spaces [148]. Sustainable software operations in cyberphysical spaces require new design, development, deployment, and maintenance approaches that minimize the ecological footprint, improve resource efficiency, and promote social responsibility [204].

Section 4 of this article presents the challenges and opportunities of sustainably engineering sustainable software systems for a sustainable world.

The five papers that comprise Section *Sustainable Software Engineering* of this special issue highlight the main challenges to sustainable software engineering. König et al.'s [97] and Betz and Penzenstadler's [17] papers discuss the challenges of engineering software for sustainability. König et al. [97] discuss the role of software engineering to address global sustainability and social inequality by reducing both the increasing consumption of resources and digital inequalities. Betz and Penzenstadler [17] highlight the great change in the role and responsibility of software engineers and discusses the social and environmental impacts of technology, with a focus on ethical and educational issues.

Shi et al.'s [182], Cruz et al.'s [45], and Moreira et al.'s [131] papers consider the environmental impact of software engineering. Shi et al. [182] highlight the main challenges of reducing the energy consumption of large language models for software engineering. Cruz et al. [45] discuss the impact of adopting environmentally friendly practices to create artificial intelligence-enabled software systems. Moreira et al. [131] focus on new curricula integrating sustainability into software development.

Automatic Programming. Machine learning, deep neural networks, and large language models are the largest magnitude factor of human productivity ever seen in software engineering since the early days. They open new frontiers towards automated programming, disrupt the quality and security scenario, and raise new societal and legal issues.

Section 5 of this editorial offers a fish-eye view of the impact of machine learning on programming, and highlights the different dimensions of generating, repairing, maintaining, and evolving code.

The four papers that comprise Section *Automatic Programming* of this special issue highlight the main challenges of automatic coding with machine learning. The *editors' paper* [117] discusses in detail the main challenges of automatically generating, repairing, maintaining, and evolving code with large language models, by emphasizing quality and trustworthiness. Robinson et al. [166] focus on the impact of large language models on both end-user software engineering and the software development lifecycle. Assunção et al. [8] highlight the transition from maintenance to modernization and discuss the challenges and opportunities of software modernization as a reengineering of entire legacy systems. Ran et al. [163] move beyond automatic programming and call for a strategic response to the anticipated formal method crisis with principled engineering methodologies.

Security and Software Engineering. The revolution in software production, the many emerging domains, and the enormous growth of software systems in both size and complexity open new security issues far beyond classic security engineering. The engineering of secure software systems is a key element of cybersecurity and opens many new challenges.

Section 6 of this editorial analyzes the core issue of cybersecurity in the era of artificial intelligence. It looks at the new challenges of assessing the security of automatically generated code and exploiting generative artificial intelligence to improve security.

The four papers that comprise Section *Security and Software Engineering* discuss the new challenges of analyzing security in the era of generative artificial intelligence. The *editors' paper* [26] discusses the challenges for assessing and maximizing the security of code co-written by machines,

defining approaches that work even if some functions are automatically generated, and tools that scale to an entire ecosystem. Zhou et al. [246] discuss the main challenges to improve vulnerability detection and repair using a large language model.

Williams et al.'s [219] and Wang et al.'s [211] papers discuss software supply chain security. Williams et al. [219] present the challenges of closing software supply chain attack vectors and supporting the software industry. Wang et al. [211] analyze the layers of the supply chain and discusses the challenges for robust and secure development with large language models.

Verification and Validation. Generative artificial intelligence offers powerful tools to enhance software verification and validation activities and the quality process, and challenges software engineers with the need to verify artificial intelligence-powered tools as well as artificial intelligence-powered smart ecosystems.

Section 7 of this editorial overviews the main challenges of verifying artificial intelligence engines and artificial intelligence-powered software, and of empowering validation and verification with artificial intelligence and machine learning.

The *editors' paper* [1] discusses the impact of generative artificial intelligence on developers and teams, as well as the new challenges of developing reliable, smart human-centric ecosystems. The six papers comprising this section deeply discuss the challenges and opportunities of verifying complex software systems with the aid of generative artificial intelligence.

Wang et al.'s [209], Li et al.'s [106], Molina et al.'s [129], and Cederbladh et al.'s [32] papers analyze the challenges of using large language models for verifying software systems. Wang et al. [209] discuss the challenges and opportunities of artificial intelligence-centric testing focusing on the interrelated dimensions of process, personnel, and technology. Li et al. [106] share the challenges and opportunities of using a large language model for metamorphic testing. Molina et al. [129] spotlight the role of large language models in automatically generating test oracles. Cederbladh et al. [32] discuss the need and challenges of new models for model-based early verification and identify six main challenges for early verification and validation that concern human factors (community and organization), automation (tools), and conceptualization (models, scope and methodology).

Birchler et al.'s [21] and Casadei et al.'s [29] papers discuss the challenges of verifying software-in-the-large context of cyberphysical systems. Birchler et al. [21] overview the challenges of simulation testing of cyberphysical systems; Casadei et al. [29] widen the discussion to large-scale cyberphysical systems.

Quantum Software Engineering. Quantum computing fundamentally reshapes the landscape of software engineering with new ways of developing and designing software. At the same time, quantum computing opens enormous opportunities to solve complex problems that currently challenge classic computing.

Section 8 of this editorial delineates the new landscape that emerges from quantum computing: Quantum software engineering and summarizes the state of the art, challenges, and future trends of quantum software engineering from the typical phases of the software development lifecycle, such as requirements engineering, architecture, modeling, programming, testing, and debugging.

Murillo et al.'s [134] and Ramalho et al.'s [162] papers that comprise Section 8 of this special issue dive into the key issues and present important challenges of quantum software engineering. Murillo et al. [134] provide an overview of the new scenarios that quantum computing enables, and discuss the impact of quantum computing on software engineering, and on verification and validation. Ramalho et al. [162] discuss the challenges and opportunities of quantum computing for software testing.

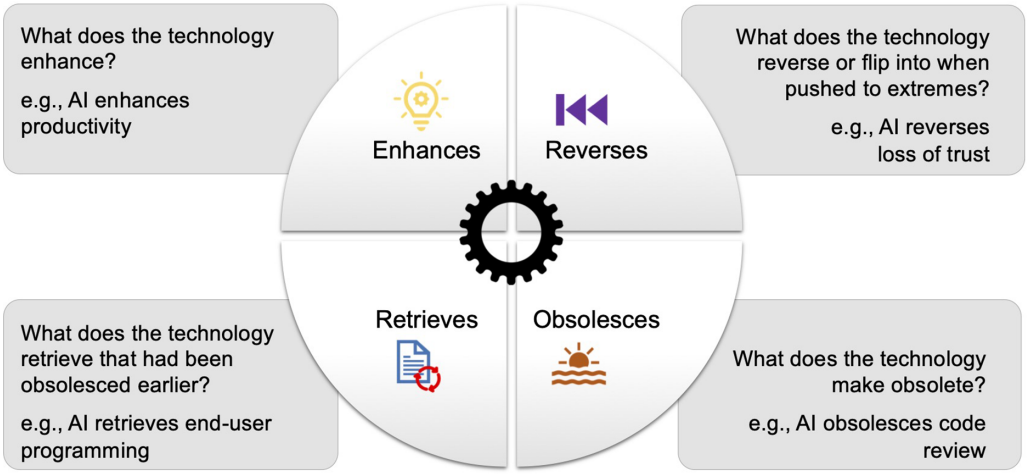

Fig. 2. Overview of McLuhan's tetrad.

Organization of the Sections

The following seven sections of this editorial (i) overview the state-of-the-art and practice, (ii) discuss the main Challenges and Trends, and (iii) propose a roadmap for the major research area. These sections illustrate the disruptive impacts of new technologies on software engineering using *McLuhan's tetrads* [124].[4] Figure 2 presents the four diamonds of McLuhan's tetrad, with an icon in the center representing the technology, and four diamond that indicate what the technology enhances, what it makes obsolete, what it retrieves from the past, and what it reverses and flips into when pushed to the extremes. Section 9 concludes the editorial with a comprehensive roadmap for software engineering, summarizing the key open research directions for the next decade.

2 Artificial Intelligence for Software Engineering

Modern software engineering evolves rapidly as artificial intelligence reshapes the development lifecycle. Traditional software engineering relies on structured code and predictable execution, using techniques such as program analysis, testing, and debugging. In contrast, artificial intelligence brings probabilistic, data-driven, and often opaque behaviors to the workflow [62, 73]. This transformation changes the traditional software development paradigms, not only in tooling and automation, but also in developer collaboration and decision-making [160, 189]. The shift to artificial intelligence reshapes the core assumptions of software engineering. Developers no longer just write and test code; they design prompts, interpret output, validate generated artifacts, and coordinate with autonomous agents. Qiu et al. [160] highlight how this evolution transforms developers into orchestrators who manage intelligent systems rather than simply composing instructions for machines. In this new context, artificial intelligence serves not just as tool, but as a collaborator–agent capable of writing, refactoring, and reviewing code.

Integrating artificial intelligence into software engineering improves developer productivity, enables tools that accelerate prototyping, and introduces artificial intelligence agents that transform the software lifecycle. He et al. [73] illustrate how large language model-based multi-agent systems enable distributed problem solving, pushing the boundaries of automation and scalability.

[4]The Canadian philosopher Herbert Marshall McLuhan defined the *Tetrad* in the mid-70s to illustrate the disruptive effect of new media and more generally new technologies and innovations in terms of abilities that the new technologies enhance, obsolesce, retrieve, and reverse, captured with the four diamonds of the tetrad.

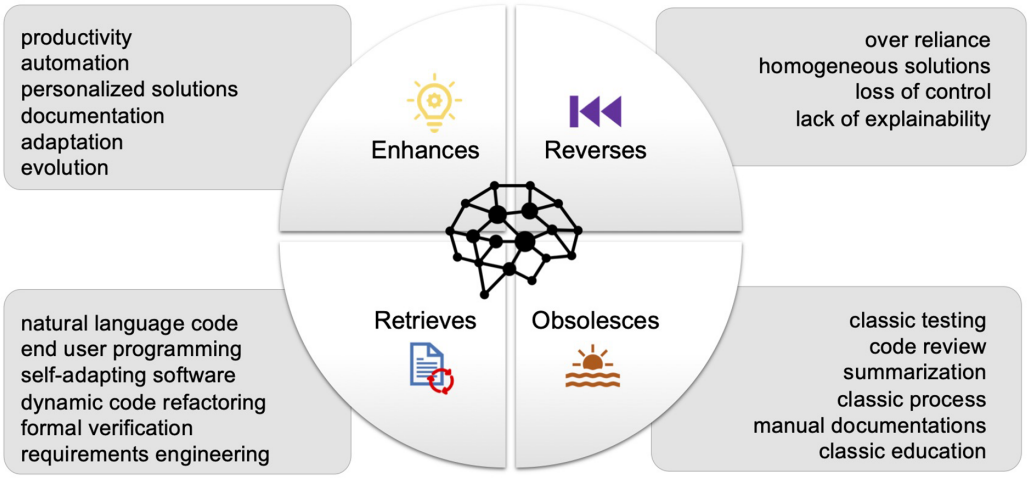

Fig. 3. The disruptive impact of artificial intelligence on software engineering.

Automation significantly improves software adaptability to complex environments and redefines the maintenance process.

This transformation obscures practices that depend on manual code inspection, fixed test oracles, and predictable behavior. Burgueno et al. [25] highlight the growth of artificial intelligence and encourage a fresh look at both the use of models and the importance of human collaboration, with trustworthiness being a critical factor. Terrangi et al. [189] warn against blind reliance on artificial intelligence tools, advocating for human oversight and critical thinking as essential elements to ensure safety, correctness, and ethical responsibility. Morescient et al. [88] argue that current large language model-based code models fall short in tasks requiring semantic understanding. Artificial intelligence reverses the assumption of software engineering on classic non-functional requirements (*for instance*, maintainability, performance, and scalability), by emphasizing the role of new requirements and concerns, such as explainability and fairness [62]. Zhao et al. [243] call for governance frameworks for emerging large language model app stores and their impact on society.

The McLuhan diagram in Figure 3 visualizes the disruptive impact of artificial intelligence on software engineering. Artificial intelligence dramatically *enhances* productivity by automating many tasks. It enhances documentation by improving and augmenting comments and documents. It enhances the adaptation and evolution of software systems and personalized solutions, with flexible solutions. Artificial intelligence *retrieves* natural language pseudo-coding and end-user programming. It retrieves self-adaptive and autonomic software systems, dynamic code refactoring, formal verification, and requirements engineering. Artificial intelligence *obsolesces* classic software engineering practice, process, and education, with a dramatic paradigm shift from coding to "prompting" and "validating." It obsolesces code review, summarization, and manual documentation by automating many activities. When pushed to the extremes, artificial intelligence *reverses* over-reliance in software systems, lack of control, and explainability, by hiding core computational aspects. It reverses homogeneous solutions by pushing human creativity out of the loop.

2.1 State of the Art and Trends

In this section, we explore the evolving landscape of artificial intelligence for software engineering by examining the challenges, tools, and emerging practices that shape the field. We bring together recent advances in prompt engineering, model evaluation, software lifecycle integration, and

explainability for software engineering. We present the state of the art and our vision for the current challenges and future roadmap.

2.1.1 Prompt Engineering. The effectiveness and reliability of large language models for code greatly depend on the prompts used to query them. The design of prompts to optimize these models is complex and poses a significant challenge across various application domains. Existing methods for prompt engineering, such as *Chain-of-Thought, ReAct, Tree-of-Thoughts*, and more elaborate approaches like *Retrieval-Augmented Generation*, provide in-context learning to guide the model answer. The current methods are primarily tailored for natural language processing tasks, and it is not yet clear whether these techniques can be effectively applied within the realm of software engineering, since the source code fundamentally differs from the natural language [30].

Challenges and Trends. Optimizing large language models for code through engineering prompts poses significant challenges across applications. In-context learning guides models for accurate answers and reduces the costly pretraining and fine-tuning for specific tasks. The successful use of zero-shot and one-shot learning to summarize the code [229, 235] and of conversational prompts for automatic program repair [222–224, 230] leads to new efficient prompting strategies in software engineering. The main advantage of in-context learning is the reduced cost of model training. It remains a challenge to confirm both the promising results of early studies on the composition of effective in-context learning demonstrations [63] and the benefits of design patterns and principles that minimize the impact of in-context learning on the performance of large language models. Recent research on prompts has demonstrated the feasibility of evaluating the effect of prompts on the generated code required [168] to detect meaningful prompts that enable the detection of code smells. Improving prompt design for code summarization and software engineering tasks requires careful refinement beyond simple instructions (*that is*, ask for generating a specific function, complete the code, and act as a programmer expert). A deeper analysis of different prompt components could improve their effectiveness, as recent studies have investigated the combination of multiple prompts to refine code and code translation [235].

2.1.2 Evaluation of Large Language Models. Current studies that explore the use of artificial intelligence for software engineering research remain in their early stages [226]. Existing metrics for assessing deep learning models in software engineering mainly focus on code generation, measurement accuracy—comparing model-generated code to manually designed code—and efficiency [31, 113, 240], measuring the time taken to generate code, often alongside traditional readability metrics and robustness metrics [210]. Few studies propose new automated testing methods to address the diverse scenarios of deep learning models in software engineering [70, 195, 225]. Many organizations either rely on third-party data labeling companies for manual labels [139] or use large language models as evaluators [244]. Manual dataset labeling can incur considerable costs and is prone to errors [139]. It is essential to define a standardized data pipeline to evaluate artificial intelligence models used in code and fairly compare different approaches. Software engineering requires a cohesive framework that facilitates the smooth integration of new metrics and scenarios while maintaining a uniform infrastructure. This framework would simplify the process from concept to hypothesis validation, it will reduce time and effort, and thus enhance the reliability of evaluations of the performance of deep learning-based approaches. A benchmark should encompass three key components with their challenges: the dataset, the metric, and the protocol used for the metric [167].

Challenges and Trends. Datasets are essential for both training and testing. Current datasets for training deep learning models collect data from open source platforms such as GitHub and Stack Overflow. These datasets often lack ethical considerations [185], exhibit varied quality, and are not

self-contained [37]. Many code examples rely on external modules and are poorly documented, making them difficult to understand or learn. Specific datasets are narrow in focus and capture a small fraction of the diverse scenarios one may face, resulting in biases [77]. We need robust and unbiased test oracles [11] to guarantee the quality and reliability of deep learning-based systems, including large language models.

Most currently available datasets focus on source code, input and output examples, repository metadata, and software programmer interviews [83]. We need specialized datasets, for instance, explaining and interpreting a model requires designing experiments to answer causal questions [167]. Most testing datasets also suffer from contamination, where benchmarks may unintentionally overlap with model training data. The contamination between training and evaluation datasets compromises the model evaluation. We need unbiased datasets and benchmarks.

Metrics are essential tools for evaluating models, and are either grounded in inherent properties or defined through comparative analysis of multiple models within empirical studies. Current metrics are limited in evaluating accuracy, precision, recall, and perplexity [37, 110, 111, 227]. We need new metrics to assess the inherent deep learning properties of software engineering, *for instance*, reasoning capacity, causal questioning, trustworthiness) [91].

The inadequate evaluation of deep learning systems can have significant consequences, including risks to patient well-being [165] and safety [208]. The metric depends on rigorous testing, which is currently limited by the lack of test oracles [11]. We need protocols that use reliable metrics and datasets to new automated testing methods evaluate a model and interpret results. The protocols shall outline the curation of the dataset, the purpose of the metrics, the evaluation properties, and the interpretation of the results. Correctly interpreting the evaluation of models involves the use of taxonomies about the presence and type of vulnerabilities and code smells [144].

2.1.3 Integrating Deep Learning with Traditional Software Engineering. Deep learning significantly influences every phase of the software engineering lifecycle, from requirements gathering to code generation, testing, and maintenance.

Software Requirements and Design. A few studies [190] exploit deep learning techniques to support the software design process, and focus on specific tasks such as design pattern identification [190] user interface detection [34, 130], requirement classification [92], extraction [105], traceability [212], validation [220], generation [192], and completeness enhancement [114]. Deep learning has yet to see widespread adoption in software requirements and design.

Generation of Software Source Code. Deep learning models accelerate and enhance the accuracy of complex coding tasks [122, 214, 234]. Generation tasks include code representation generation [85], code generation [39], code completion [41, 104, 217], code summarization [101], code comment generation [67], and method name generation [138]. Classification tasks involve code localization [3], which focuses on identifying source code within screencasts, as well as type inference [120], code search [107], and clone detection [216]. Empirical research suggests that the accuracy of Copilot-generated code depends on factors such as the programming language and the complexity of the task [46, 145, 236], and stress the need to evaluate the reliability of artificial intelligence-generated code, which is crucial in software engineering. Several studies examine the capabilities and limitations of code intelligence tools [28, 191], and conclude that current code intelligence tools excel in simple tasks and falter in complex tasks that require deeper semantic understanding [191].

Software Testing. Artificial intelligence is transforming software testing by automating test generation and improving fault detection. Research in this area has explored the use of large language models to generate test cases, showing promising results [19, 47, 108, 140, 178, 232]. However, current techniques do not guarantee compilable or executable test cases [238]. To improve

the reliability, quality, and fault detection effectiveness of generated tests, large language model-based test generation can be integrated with automated test generation tools such as Randoop [142], EvoSuite [59], and Pyguin [115], as presented by [102].

One key challenge in using large language models to automate software testing is the generation of effective test oracles that verify whether software behavior aligns with expected outcomes. Existing unit test generators primarily produce regression oracles based on implemented behavior rather than intended behavior, making them unsuitable for exposing faults in artificial intelligence-generated code [80, 181]. Studies indicate that generated test oracles often capture the actual behavior of the program instead of the expected behavior, limiting their effectiveness [76, 94].

Chen et al. [38] and Segura et al. [179] propose metamorphic testing as a promising solution to the oracle problem, by using metamorphic relations to infer expected behavior based on input–output relationships. Metamorphic testing is useful in this context when conventional test oracles are unavailable or difficult to specify. Recent research has used large language models to both automate the discovery of metamorphic relations and improve the effectiveness of metamorphic testing [6, 183, 194]. Despite these advances, the automatic generation of metamorphic relations remains a significant challenge. Some interesting approaches derive test oracles with neural networks [49, 53].

IT Operations. Artificial intelligence for IT operations involves leveraging artificial intelligence techniques to enhance IT operations. Research has focused on tasks such as anomaly detection [90, 237], incident classification [36, 68], and root cause analysis [228, 247]. Despite increasing attention, a significant gap between research and industry adoption, particularly with regard to goals, techniques, and practical challenges of log analysis, continues to widen [74].

Program Analysis. Program analysis—the automated evaluation of the features of a program, including correctness, robustness, and security, plays a critical role at various stages of the software lifecycle, such as optimization, validation, testing, debugging, comprehension, and maintenance. The increasing scale, complexity, and diversity of modern software systems pose several significant challenges to the effectiveness and assessment of artificial intelligence-assistant tools. The variety of programming languages and runtime environments hinders the advancement of analysis techniques, and analyzing dynamic programming languages yields incomplete results.

Challenges and Trends. Despite rapid advancements, deep learning in software engineering faces challenges that hinder its adoption. In software requirements and design, artificial intelligence struggles with ambiguity, domain-specific understanding, and incompleteness, affecting tasks such as requirement extraction, classification, and traceability [92, 105, 212]. Researchers have relied primarily on convolutional neural networks to explore these tasks, as the available data consist of images. Expanding research to incorporate retrieval-augmented generation techniques, transformer-based models, and decoder architectures could improve contextual awareness and artificial intelligence-driven software requirements and design.

The challenges associated with artificial intelligence in code generation encompass limited generalization across diverse datasets, which significantly impairs their effectiveness within various real-world contexts. This limitation exacerbates concerns about out-of-distribution instances and overfitting, as noted in previous studies [52]. In addition, issues related to the copyrightability and ownership of generated code are also pertinent [184]. The inability of artificial intelligence systems to synthesize and derive novel code from examples and descriptions restricts potential innovation. Although artificial intelligence tools well execute straightforward tasks, they exhibit difficulties when scaling to intricate and semantically nuanced assignments. We still need robust evaluation approaches to assess the quality, security, and maintainability of generated code.

Artificial intelligence-driven software testing faces key challenges in test case correctness, oracle reliability, and automation of metamorphic testing. Test cases generated with large language

models often fail to compile or run and need additional validation. The generation of test oracles remains a challenge, as large language models tend to capture actual rather than intended behavior. Although metamorphic testing addresses this issue through metamorphic relations, automating the generation of metamorphic relations remains a complex and open research problem. The most recent attempts to generate test oracles with deep neural networks indicate a promising research direction.

Artificial intelligence for IT operations faces efficiency constraints and prompt window limitations. Bug triangle, anomaly detection, and root cause analysis require context from multiple system components, often exceeding model capacities. Chain-of-thought reasoning improves interpretability, but generates excessive intermediate output, increasing the computational overhead. Artificial intelligence for IT operations requires both prompt engineering techniques to condense context with limited input sizes while preserving essential information and efficient models to process large-scale data.

2.1.4 Explainable Artificial Intelligence for Software Engineering. Interpretability is essential to ensure reliable artificial intelligence for software engineering, as it allows developers to understand and verify model decisions [91]. Explainable artificial intelligence [12] offers both knowledge-driven and data-driven interpretability approaches [96]. Knowledge-driven approaches generate explanations from domain knowledge and model-specific insights, while data-driven approaches rely directly on the data. Data-driven approaches include *intrinsic* and *post hoc* methods with either *local* or *global* scope.

Intrinsic interpretability approaches define inherently transparent models to directly comprehend the decision-making processes. Many software engineering tasks, like defect prediction [231] and effort estimation [71], rely on common intrinsic methods such as linear regression. *Post hoc* interpretability approaches such as Local Interpretable Model-agnostic Explanation [164] and Shapley Additive Explanations [116], widely adopted in defect prediction [81, 82], performance analysis [173], code analysis [177, 188], and code generation [112], explain the predictions after the model training and without altering the models.

Local approaches, like just-in-time defect prediction [158], line-level defect prediction to improve debugging [215], analysis of the quality of code [200], prediction of the retention of syntax knowledge [143, 199], and counterfactual explanations [43, 75] explain individual predictions. Global approaches, like sequential rationales to explain code completion tasks [196] and software analytics to support strategic decision-making [48], extend statistical findings from local explanations to explain the overall behavior of the model.

Post hoc explainability approaches badly handle multicollinearity, a common characteristic of source code, leading to misleading importance scores with inflated or distorted values for highly dependent features. Recent studies indicate that causal interpretability [135] can successfully reduce misleading interpretations by mitigating the influence of confounders.

Mechanistic interpretability dissects the internal mechanisms of complex models to infer the causal relations of operations. Recent studies indicate that sparse autoencoders can effectively detect regions in the activation spaces [56, 112], probes can explain how deep models encode structural information, by mapping hidden representations onto abstract syntax trees [119, 193], attention distributions can serve as interpretable signals to understand model decisions [128], the integration of neurosymbolic artificial intelligence can bridge the gaps of traditional explainability techniques while enhancing model transparency [198].

Challenges and Trends. The main challenges of explainability are multicollinearity among source code features and the granularity of the explanations. *Multicollinearity* among source code features,

which are often highly correlated, makes it difficult for interpretability techniques to isolate the true contribution of individual features.

Local interpretability provides insights specific to individual instances; however, it often lacks generalizability. Global interpretability captures overarching patterns across the model; however, it may oversimplify by overlooking important details in individual predictions. Most global approaches rely on correlational explanations that highlight statistical associations but do not reveal causal relationships, thus lacking the depth of understanding needed for critical tasks such as debugging or model refinement. Subjectivity plays an important role in how users perceive and evaluate explanations, and we need user studies to evaluate interpretability methods from a human-centered perspective.

2.2 Roadmap

- *How to develop domain-specific prompt design patterns that address the syntactic rigor, dependency management, and functional correctness requirements of code?* We need dataset snippets, APIs, and documentation to inform prompt design, as well as syntactic and semantic parsers to analyze code structures and extract meaningful patterns for prompting.
- *What critical components to create standardized and contamination-free benchmarks tailored to software engineering tasks?* We need ethically sourced [185] datasets spanning multiple languages, domains, and code complexities [40], as well as contamination data detectors [213] for fairly evaluating model.
- *How to reliably integrate deep learning in the software engineering lifecycle?* We need approaches that ensure model transparency, robustness, and alignment with engineering goals in various phases of the software engineering lifecycle, spanning from requirements design to code generation, testing, artificial intelligence for IT operations, and program analysis. We need continuous validation of these approaches through real-world testing to ensure the truthfulness of deep learning systems throughout the development process.
- *What interpretability techniques for plausible and feasible explanations of complex software engineering tasks?* We need methods that account for the multicollinearity inherent in software engineering artifacts, to generate explanations that align with developer expectations, by leveraging causal reasoning and hybrid interpretability (local and global).
- *What strategies to enhance the generalizability of explainability methods across various software engineering tasks and models?* We need task-agnostic interpretability approaches, supported by cross-domain benchmarks and user studies, to deliver consistent and trustworthy explanations in software contexts.

3 Software Engineering by and for Humans

Human–artificial intelligence interactions have a deep impact on software development, software engineering education, and society at large. While early research on software engineering for artificial intelligence and artificial intelligence for software engineering primarily focuses on technical aspects, there is now a broad agreement that humans play a crucial role in shaping the next generation of development approaches for artificial intelligence-powered systems.

The tetrad in Figure 4 illustrates the disruptive impact of human–artificial intelligence interaction on both the technical and social dimensions of software engineering. Human–artificial intelligence interaction *enhances* the automation of a wide range of software engineering tasks and the developer productivity by supporting a new generation of hybrid teams composed of humans and artificial intelligence-powered agents. It enhances creativity by acting as a catalyst for generating new ideas, expanding problem-solving approaches, and automating tedious tasks to free up mental space for innovation. It enhances developer experience by automating repetitive tasks, optimizing workflows,

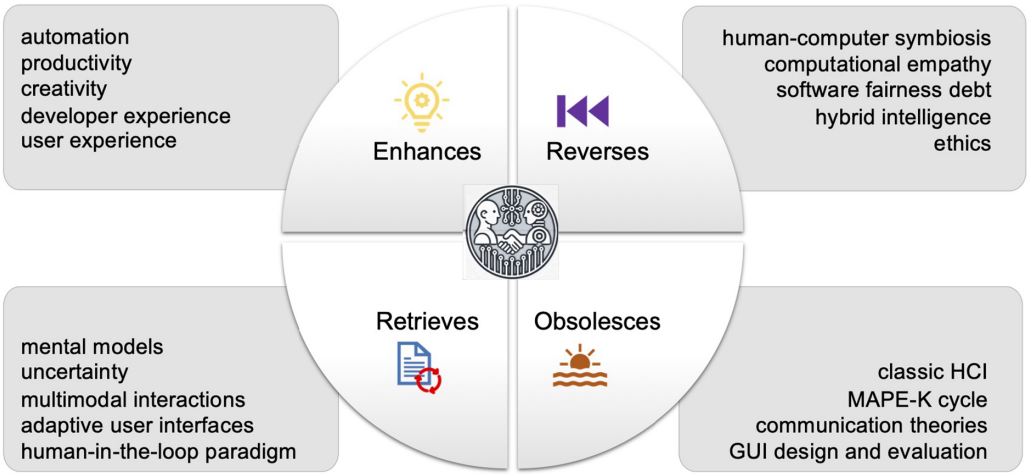

Fig. 4. The disruptive impact of human–artificial intelligence interaction in software engineering.

improving the quality of the code, and boosting productivity. It enhances user experience as artificial intelligence-powered systems have the potential to analyze user behavior, anticipate needs, and adapt user interfaces to provide seamless experiences.

Human–artificial intelligence interaction *retrieves* mental models and human-in-the-loop paradigms, by reinforcing continuous, interactive learning between artificial intelligence, developers, and users. When humans interact with an artificial intelligence system for a software engineering task, they build both a shared understanding and a mental model that gets seamlessly updated through the interaction. Human–artificial intelligence interaction retrieves multimodal interaction (*for instance*, speech, text, gestures, eye movement, facial expressions) and uncertainty to create more natural, reliable, and effective artificial intelligence systems. It is crucial to consider uncertainty in human–artificial intelligence interaction to ensure trust and safety in artificial intelligence-assisted decision-making. Humans shall trust artificial intelligence systems in high-risk situations when the uncertainty is high.

Human–artificial intelligence interaction also retrieves AUIs by reintroducing and significantly enhancing their core principles. The many human–computer interaction studies of AUIs, which dynamically adjust based on user behavior, context, and preferences, face important limitations due to rigid rule-based adaptation. Artificial intelligence can both inject prediction and personalization and enable real-time learning and continuous optimization.

Human–artificial intelligence interaction *obsolesces* traditional human–computer interaction and GUIs. Artificial intelligence-driven interfaces replace both static layouts and manual interactions with adaptive, multimodal, and conversational experiences, and reduce the relevance of conventional GUI design and evaluation methods. Human–artificial intelligence interaction introduces highly dynamic, interactive, and real-time adaptation, by reducing the relevance of the rigid Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) cycle. Artificial intelligence assistants continuously adjust to user behavior without waiting for an explicit "plan-execute" phase. Artificial intelligence systems learn dynamically from human interactions and allow humans to directly control adaptation. Human–artificial intelligence collaboration eliminates the need for rigid execution, that is, human–artificial intelligence systems co-adapt, where the artificial intelligence adjusts on-the-fly instead of following a rigid execution phase.

Human–artificial intelligence interaction challenges traditional communication theories, which were primarily designed for human-to-human interactions. Artificial intelligence introduces new dynamics that require a human-centric rethinking of communication and collaboration models that address the complexities of human-to-human, human-to-artificial intelligence, and artificial intelligence-to-artificial intelligence interactions.

When pushed to extremes, human–artificial intelligence interaction *reverses* hybrid intelligence—the synergy between human cognition and artificial intelligence—by either over-relying on artificial intelligence (leading to human obsolescence) or limiting artificial intelligence autonomy (hindering innovation). This dynamic recalls J.C.R. Licklider's concept of man–computer symbiosis [109], and advocates for balanced and iterative collaboration, where artificial intelligence agents and humans continuously learn from each other. Ethical concerns also intensify, pushing beyond classic human-centric software engineering towards cognitive and emotional fit—artificial intelligence's ability to understand user intent and provide empathetic responses. We need to ethically design computational empathy to avoid manipulation. Software fairness debt [51], *that is*, accumulated bias and unfairness in decision-making software systems, can flip in two extremes: artificial intelligence may either overcorrect bias and cause reverse discrimination or amplify existing biases and reinforce systemic inequalities. It is crucial to both define transparent and adaptive artificial intelligence systems with human oversight to ensure fairness without distortion and to integrate fairness management in the lifecycle of the artificial intelligence system.

3.1 State of the Art and Practice

3.1.1 Software Engineering by Humans. Generative artificial intelligence is profoundly transforming software engineering practices. It brings back human-centric computing, where instructions are given in natural language rather than strict programming syntax. This shift enables developers to focus on higher-level abstractions and problem-solving rather than low-level code implementation. Artificial intelligence-powered tools are reshaping software development not only from a technical standpoint but also in terms of human collaboration, roles, and team dynamics.

Several studies leverage artificial intelligence to automate repetitive tasks such as coding, debugging, and refactoring to *improve developer productivity and well-being*: Artificial intelligence can improve efficiency and reduce frustration by fixing errors, recommending best practices, and generating documentation [214]. Artificial intelligence lets developers focus on creative problem-solving. Mastropaolo et al.'s paper [123] in this special issue discusses the interaction between artificial intelligence-driven automation and human innovation, emphasizing the need for seamless integration of artificial intelligence while preserving human creativity. The article outlines key elements vital for this integration, aiming to advance software engineering methodologies and standards in the era of artificial intelligence.

Artificial intelligence *shifts traditional roles and responsibilities*. Developers are no longer creators, but also curators; they shift from manually writing code to reviewing, refining, and steering artificial intelligence-generated outputs. This shift requires strong critical thinking skills. The rise of artificial intelligence in software development shapes new roles, such as the *artificial intelligence whisperers*, developers skilled in prompt engineering, fine-tuning artificial intelligence suggestions, and understanding artificial intelligence biases to maximize the potential of the technology. Artificial intelligence-powered tools foster *collaborative and inclusive development teams*. Artificial intelligence assists team members with little coding experience and enables domain experts to contribute more directly to software development. Artificial intelligence tools that provide natural language explanations enhance cross-functional collaboration by making code and technical decisions understandable to non-software-engineering-experts.

The integration of artificial intelligence in software development creates *new team dynamics and workflows*. Teams incorporate artificial intelligence-powered tools early in the development process to accelerate both early prototyping and iteration cycles. Artificial intelligence increases the need of code review and ethics. The presence of hidden biases and inefficiencies in artificial intelligence-generated code highly emphasizes the need of explainability, security audits, and ethical considerations to ensure high-quality software development. Generative artificial intelligence deeply impacts on software development practices that focus on humans. Several studies define principles and strategies to proactively identify and mitigate biases, and ensure fairness, accountability, and trustworthiness in artificial intelligence-powered systems [51]. Other initiatives focus on promoting inclusivity and equity within software development processes [78].

3.1.2 Software Engineering for Humans. Autili et al.'s paper [9] in this special issue explores the societal and human impacts of autonomous software technologies. The authors emphasize the need to integrate human, societal, and environmental values into digital system engineering. They identify four key challenges based on human interaction roles: the *proactive role* (humans initiate actions, shaping system behavior), the *reactive role* (humans respond to system-generated events), the *passive role* (humans experience system decisions without direct interaction), and the *duality of trust and trustworthiness* (balancing human trust in systems with their reliability and ethical behavior). To address these challenges, the article outlines a research roadmap focusing on development processes, requirements engineering, software architecture, and verification, ensuring digital systems align with ethical and societal needs for long-term sustainability and well-being.

The field of human–computer interaction has a long tradition of creating and applying design principles or heuristics for assessing and improving user experience. Sun et al. [186] present a general framework for human–artificial intelligence interaction, and focus on the alignment between artificial intelligence systems, human users, and specific tasks. Sun et al. emphasize two key aspects: how well artificial intelligence fits human needs and capabilities (human–artificial intelligence fit) and whether artificial intelligence is suitable for the tasks it performs (Task–AI fit). Sun et al. also introduce the concept of human–artificial intelligence collaboration continuum, which accounts for varying degrees of artificial intelligence agency in interactions. This continuum ranges from scenarios where artificial intelligence serves as a passive tool under human control to situations where artificial intelligence acts as an autonomous agent making independent decisions. By integrating these elements, the framework provides a structured approach to analyze and design Human–artificial intelligence interactions, emphasizing the importance of compatibility and collaboration dynamics to enhance user experience and system performance.

Some papers propose guidelines for both designing artificial intelligence systems and understanding human–artificial intelligence interactions. Amershi et. al. [7] propose 18 guidelines for human–artificial intelligence interaction grouped into four categories that guide the different interaction stages: initially (make clear what the artificial intelligence system can do), during the interaction (make the system processes transparent), when the interaction is wrong (help users recover), and over time (foster user trust and improve the system). These guidelines emphasize usability, transparency, error recovery, and adaptability, ensuring that artificial intelligence systems support and empower users rather than frustrate or confuse them.

3.1.3 Challenges and Trends. As artificial intelligence becomes more integrated into human workflows, several challenges emerge alongside future trends shaping the next generation of human-centered approach to human–artificial intelligence interaction in software engineering.

3.1.4 Developer Productivity, Experience, and Creativity. The *editors' paper* [1] in this issue explores future research directions on developer productivity, experience, flow, and creativity,

and discusses the impact of artificial intelligence on software development tools, by highlighting opportunities and challenges across different tool categories throughout the software development lifecycle. Jackson et al.'s paper [79] in this special issue presents a research agenda that addresses the impact of generative artificial intelligence on creativity in software development, and propose six interconnected themes: individual capabilities, team capabilities, product, social impact, and human aspects. Jackson et al. emphasize that human creativity will play a crucial role in maintaining a competitive advantage in software development, as generative artificial intelligence integrates into the developer toolchain and practice.

3.1.5 Developer Diversity and Inclusion. Hyrynsalmi et al.'s paper [78] in this special issue explores the key challenges and the research opportunities for advancing software developer diversity and inclusion. Hyrynsalmi et al. propose a research roadmap that guides both researchers and practitioners in creating more inclusive software development environments, with a focus on maximizing benefits while minimizing harm, particularly for vulnerable groups. Hyrynsalmi et al. examine the relationship between artificial intelligence and software developers' diversity and inclusion, and highlight how artificial intelligence can both support and hinder diversity in software development teams. Although artificial intelligence-driven tools can enhance productivity, they may also reinforce existing biases if not carefully managed. Hyrynsalmi et al. stress the importance of implementing proactive measures to ensure that artificial intelligence technologies contribute to greater inclusivity and equity in software engineering.

3.1.6 Collaboration Practices and Hybrid Human–Artificial iIntelligence Teams. The *editors' paper* [1] in this issue discusses a number of challenges that arise from evolving collaboration practices in software development in the artificial intelligence era. Teams are becoming increasingly hybrid in terms of both distribution and integration of artificial intelligence-powered development agents alongside human developers. The article examines the impact of artificial intelligence-powered development tools and agents on team dynamics and developer-stakeholder interactions, and observes the importance of an active role of artificial assistants for truly effective hybrid human–-artificial intelligence teams. The key success factor of hybrid human–artificial intelligence teams lies in both balancing automation with human oversight and ensuring that artificial intelligence increases productivity without compromising creativity, security, or ethical considerations:

— *Artificial Intelligence as a Code Collaborator*: artificial intelligence-powered co-programmers analyze tradeoffs, refactor autonomously, and propose architectural improvements beyond simply suggesting code as done by current artificial collaborators, *for instance*, GitHub Copilot, Code Llama.
— *Artificial Intelligence in Pair Programming*: artificial collaborators as partners in pair-programming teams, learning and adapting to developers' coding style and project-specific patterns.
— *Artificial Intelligence in Code Reviews and Quality Assurance*: artificial collaborators provide rationale, learn from human feedback, and continuously improve the quality of the review process, beyond simply flagging syntax errors.
— *Artificial Intelligence as an Agile Team Member:* artificial collaborators as active members of agile teams, tracking project progress, predicting sprint outcomes, and identifying potential bottlenecks.

3.1.7 Human-Centered Approaches to Generative Artificial Intelligence in Software Engineering. Russo et al.'s Copenhagen Manifesto [170] advocates for a human-centered approach to integrate generative artificial intelligence in software engineering. The Copenhagen Manifesto emphasizes ethical responsibility, transparency, fairness, and social well-being, and urges that artificial

intelligence-driven software engineering enhances rather than diminishes human capabilities. The implementation of the Copenhagen Manifesto faces several challenges:

- *Ethical and Regulatory Uncertainty*: Defining *universal ethical standards* is difficult due to regional and cultural differences. Artificial intelligence regulations are constantly evolving, making it difficult for software developers to ensure artificial intelligence-powered systems align with ethical guidelines and legal requirements. We advocate for a *global artificial intelligence ethics framework* and foster *collaboration with policymakers* to establish clear guidelines.
- *Balancing Human-Centered Design with Artificial Intelligence Efficiency*: Prioritizing human oversight and values can slow down artificial intelligence-driven automation and reduce efficiency. Ensuring fairness and inclusivity in artificial intelligence models requires significant computational resources and data curation, thus increasing costs. We advocate for hybrid artificial intelligence–human workflows, where artificial intelligence automates routine tasks but keeps humans in control of critical decisions.
- *Artificial Intelligence Bias and Fairness*: Artificial intelligence bias remains a major issue, as artificial intelligence models may still reflect human prejudices embedded in training data. Ensuring fairness and transparency in artificial intelligence decisions requires *continuous monitoring and auditing*, which can be resource-intensive. Souza et al.'s paper [51] in this special issue identifies some key causes of fairness deficiencies in software development and examines their negative impact on individuals and communities, including discrimination and the perpetuation of inequalities. Souza et al. propose a socio-technical roadmap with six key goals to build equitable and socially responsible artificial intelligence-driven software systems: bridging the gap between research and real-world applications, developing a framework for fairness debt, equipping practitioners with tools and knowledge, improving bias mitigation, integrating fairness tools into industry practice, and enhancing explainability and transparency in artificial intelligence systems.
- *Measuring Human-Centered Artificial Intelligence Success:* Defining clear *metrics for fairness, explainability, and user well-being* remains an open challenge. The impact of human-centered artificial intelligence approaches is difficult to quantify. We need standardized assessment frameworks that track fairness, usability, and trust in artificial intelligence-driven systems.

Although the Copenhagen Manifesto [170] sets a vision for ethical artificial intelligence in software engineering, its success depends on clear regulations, technical advances, cultural shifts, and practical measurement strategies. Overcoming these challenges requires collaboration between artificial intelligence and software engineering researchers, software engineers, and policymakers.

3.1.8 Personalization and Context Awareness. Personalization and context awareness will transform software engineering by making artificial intelligence-centered systems intelligent, proactive, and user-centric. Artificial intelligence-centered systems will adapt in real time based on user preferences, past behavior, and emotional state. Personalization and adaptability require artificial intelligence to access sensitive user data (*for instance*, behavioral patterns, coding habits, and work preferences), raising significant privacy concerns. Privacy-preserving artificial intelligence techniques, such as federated learning, will play a crucial role in ensuring privacy while maintaining personalization.

3.1.9 Emotional Intelligence and Computational Empathy. Computational empathy refers to the ability of artificial intelligence systems to recognize, interpret, and respond to human emotions in a way that mimics empathetic behavior. Pataranutaporn et al. [146] highlight both the importance of initial user perceptions in shaping human–artificial intelligence interactions and the importance of emotion-aware artificial intelligence systems to improve both user experience and developer

well-being. Artificial intelligence systems still struggle with emotional understanding, often leading to unnatural or inappropriate responses, as evident in current artificial intelligence-driven customer support systems, mental health chatbots, and virtual assistants. Artificial intelligence systems shall enhance emotional recognition and contextual awareness to ensure natural and empathetic interactions.

3.1.10 Evolving Software Engineering Approaches to Support Human–Artificial Intelligence Interaction. The adaptive behavior of software systems has largely increased and has become extremely critical. The traditional *MAPE-K* cycle that is widely used in self-adaptive systems [87] does not adequately cope with the dynamic, interactive, and real-time adaptivity of human-centric artificial intelligence systems. Software engineering can cope with the new adaptive behavior with a paradigm shift from the fixed MAPE-K adaptation cycle to real-time and continuous learning that seamlessly adjusts based on user behavior, and from a system-driven approach to a human-in-the-loop model, where users actively guide learning and decision-making. The new paradigm shall leverage deep learning and reinforcement learning to support end-to-end, self-improving processes.

The shift of artificial intelligence systems from rigid to human–artificial intelligence-co-guided adaptation, from reactive monitoring to proactive, anticipatory adaptation, and from rule-based and reactive responses to flexible probabilistic artificial intelligence models that continuously refine outputs requires rethinking software engineering approaches to accommodate the dynamic nature of human–artificial intelligence interactions, and ensure adaptive, transparent, and user-centric artificial intelligence systems.

Cleland-Huang et al.'s MAPE-K-HMT framework [44] moves towards co-guided adaptation with a structured method for designing systems for effective human-autonomous-systems collaboration that enhances hybrid human–machine teamwork. Several recent artificial intelligence-driven self-adaptive models, like Digital Twin-Driven Adaptation [233], Federated Learning for Decentralized Adaptation [161], and Explainable artificial intelligence-Driven Self-Adaptation [175], enhance real-time, decentralized, and intelligent adaptation. However, all the models proposed so far are tailored to specific domains, such as artificial intelligence-powered code assistants, autonomous cloud resource management, artificial intelligence-driven cyber-physical systems, and adaptive security systems, and suffer from severe limitations that include high computational costs, model accuracy constraints, data privacy risks, and challenges in handling heterogeneous data. The next generation artificial intelligence-driven self-adaptive systems shall integrate *hybrid intelligence* [2] that combines symbolic artificial intelligence, neural networks, and human feedback, to create transparent, reliable and ethically aligned adaptation models.

3.1.11 Effectively Supporting Human–Artificial Intelligence Interaction. Amershi et al.'s guidelines [7] lay the foundation for designing artificial intelligence systems that effectively support humans, and open several software engineering challenges:

— *Complexity in Implementing Guidelines*: Many of the guidelines, such as *making artificial intelligence processes transparent* and *enabling user feedback*, require sophisticated artificial intelligence models that can *explain their decisions*. This is particularly difficult for deep learning systems, which often function as black boxes. Designing *adaptive artificial intelligence behavior* that learns from user interactions without introducing biases or errors is highly complex.
— *Tradeoffs between Transparency and Usability*: On one side excessive *system transparency* (*for instance*, exposing detailed decision-making processes) can overwhelm users with unnecessary information, leading to *cognitive overload*, on the other side *simplified and opaque* artificial intelligence systems limit trust and lead to *miscommunications* about the system capabilities.

— *Error Recovery and User Control*: On one side, it is crucial to allow humans to *correct artificial intelligence errors*, on the other side it is difficult to design effective error recovery mechanisms without disrupting the human experience. In some high-stakes domains (*for instance*, healthcare, finance), humans may *lack the expertise* to identify or correct artificial intelligence errors.
— *Balance Artificial Intelligence Assistance and Over-Reliance*: Excessive automation and guidance can lead to overreliance and can reduce human oversight and critical thinking. An excessive dependence on artificial intelligence recommendations may lead humans to trust incorrect outputs without verification.
— *Diverse Human Needs*: Humans use artificial intelligence systems with varying levels of technical expertise, making it difficult to design *universal artificial intelligence interaction models* that cater to novices and experts.
— *Continuous Artificial Intelligence Evolution and Maintenance*: Artificial intelligence models and humans evolve over time, and artificial intelligence systems shall be *continuously updated.* Implementing human feedback loops can be resource-intensive and may require ongoing *human oversight* to ensure artificial intelligence systems to remain effective and ethical.

Mitigating the challenges of effectively supporting human–artificial intelligence interactions requires to suitably combine technical solutions, best practices of design, and ethical considerations. Many strategies address specific challenges in human–artificial intelligence interaction. Barredo Arrieta et al. [12] propose both explainable artificial intelligence that mitigates the complexity of implementing the guidelines with insights into how artificial intelligence systems make decisions, and a modular approach for designing transparency, adaptability, and error recovery as independent components of artificial intelligence systems. Selective transparency is an interesting tradeoff between transparency and usability, *for instance*, *layered explanations* enable human to access *high-level summaries* by default and *detailed technical explanations* on demand, and to *adjust the level of artificial intelligence transparency* based on human expertise and needs. Personalization and AUIs [176] adjust to different expertise and needs, by offering simplified artificial intelligence interactions to new users and detailed control and transparency to experts, and by provide customization settings to fine-tune the behavior of the artificial intelligence system. While there are several strategies to address specific challenges in human–artificial intelligence interaction, the integration of the different strategies into artificial intelligence systems that fully align with human-centered principles and guidelines is still an open challenge.

3.2 Roadmap

— *How to design effective collaboration practices for hybrid human–artificial intelligence teams?* We need new frameworks that seamlessly integrate human and artificial intelligence teams, by defining clear roles, responsibilities, and decision-making boundaries, and ensuring mutual trust and explainability with adaptive interfaces that facilitate intuitive interactions. We need metrics to assess team efficiency, artificial intelligence reliability, and user satisfaction in hybrid collaboration settings.
— *How to develop equitable and socially responsible artificial intelligence-powered software systems?* We need to integrate heterogeneous datasets, bias detection tools, and explainable artificial intelligence approaches into a development process of artificial intelligence systems to proactively identify and mitigate biases.
— *How to balance transparency, user control, privacy, and adaptability, to ensure that artificial intelligence systems enhance human capabilities rather than hinder them?* We need approaches

that balance transparent decision-making, human control mechanisms, robust privacy protections, and adaptive capabilities that align with human needs, to design artificial intelligence systems to empower users while maintaining ethical safeguards.

— *How to design artificial intelligence systems that incorporate computational empathy?* We need models that recognize, interpret, and respond to human emotions to create artificial intelligence systems that interact naturally and effectively with humans. We need advances in affective computing, sentiment analysis, user-centered design, and software engineering to design artificial intelligence systems that ethically adapt to emotional and contextual cues.

— *How to ensure artificial intelligence systems meet evolving human needs?* We need models of human–artificial intelligence interaction to effectively capture the dynamics of human–artificial intelligence evolution. We need to borrow and integrate communication theories from diverse disciplines to develop new abstractions. We need clear metrics to assess the effectiveness of human–artificial intelligence interactions. We need software engineering methodologies that accommodate the dynamic and interactive nature of human–artificial intelligence collaboration.

4 Sustainable Software Engineering

We have been well aware of the limits of a planet with finite resources in juxtaposition with the promise of infinite economic growth since the early 1970s [125]. The limitations of the finite resources of the planet are well discussed in the 1972 Brundtland[5] report of the UN World Commission on Environment and Development [24]. The report proposes the most commonly referred to definition of *sustainable development* as *development that meets the needs of the present without compromising the ability of future generations to meet their own needs*. However, despite great policy efforts (for example, the Millennium Development Goals and the Sustainable Development Goals [171] signed by many nations) and our considerable progress [172], we still struggle to change that paradigm and effectively change our relationship with the planet [127]. Halkos and Gkampoura's sustainable development paper [72] claims that we are close to meet economy-related targets, such as the targets included in*decent work and economic growth* (Sustainable Development Goal 8), *industry, innovation and infrastructure* (Sustainable Development Goal 9), and *responsible consumption and production* (Sustainable Development Goal 12), while we need to accelerate to meet the targets and goals that concern *education* (Sustainable Development Goal 4), *cities and communities' sustainability* (Sustainable Development Goal 11), and *climate change* (Sustainable Development Goal 13).

Sustainable software engineering is a key research area in software engineering that encompasses (i) a reflection on principles of software design for a positive impact in the world, (ii) implemented in systems that can be maintained over an extended period of time, and (iii) with methods and practices that favor the inclusion of systems thinking to identify potential second and third order impacts.

This research area has become crucial for society over the last ten plus years for a number of reasons: (i) the footprint of IT systems, especially including second- and third-order effects, is ever increasing despite a large body of work on energy efficiency, (ii) unpredictable natural disasters have become more frequent and larger in scale and impact, and (iii) the recent rapid development of artificial intelligence makes balancing interventions and approaches that question techno-solutionism ever more important.

Sustainable software engineering includes *sustainability in software engineering*, that is, making software engineering itself a sustainable practice with sustainable systems, often with an emphasis

[5]Norwegian minister Gro Harlem Brundtland who chaired the commission.

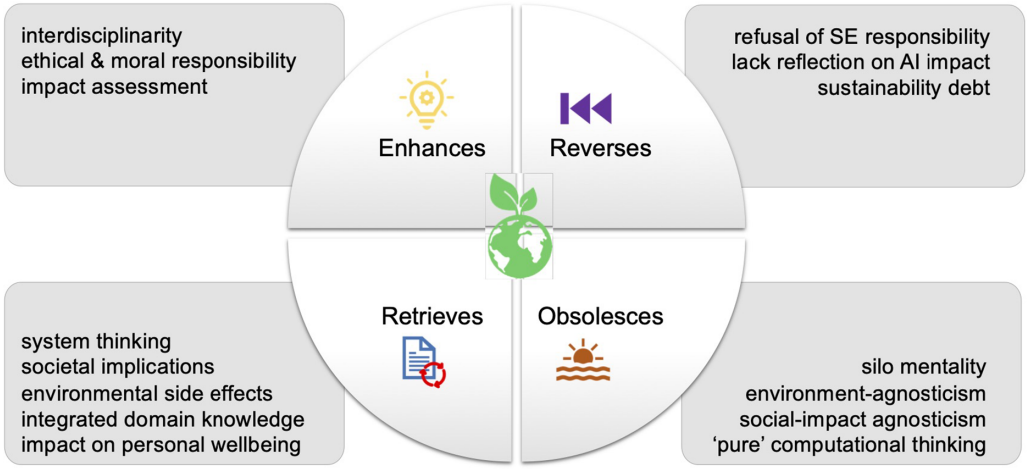

Fig. 5. The disruptive impact of sustainability on software engineering.

on technical sustainability and energy efficiency, and software *sustainability engineering* [148], that is, approaches to engineering software systems that create a positive environmental impact, for example, nature restoration or carbon capture. Sustainability *in* software engineering and engineer *for* sustainability refer to slightly different concepts of sustainability, which depend on the context and scope from which sustainability is considered. The anthropologist and systems thinker Joseph Tainter explains this need of scope [187], and clearly discusses that we must ask (i) which system to sustain, for(ii) whom, (iii) over which time frame, and (iv) at what cost, to properly address some sustainability concerns. The Oxford English Dictionary defines sustainability as "*the capacity to endure*." The Brundtland Commission defined *sustainable development* as "*meeting the needs of the present without compromising the ability of future generations to meet their needs*," which is the basis for the 2030 Sustainable Development Goals that many governments use as target references when implementing sustainability measures.

The McLuhan tetrad in Figure 5 illustrates the disruptive impact of sustainability on software engineering. Sustainable software engineering *enhances* multidisciplinary approaches by requiring integration of disciplines for the sake of social contribution, education on responsibility, and assessment of the impact of software systems on sustainability. Sustainable software engineering *retrieves* system thinking, environmental side effects, integrated domain knowledge, and impacts on individual well-being and society. Sustainable software engineering has the potential to *reverse* the refusal of software engineering responsibility for the impacts of software systems, sustainability debt, and a lack of sufficient reflected large-scale artificial intelligence development. Sustainable software engineering *obsolesces* pure computational thinking, silo mentalities, techniques and approaches that ignore environmental and social impact.

4.1 State of the Art and Trends

We discuss both sustainability *in* software engineering and software engineering *for* sustainability. König et al. [97] further differentiate and discuss the sustainability lifecycle assessment [157] in their paper in this special issue. Venter et al.'s definition of software sustainability as a "*composite, first-class, nonfunctional requirement, that is, a measure of a range of core software quality attributes, which includes at a minimum maintainability, extensibility, and usability*" [206], offers a purely

technical perspective that does not take into account the important dimensions of sustainability that Becker et al. discuss in their IEEE software paper [14]:

— *Environmental Dimension:* The use and stewardship of natural resources, ranging from immediate waste production and energy consumption to the balance of local ecosystems and concerns about climate change.
— *Individual Dimension:* The freedom of individuals and agencies, human dignity and fulfillment, the ability of individuals to thrive, exercise their rights, and develop freely.
— *Social dimension:* The relationships between individuals and groups, the structures of mutual trust and communication in a social system, and the balance between conflicting interests.
— *Economic Dimension:* The financial aspects and business value, capital growth and liquidity, investment questions, and financial operations.
— *Technical Dimension:* The ability to maintain and evolve artificial systems (such as software) over time.

4.1.1 Sustainability in Software Engineering. Sustainability *in* software engineering focuses on energy-efficient coding, energy-efficient algorithms, lean development environments, and what is often called *green* software engineering. Sustainability *in* software engineering aims to minimize both the environmental footprint of development and the subsequent energy consumption of software in production, independently of the purpose of the software itself. Sustainable approaches in software engineering focus on the different phases of development, ignoring the usage and impacts of the software produced.

There has been a significant amount of work in the area of requirements engineering on this topic. Venters et al. [201] define sustainability requirements as *a composite non-functional requirement*, and call for both a *definition that is tailored to quantitative sustainability objectives that encompasses its complexity and multidimensional nature* [205] and a consolidation throughout the community [202] based on the Karlskrona manifesto [15]. Penzenstadler [149] integrates sustainability within an artifact-based requirements engineering method, and Duboc et al. [54] integrate an analytic perspective on sustainability within requirements engineering.

Research in the area of architecture and design frames sustainability as *preserving the function of a system over an extended period of time* [99], based on an etymological understanding of the word applied to any system, regardless of the presence of software components, by merging the individual dimension into the social dimension. Venters et al. [203] emphasize the need to capture the rationale of significant sustainable design decisions to define sustainable architectures. Oyedeji et al. [141] contribute a software sustainability design catalog based on an implementation of the principles put forth in the Karlskrona Manifesto [15]. Naumann et al.'s GREENSOFT framework [136] and Calero and Piattini's *Green in Software Engineering* book [27] further elaborate on a sustainable architectural design.

4.1.2 Software Engineering for Sustainability. Software engineering *for* sustainability focuses on the purpose of the software system or service and emphasizes the responsibility of software engineers about the impact of software systems, as they all operate in the world [14]. Seyff et al. [180] and Penzenstadler et al. [153] propose five main sustainability dimensions for requirements engineering to reflect on the sustainability of the system, among the many approaches presented in Section 4.1. Seyff et al. [180] propose a model to negotiate stakeholder demands to achieve a positive impact on sustainability, and Penzenstadler et al. [153] define four approaches to identify stakeholders relevant to sustainability.

The relevant frameworks for including sustainability in the design of software systems are (i) the sustainability awareness framework [18], mostly applied in requirements engineering, in

initial development [150], in different iterations [151], and in reviews [159], (ii) the sustainability assessment framework [98], which offers a set of tools to support software architects and design decision makers in modeling sustainability as a software quality property, and (iii) ENSURE [174] for eliciting and decomposing sustainability requirements. Kienzle et al. [93] contribute a vision of a model-based framework that enables a broader engagement with and informed decision-making about sustainability issues, and identify core emerging challenges that include dealing with open-world contributions, uncertainty, and conflicting worldviews, and that still hold 5 years after their publication.

4.1.3 Challenges and Trends. The main open challenges of sustainable software engineering and software engineering for sustainability concern *multidisciplinary approaches*, *assessment*, *education*, and *large language models*.

4.1.4 Multidisciplinary Approach. The pervasiveness of software-intensive systems in all areas of our daily life leaves large shares of the population behind, and increases the digital divide [13]. Inclusive design methods can bridge some of the gaps between developers and stakeholders, still limited to developers and potential users. Multidisciplinary research combines the knowledge and ideas of experts and non-academic key stakeholders to address socially relevant problems, by both fostering cooperation and collaboration among scientific disciplines and non-academic stakeholders, and creating solution-oriented knowledge well suited for both scientific and societal practices [103]. Multidisciplinary research for sustainability comes with a set of challenges and requirements to establish mutual learning among researchers in different disciplines [100].

Multidisciplinary approaches are essential to tackle sustainability challenges that are inherently socially relevant. We need new methods that integrate knowledge from environmental science, economics, social sciences, and domain-specific areas with software engineering to develop sustainable software systems. We need new collaborative frameworks to facilitate cross-disciplinary innovation while ensuring effective knowledge transfer to software engineering practices. We need to understand how to embed sustainability-focused insights into software design and decision-making to build software systems that not only minimize environmental impact but also promote social responsibility and economic viability. König et al. [97] identify opportunities to address global sustainability challenges, such as climate change and social inequality, by enabling energy savings and social innovations. Currently, software systems threaten to exacerbate these crises, as evident in the increasing consumption of resources and widening digital inequalities. Software engineering plays a key role in tackling the problems and exploring the potential of software technology for sustainability. König et al.'s research vision of sustainability-driven software engineering and multidisciplinary research formats revisit the understanding of sustainability in software engineering, and then differentiate into the challenges of (i) sustainability lifecycle assessments, (ii) sustainability criteria with multi-criteria decision analysis for tradeoff decisions, (iii) sustainability assessment approach and implementation, and (iv) multidisciplinary research and real-world laboratories.

4.1.5 Assessment. Assessment plays a key role in measuring sustainability beyond just energy consumption. We need meaningful **Key Performance Indicators (KPIs)** that reflect a wide spectrum of sustainability, including carbon footprint, resource usage, social impact, and long-term viability [203]. Venters et al.'s [204] review of 234 studies offers a foundation and a road map of emerging research themes in the area of sustainable software engineering, with an emphasis on the assessment and identification of suitable KPIs. Guldner et al. [69] offer a reference measurement model that focuses on the energy and resource efficiency of software systems. Bets et al.'s [18], Lagos et al.'s [98], and Naumann et al.'s [136] studies identify several potential indicators that need a large-scale benchmark evaluation and the integration within the ISO standards, like the

environmental ISO 14,000 series and the social responsibility ISO 26000. Both Forrester's [58] and Penzenstadler et al.'s [152] papers stress the importance of systems thinking as a methodological approach for comprehensively assessing software systems, given their complex interplay with the world. We need standardized sustainability dashboards that integrate a set of suitable KPIs in an accessible and actionable manner, to enable organizations to effectively and comparatively assess the sustainability performance of the software systems and adapt to evolving environmental and ethical standards.

4.1.6 Education. Education is a foundational element in preparing future software engineers to address sustainability challenges. Called for already by Penzenstadler and Fleischmann in 2011 [154], teaching sustainability in software engineering remains a challenge. Universities are barely starting to integrate sustainability into their educational programs [155]. We needs to define (i) the new role of software engineers in a world increasingly shaped by environmental constraints and ethical considerations [17], (ii) educational curricula that embed sustainability principles into software engineering programs, to make students understand the environmental and social impact of their work [131], and (iii) new pedagogical approaches—such as experiential learning, interdisciplinary courses, and industry collaboration [155]—to help future engineers develop the skills and mindset needed to drive sustainable innovation.

Moreira et al. [131] highlight the two antithetical roles of software engineering in sustainability. On one side, software systems contribute to environmental issues through high energy consumption. On the other side, software systems hold the potential for solutions that improve efficient and equitable resource management. We need to (i) move beyond traditional curriculum models and fully integrate sustainability into every aspect of software development, (ii) embed sustainability as a core competency, and (iii) trigger a major shift in software engineering education. We shall start by raising awareness and establishing harmonized and clear sustainability concepts, integrate ethical thinking, and create a holistic view. On that foundation, we can establish sustainability metrics and indicators, integrate software engineering competencies for sustainable software, integrate skills for inter- and intra-personal teamwork, and build the business case for sustainability. We shall adopt evolving legal requirements and standards, change cultures through advocacy and lobbying, reorient artificial intelligence to drive sustainability, activate academic organizations, facilitate industrial adoption, and identify greenwashing. Betz and Penzenstadler's [17] describe the shifts in the role and responsibilities of software engineers, advocate for large language model-based approaches for coding, and highlight the deep shifts driven by the profound societal and environmental impacts of technology.

4.1.7 Artificial Intelligence and Rapid Global Digitization. Artificial intelligence systems carry both risks and opportunities for environmental sustainability [89]. We shall focus on both assessing artificial intelligence business cases through the lens of ethics and sustainability, and ensuring that artificial intelligence solutions contribute positively to environmental and social goals, rather than exacerbating issues such as resource consumption and bias [197]. Integrating artificial intelligence sustainably into software engineering requires investigating energy-efficient artificial intelligence models, responsible data usage, and scalable best practices that align with long-term sustainability objectives. By addressing these concerns, we can ensure that artificial intelligence serves as an enabler of sustainability rather than a source of additional challenges.

Kunkel et al. [95] contribute a scoping review of six software and artificial intelligence sustainability frameworks with respect to their recognition of environmental sustainability and the role of stakeholders, and provide recommendations for future research on how stakeholder involvement can help firms and institutions design and use more sustainable artificial intelligence systems. Shi

et al. [182] focus on reducing energy consumption of large language models for software engineering, and call for comprehensive benchmarks, efficient training methods, effective compression techniques, improved inference acceleration, and program optimization. Cruz et al. [45] focus on the impact of adopting environmentally friendly practices to create artificial intelligence-enabled software systems and discuss the environmental impact of using foundation models for software development, and claim the need to consider the business case, consolidate fundamental concepts, monitor sustainability, clarify roles' involvement, and change the machine learning lifecycle.

4.2 Roadmap

Sustainable software engineering requires a research approach that integrates diverse disciplines, establishes meaningful assessment metrics, enhances education, and ensures responsible adoption of artificial intelligence. This roadmap outlines key research areas: the role of interdisciplinary in fostering collaboration, and the need of assessment methodologies, education strategies, and ethical and sustainable integration of artificial intelligence.

— *How to integrate disciplines to work effectively together for sustainability?* We need interdisciplinary approaches that merge the knowledge of many disciplines, including environmental science, economics, social sciences, and domain-specific areas, into software engineering. We need collaborative frameworks to foster cross-disciplinary innovation and knowledge transfer.
— *What KPIs for sustainability beyond energy consumption?* We need to define KPIs that capture the many facets of sustainability. We need standardized sustainability dashboards that integrate a set of suitable KPIs.
— *How to educate future software engineers?* We need to define the role of software engineering for sustainability and educate software engineers on sustainability. We need curricula that fully integrate sustainability into every aspect of software development and embed sustainability as a core competency. We need new pedagogical approaches to help future engineers develop the skills and mindset needed to drive sustainable innovation.
— *How to assess the ethical and sustainable dimensions of artificial intelligence systems?* We need to sustainably integrate artificial intelligence into software engineering.

5 Automatic Programming

Automatic programming has gained prominence due to the capability of artificial intelligence-based coding assistants such as GitHub Copilot. The adoption of automatically generated code crucially depends on both the quality of the generated code and the trust that can be engendered for automatically generated code. The shipping of automatically generated code as part of commercial software products leaves important legal issues to be addressed.

It is fair to say that for the last 50 years, since the first "hello world" program of Brian Kerninghan in 1972, in the B programming language, a precursor to C, the focus of programming has been on programming in the large. With automatic programming, the integration of automatically generated software components becomes critical, and the attention turns to programming with trust [169]. With human programmers involved, there is an implicit expectation of *passing the blame* if the event of an error is found in the software. With automatically generated code, the core issue is trust, and the automatically generated code needs to carry with it a *high degree of trust*. In the last half century, the relevant work on software validation has focused on how to program larger and larger code bases. With code being automatically generated from natural language specifications, the focus shifts from programming at scale to the trustworthy *integration* of automatically generated code.

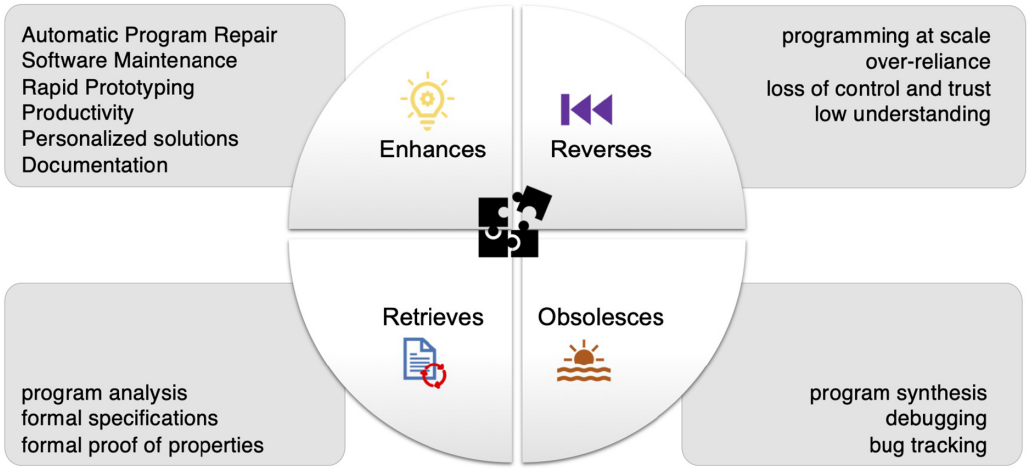

Fig. 6. The disruptive impact of automatic programming on software engineering.

What measures of trust future programmers will be comfortable is a core research question that remains to be explored. Due to the prevalence of agentic artificial intelligence approaches, automatic programming is practiced primarily via large language model agents for coding. Thus, trust issues will play an important role in the design of large language model agents for coding. In particular, when large language model agents generate code-related artifacts, they need to carry with them evidence of correctness, which can engender trust in the generated artifact.

The McLuhan tetrad in Figure 6 illustrates the disruptive impact of automatic programming on software engineering. Automatic programming *enhances* rapid prototyping and program repair, by largely reducing time and effort. It enhances productivity, documentation, and personalized solutions by substituting humans in many effort and time-demanding activities. Automatic programming *retrieves* program analysis, formal specifications, and formal proof of properties to check for the validity of automatically generated code. Automatic programming *obsolesces* classic program synthesis, debugging, and bug tracking. When pushed to the extremes, automatic programming *reverses* the emphasis of programming large code bases at scale, instead turning the attention to programming with trust. It reverses over-reliance on code, loss of control and trust, and reduces the understanding of code.

5.1 State of the Art and Trends

From a historical perspective, automatic programming traces back to Manna and Waldinger's seminal work on program synthesis [121] in the early seventies of the last century, and flourished in the following decades. The main approaches for synthesizing programs transform certain semantic and syntactic specifications into either expressions or program snippets that meet the semantic and syntactic specifications. Semantic specifications can sometimes be very intuitive, such as input–output examples and semi-formal specifications, while syntactic specifications are typically restrictions on the syntax of the program to be generated. The early work on automatic code generation from UML state diagrams still required a significant human effort for modeling.

Large language models raise significant attention towards using natural language prompts to generate code. Recently, the focus has shifted towards a more agentic artificial intelligence approach, where large language models can invoke tools, including program analysis tools, to autonomously complete programming tasks [242]. The main difference between prompts and agents

is the autonomy of agents. Agents can autonomously employ program analysis and file navigation tools to generate code, test cases, and patches. The autonomy of large language model frees the agent to plan and execute several steps to accomplish a programming-related task.

Many of the agentic artificial intelligence approaches for automatic programming available today are suitable for solving "software issues." An "issue" is a natural language report given as a unit of work to a software engineer. It could be a task such as a bug fix or a feature addition. The current state of the art in automatic programming aims to resolve such issues automatically and *autonomously*. A key issue for driving innovation in autonomous software engineering is the availability of challenging problems or data sets. The program repair community had previously proposed the Manybugs and IntroClass benchmarks for repairing C programs [64], while a large number of research publications have looked at the Defects4J benchmark [84] for repairing Java programs. These benchmarks, though influential, are hard to visualize as challenge datasets to spur interest in automatic programming. Usually, program repair benchmarks document the desired behavior via test cases, and a suitable test suite may not be available in all software projects. The natural language processing research community has recently proposed the SWE-bench dataset [], a set of GitHub Python software projects that challenge autonomous software engineering with GitHub issues, such as fixing bugs and adding features. There is still the opportunity to extend the SWE-bench beyond fixing natural language issues. In general, we need datasets that cover a wide variety of software engineering tasks.

Some very recent technological disruptions in the hardware space, such as the float of NVIDIA Inference Microservices as a scalable mechanism to harness generative artificial intelligence capabilities via applications, can dramatically impact future software engineering in the presence of generative artificial intelligence. Such technology trends can dramatically lower the entry barrier to using generative artificial intelligence in producing commercial software.

The frontier of large language models for automatic programming is a largely open research question. A software professional today does not only perform individual tasks, such as testing, coding, and patching. An expert software engineer handles many complex *scenarios* (by borrowing a term form the software industry), such as (i) adding a feature and then taking care of any bugs it can introduce in a codebase, (ii) adding a fix according to an issue, and if the fix is incomplete, completing it, and (iii) taking over the code of a developer who has left the organization, running test suites to understand the code, and adding more test cases to improve the understanding. Overall, scenarios are chains of primitive tasks. Whether large language model agents can generalize to complex scenarios is still an open research question.

The papers in this special issue highlight three interdependent research directions in automatic programming: (i) how to autonomously improve code automatically generated from artificial intelligent coding assistants, (ii) what techniques to safely and securely integrate automatically generated code into software projects, and (iii) what future programming workflows and the role of software developers and engineers in the future workflows. The papers take a forward-looking outlook by envisioning future-day software projects as combinations of manually written code, automatically generated code, and natural language prompts, which represent the specifications of code snippets.

The *editors' paper* [117] provides an overview of automatic programming and discusses the future research directions for the software engineering community. The paper stresses the role of large language model-based agents in achieving autonomous software engineering, for possibly autonomously improving automatically generated code. A space for innovation in automatic programming is the design of different large language model agents that can automate different software engineering tasks and can be combined with automatic programming. A case in point for work in the software engineering community along these lines is AutoCodeRover [242], which

gleans the programmer's intent from the structure of the buggy program through a layered code search. He et al.'s paper [73] in this special issue outlines the importance of large-language model agents in a general sense. Lyu et al.'s paper [117] in this special issue underlines the role of large language model agents for autonomous software engineering, specifically in improving automatically generated code: Large language model agents can serve to achieve the repair of automatically generated code. Chen et al.'s paper [35] in this special issue discusses large language models for developing and maintaining code for mobile apps.

5.2 Roadmap

- *How to automatically generated complex scenarios?* Where do we go from the current burst of interest in large language model agents? Will it persist or will it morph in some ways to define the software engineering of the future in a consolidated fashion? Is this as far as we can progress the capabilities of an artificial intelligence software engineer? We need transparent approaches to trustfully generate complete scenarios that software engineering can understand and review.
- *What automatic programming for generating useful and trustworthy software systems?* How to autonomously improve code automatically generated from artificial intelligent coding assistants? What techniques to safely and securely integrate automatically generated code into software projects? What measures of trust for automatically generated code? Automatic programming shifts focus from programming at scale to programming with trust. Artificial intelligence-powered systems will generate large-scale programs from artifacts in natural language as well as other human accessible means that describe humans' needs. We need generative agents that encompass evidence of correctness, which can engender trust in the generated artifact.
- *What hybrid human–artificial intelligence agent teams?* What future programming workflows, and what will be the role of human programmers in future workflows? The integration of artificial intelligence into software teams raises questions about team dynamics. How will software engineers interact with artificial intelligence agents? How should artificial intelligence agents mutually collaborate? Traditional cognitive theories of cooperative work, which focus on human goal influence, may not fully encapsulate the nuances of software-engineering collaboration [10]. We need new processes that address the dynamics of hybrid teams.
- *What role of software engineers in the era of automatic programming?* The role of software engineers and developers shifts from designing and programming code to controlling the front-end artificial intelligence-powered programmer, and certifying the back-end final automatically generated programs. The attention turns to programming with trust [169]. We need to refine the individual competencies of software engineers.
- *What iterative improvement and architectural vision?* Unlike environments where game-theoretic incentives induce behavior, software development is driven by long-term design considerations, iterative improvements, and architectural vision [169]. We need to shape a new horizon to incentivize software development.

6 Security and Software Engineering

As software engineering practice becomes more automated and autonomous, we need to cope with new types of software vulnerabilities, such as prompt injection attacks, with a relevant impact on software security. At the same time, artificial intelligence offers new opportunities to detect and remove vulnerabilities.

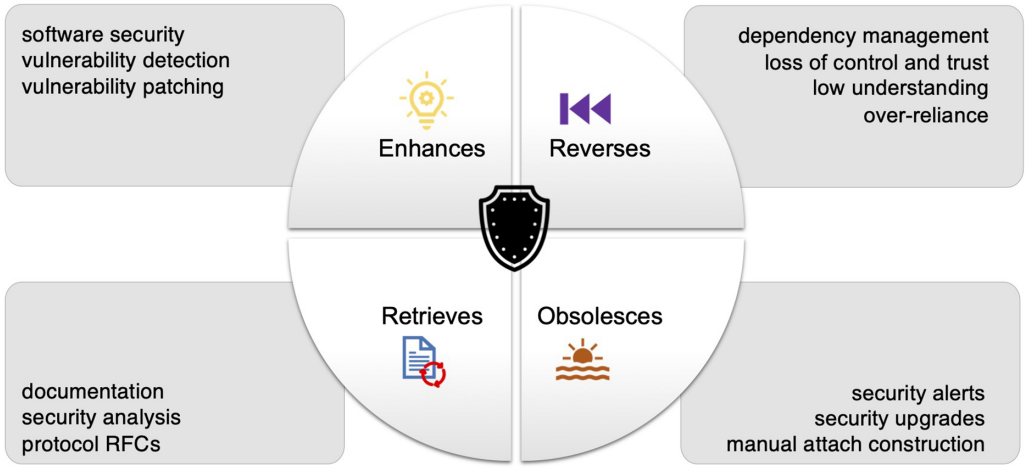

Fig. 7. The disruptive impact of artificial intelligence on software security.

The McLuhan tetrad of Figure 7 illustrates the disruptive impact of artificial intelligence on cybersecurity. Artificial intelligence in software security *enhances* software security testing, vulnerability detection, and patching. Artificial intelligence in software security *retrieves* documentation, protocol Request for Comments, and software security analysis. Artificial intelligence in software security *obsolesces* manual attack vector construction, security alerts, and security upgrades. When pushed to the extreme, artificial intelligence in software security *reverses* manual approaches for dependency management, lack of control and trust, and over-reliance on software security.

6.1 State of the Art and Trends

Artificial intelligence in software engineering raises complementary issues and opportunities. On one side, new types of errors and vulnerabilities arise from code generated with artificial intelligent agents [57, 221]. On the other side, artificial intelligence offers new ways to timely detect and repair vulnerabilities [57, 147].

Artificial intelligence impacts every phase of the critical digital infrastructure, including static code analysis, vulnerability detection, fault localization, fault patching, and patch ranking. Static code analysis commonly relies on either pattern-based approaches or flow analysis. Large language models can largely improve static code analysis; however, large language models are primarily based on natural language processing, and code cannot be simply treated as text. Fuzzing and symbolic execution approaches, such as white-box fuzzing, are often used for detecting vulnerabilities on a day-by-day basis. Large language models can deeply improve fuzzing, with their capability to ingest and learn system documentation and protocol specifications that can guide fuzzing [126]. Fix localization and patching approaches combine metaheuristic search and (symbolic) analysis, to capture the intent of the developers and guide vulnerability patching [65]. Large language model agents for program repair will use (symbolic) analysis tools in the back-end as part of agentic artificial intelligence approaches. Currently, patch ranking relies on simple metrics, like code edit distance. Suitably tuned large language models will soon substitute simple metrics for ranking patches. Table 1 summarizes the deep changes that we foresee in the different phases of the pipeline.

Artificial intelligence will impact on all the steps of the software security technology pipeline. The evolution of the 2016 DARPA Cyber Challenge (CGC), mainly focused on vulnerability detection, into the 2023-2025 DARPA artificial intelligence Cyber Challenge that widens the focus on both

Table 1. The Evolution of the Software Security Technology Pipeline to Protect Critical Digital Infrastructures

Phase	Current	Future
Static Code Analysis	Pattern-based, flow analysis	Large language model guided text-based analysis
Vulnerability Detection (Testing)	Fuzzing	Large language model guided random and fuzzing search
Fix Localization, Patching	Search and analysis	Large language model agent with analysis support
Patch Ranking	Edit distance or other metrics	Fine-tuned large language models

vulnerability detection *and* fixing, witnesses the interest in holistic approaches for detecting and fixing security issues. We envision a growing interest in zero-day patching, where detections will pair vulnerabilities with patch suggestions. We also envision an increasing development of approaches that combine vulnerability detection with agentic artificial intelligence approaches to produce production-grade tooling for end-to-end vulnerability detection and remediation. The research in security engineering will be closely related to industry innovation and trends, with approaches that deal with software systems in multiple programming languages, thanks to the involvement of large language models that do not suffer from the limitations of conventional program analysis tools, which typically focus on a single programming language.

The papers in the special issue clearly analyze both aspects. The *editors' paper* [117] in this special issue provides a comprehensive review of the literature on the use of large language models for detecting and preparing vulnerabilities. The paper discusses in detail static application security testing, a core technology for software security analysis, which performs white-box analysis through source code scanning, and detects vulnerabilities much more efficiently than manual review of source code. The paper summarizes the capabilities and discusses the open research directions of static application security testing.

Any outlook on software security is informed by the attack vectors faced by software systems. A class of attacks that has recently gained prominence is the supply chain attacks, such as those exposed by the Solarwinds attack. Software systems are vulnerable to supply chain attacks, since critical software systems may rely on other open source software, which can be exploited to launch attacks or exfiltrate critical data. Williams et al.'s paper [219] in this special issue articulates the issues of supply chain security and provides perspectives and knowledge obtained from intentional outreach with practitioners to understand the practical challenges and the extensive research efforts. The article provides an overview of current research efforts to secure the software supply chain and proposes a future research agenda to close software supply chain attack vectors and support the software industry. Wang et al.'s paper [211] in this special issue studies the supply chain of large language models through the dual lenses of software engineering and security and privacy, and analyzes each layer of the supply chain, presenting a vision for robust and secure large language model development. Zhou et al.'s paper [246] in this special provides a literature review of approaches to improve vulnerability detection and repair through large language models.

6.2 Roadmap

— *What artificial intelligence-powered approaches to protect critical digital infrastructures?* Large language models pave the way towards multi-language approaches for detecting and fixing vulnerability faults We envision artificial intelligence-powered tools to protect software ecosystems in production under human control.
How to prevent security breaches in automatically generated code? Automatic programming opens new frontiers for security engineering. We can prevent security issues in the generated code by suitably *combining large language models and neural networks with classic vulnerability*

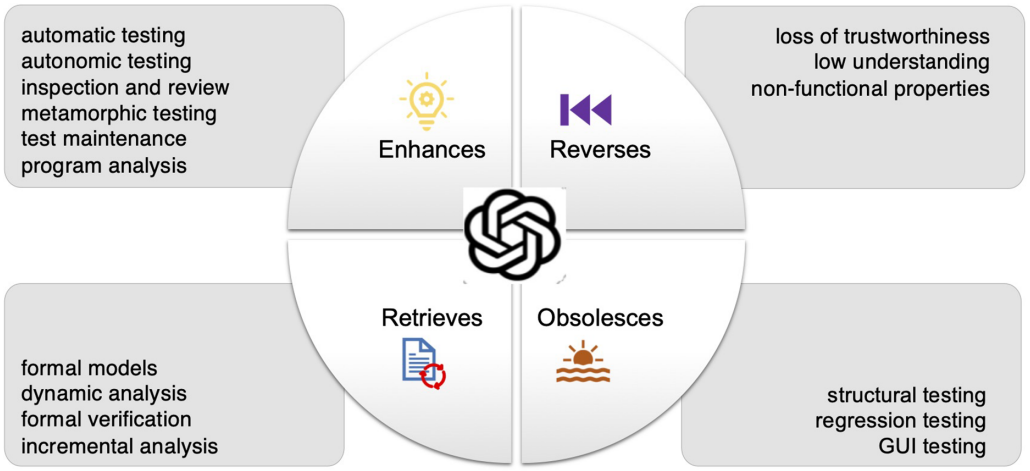

Fig. 8. The disruptive impact of artificial intelligence on software verification and validation.

analysis, to *trustworthy detect and fix bugs automatically*, and *analyze and prevent vulnerability issues in automatically generated code.*

7 Verification and Validation

Software verification and validation is well-supported by mature approaches that both sample (test) and fold (analyze) the source code and execution space of target software modules to detect failures and locate bugs [156]. Software engineers rely on mature and well-integrated tools within integrated development environments to automate many repetitive and tedious activities [55, 218]. Software testing approaches and tools execute the target software on a testbed and assume that software executions are repeatable. Program analysis approaches and tools are based on structured models of the source code. All verification and validation approaches assume that humans interact with software systems as external *users*, that is, humans *use* the software system through the provided interfaces, which drive test and analysis.

The emergence of generative artificial intelligence disrupts the overall verification and validation paradigm. It significantly enhances automation in verification and validation activities while redefining the roles of software engineers. The last generation of software engineers evaluates *intelligent human-centric ecosystems* built with *artificial intelligence-powered tools*, where *humans are embedded within the systems themselves*, rather than serving only as end users.

The McLuhan's tetrad in Figure 8 illustrates the disruptive impact of artificial intelligence on software verification and validation. Artificial intelligence *enhances* test automation by strengthening the many activities that are already well-supported by existing tools. It amplifies field testing, which is extremely useful to timely verify the evolving and emerging behavior of artificial intelligence-powered applications, enabling *autonomic testing* that ultimately flattens the border between testing and production. It radically changes code inspection, review, and pair programming, with humans controlling and cooperating with virtual inspectors and artificial intelligence-powered reviewers. It opens new frontiers to metamorphic testing, maintenance, and evolution of test suites. It enhances program analysis to verify both traditional and emerging properties of artificial intelligence-powered tools and artificial intelligence-generated software.

Artificial intelligence *retrieves* analysis in the many different forms, formal models, dynamic analysis, formal verification, and incremental analysis, to handle a new class of failures and bugs associated with the new characteristics of artificial intelligence-generated code. Artificial intelligence *obsolesces* classic approaches that rely heavily on predefined code structures and artifacts, such as structural testing, regression testing, and GUI testing, which become increasingly inadequate in artificial intelligence-based software environments. Artificial intelligence *reverses* trust and non-functional properties. It pushes to the limits the lack of trust, trustworthiness, and understanding of both artificial intelligence-driven activities and artificial intelligence-generated solutions. It pushes non-functional properties such as fairness, transparency, and trustworthiness to the extreme.

7.1 State of the Art

Recent progress in natural language, voice, image, and video processing, generative artificial intelligence, extended reality, and quantum computing has dramatically changed the landscape of software verification and validation. Natural language, voice, image, and video processing, and artificial intelligence, open new opportunities to enhance classic approaches and tools to automate all aspects of program verification, towards a completely automated process. Both artificial intelligence and quantum programs branch off from the core assumptions of classic verification and validation approaches, namely, repeatable executions and transparent models of the code. Artificial intelligence supports the fully automatic production of code, with types of failures and bugs that depend on limitations and hurdles of automatically generating code, rather than human factors. Artificial intelligence and extended reality pave the road to a new generation of artificial intelligence-powered human-centric ecosystems with both self-adaptive and evolving behavior, and with humans in and not just users of the system. Software engineers no longer only test software systems within agile teams. They test *self-adaptive human-centric ecosystems* with *artificial intelligence-enabled approaches and tools* within *hybrid human-virtual teams*.

We discuss the *impact of new technologies, and in particular generative artificial intelligence, on software engineering teams and processes* in Section 3 of this editorial, and review the *impact of quantum computing on software engineering*, and discuss the challenges of testing quantum software systems in Section 8 of this editorial. In this section, we summarize the state of the art and spotlight the current trends about the impact of new technologies, and in particular deep learning and generative artificial intelligence, on *automatically generating test cases*, the issues of *verifying and validating artificial intelligence-generated code*, *testing artificial intelligence-powered systems*, and *verifying smart human-centric ecosystems*.

7.1.1 Automatically Generating Test Cases.

State of the Art and Trends. Although most software testing activities can rely on mature tools, the generation of test cases does not go much beyond the automatic refinement of initial test suites, such as capture-and-replay and regression testing. The research results of the first two decades of the century on automatically generating unit test cases, such as Evosuite [60] and the recent studies of generating test cases and oracles with natural language processing, like CaMeMa [22], did not break through industrial practice, yet.

Challenges and Trends. The attempts of the last years to automatically generate test cases and oracles with large language models [207] indicate a clear trend toward the use of large language models and deep neural networks to generate test cases. We envision largely automated verification and validation environments where testers govern artificial intelligence-powered tools that automatically verify the software systems, with humans as a fraction of the largely virtualized population that validates the seamlessly evolving systems in production. The automatic generation

of test cases and oracles with large language models will massively substitute human-intensive activities, as soon as researchers address the many still open challenges about the datasets for training the models, the fine-tuning of the general models, the scalability to large systems, and the completeness of the generated suites.

The automatic generation of tests and oracles greatly reduces human effort, with huge impacts on maintenance and evolution of test suites that seamlessly adapt and evolve to verify the emerging and evolving behavior of artificial intelligence-powered software systems, and it will ultimately replace current test regression and maintenance activities. Generative artificial intelligence enhances validation activities by multiplying the effort-intensive contribution of humans with avatars and bots to exhaustively validate the behavior of software systems with a limited human effort.

Li et al.'s paper [106] in this special issue discusses the suitability of large language models to automatically generate metamorphic tests. Molina et al.'s paper [129] focuses on the challenges and opportunities of automatically generating test oracles. Cederbladh et al.'s paper [32] advocates the need of new models to cope with both organizational and behavioral complexity to early verify and validate software systems. Abrahão et al.'s paper [1] discusses the challenges and the impact of hybrid human–virtual teams on trust, communication, and inter-personal relationships.

7.1.2 *Verifying and Validating Artificial Intelligence-Generated Code.*

State of the Art and Trends. The current approaches to software testing aim to reveal failures and bugs that are due to human errors. Human errors are barely shared with artificial intelligence engines. Developers fail due to human factors, like misunderstandings, distractions, lack of familiarity with languages, tools, and environments, while artificial intelligence engines are trained with all required languages, tools, and environments, never distract, and do not suffer from typical human misunderstandings. Artificial intelligence-generated code suffers from new types of errors that are largely different from human errors, like hallucinations, misleading choices, and *ad hoc* solutions, and that escape most classic testing and analysis approaches.

Challenges and Trends. We argue that both static and dynamic program analysis techniques that abstract the details of the single executions by folding the infinite execution space into finite partitions [156] can reveal the new types of faults in artificial intelligence-generated code much better than common testing approaches, if properly blended with testing, by generalizing the results of executing finite test suites that sample the infinitely large execution space. Both the training dataset and the prompts can largely bias the process and consequently the generated code. We need new approaches to verify the new properties of both the generation process and the generated code. We need approaches to verify both the stability and reliability of the dataset that we use to train the artificial intelligence engines and the stability and trustworthiness of the results that we obtain by prompting the artificial intelligence engines. Verification of domain-conformance and biases of the training datasets is still a big open challenge. The validation of transparency, explainability, and the absence of both biases and ethical issues is an open issue. Ahmed et al.'s paper [33] in this special issue identifies the many challenges of generative artificial intelligence and machine learning for software engineering, and discusses the challenges of both creating suitable data for training, prompting, and validating artificial intelligence engines, and the role of explainability of the results.

7.1.3 *Testing Artificial Intelligence Powered Systems.*

State of the Art and Trends. The current approaches to test and analyze software systems assume the repeatability of the behavior. Developers and testers test software systems on testbeds, by assuming the same behavior of the system in production. Regression testing executes the test suites

on new versions of the application to check for the conformance of the behavior with respect to the former versions. The maintenance of software systems aims to both enrich the functionality of the application and remove the faults that show up only in production. The approaches to field testing monitor the applications in production to enrich the test suites with new test cases [16].

Challenges and Trends. Artificial intelligence-powered systems evolve over time and adapt to conditions that emerge only in production, by taking advantage of data from production to train the model, with behaviors that cannot be fully predicted in advance and replicated on testbeds before production. The studies of self-adaptive systems highlight the challenges of dealing with adaptive behaviors within control loops [50].

Verification and validation of the behavior of artificial intelligence-powered systems is still an open issue, despite the many recent results [241]. The adaptive and non-deterministic behavior of artificial intelligence-powered systems adds a new combinatorial dimension that exacerbates the already huge issue of the combinatorial explosion of test suites and opens unexplored challenges to the already hard oracle problem. Cederbladh et al.'s paper [32] in this issue emphasizes the importance of early validation and discusses the impact of generative artificial intelligence on model-based early verification and validation.

7.1.4 Verifying Smart Human-Centric Ecosystems.

State of the Art and Trends. The common approaches to software testing and program analysis focus on the *Software Under Test*, and the many recent approaches to testing autonomous systems, notably autonomous cars, are not an exception. They focus on the *ego vehicle* in the context of *Non-playable Characters*, that is, the behavior of an autonomous vehicle (the ego vehicle) that moves in the context of dynamic obstacles (NPCs) that behave independently from the behavior of the ego vehicle [87, 118]. The testing of ego vehicles can assess the quality of single vehicles; however, it can miss accidents due to unexpected interactions of multiple autonomous vehicles [61].

Challenges and Trends. Artificial intelligence-powered systems and, more generally, software systems commonly operate in open (cyber-physical) environments and interact with both the environment and humans. *Smart human-centric ecosystems*, such as smart cities, smart buildings, smart grids, healthcare systems, e-markets, emerge as communities of independently owned and operated (cyber-physical) systems with smart functionalities, each with its own requirements. Smart human-centric ecosystems adapt and evolve over time while systems enter and exit the ecosystem, with humans as primary actors within the ecosystem and not just users of the systems [42]. Smart human-centric ecosystems can fail due to implicit conditions and interactions that emerge only in production, even when all systems in the ecosystem behave correctly as specified [132].

The characteristics of smart human-centric ecosystems challenge verification and validation activities. The absence of complete and stable specifications of the ecosystems challenges the classic definition of correctness as conformance with some specifications. The test cases of smart human-centric ecosystems shall take into account the adaptive and evolving behavior of the ecosystem and shall encompass scenarios and requirements from production. The citizens and visitors seamlessly interact within the smart city beyond and without the boundaries of GUIs and interfaces. The testing and analysis activities shall take into account the new role of humans as primary actors in the ecosystems. Large ecosystems like smart cities cannot be fully tested in testbeds or production. Testbeds cannot reproduce all scenarios that emerge only in production, while Tests that involve people cannot be executed in production. We envision an evolution of lightweight digital siblings [66] that encompass models of humans and human groups and that mirror the smart human-centric ecosystem to fully test the ecosystem.

Casadei et al.'s paper [29] analyzes the challenges of engineering *collective cyber-physical systems* composed of uniform, collaborative, and self-organizing groups of entities. Birchler et al.'s paper [21] discusses the challenges of simulation-based testing of autonomous cyber-physical systems.

7.2 Roadmap

- *How to tune generative artificial intelligence to generate valid test suites?* We need to tune generative artificial intelligence to take into account the aspects of code, to avoid too many false positives and negatives that pollute test suites.
- *How to assess artificial intelligence-generated code?* We need metrics that address the new challenges of artificial intelligence (like data leakage, dataset bias, and reproducibility) to assess both the artificial intelligence-generation process and the artificial intelligence-generated code.
- *How to blend software testing and program analysis to verify functional and non-functional properties of artificial intelligence-generated code?* We need to blend testing and analysis to cope with classic as well as emerging properties of artificial intelligence-generated code.
- *How to define the quality of cyber-physical ecosystems?* We need to define the properties of intelligent human-centric ecosystems that capture the quality of ecosystems without complete and stable specifications.
- *How to generate test cases to test evolving systems?* We need new approaches both to identify scenarios that emerge only in production and that have not yet been tested, and to generate test cases for such emerging scenarios.
- *How to generate test oracles for evolving ecosystems?* We need test oracles that capture the desired behavior of systems and ecosystems that adapt and evolve over time.
- *How to model humans and human groups for testing smart cyber-physical systems?* We need accurate models of humans and human groups to thoroughly test smart cyber-physical systems without interfering with the ecosystem in production.
- *What digital twins to thoroughly test smart cyber-physical systems?* We need digital twins that capture all and only the aspects that are relevant for testing smart cyber-physical systems, to guarantee through testing without the costs of heavyweight twins.

8 Quantum Software Engineering

Quantum computing and quantum software engineering have stepped strongly into the science and research spotlight in the last decades, with the potential to dramatically impact many relevant sectors such as artificial intelligence, drug engineering, and climate modeling. Today's quantum computers are mainly specialized and **Noisy Intermediate-Scale Quantum Devices (NISQ)**.[6] However, quantum computers are rapidly evolving and improving. As Brook observes in his spotlight paper [23], quantum computing will largely substitute rather than simply complement classic computing, as soon as general, reliable, and fully scalable quantum computers become available.

Quantum software engineering is an interdisciplinary field that focuses on the principles, methodologies, standards, and tools for designing, developing, testing, maintaining, and managing quantum software systems, including hybrid quantum-classic systems that run on quantum computers or quantum simulators. Quantum software engineering involves the application of quantum-based techniques, including quantum algorithms such as quantum annealing, to solve complex problems in both classic and quantum software engineering. Quantum software engineering challenges

[6]NISQ devices have a small number of qubits and are subject to significant levels of noise and errors due to environmental interference and hardware imperfections.

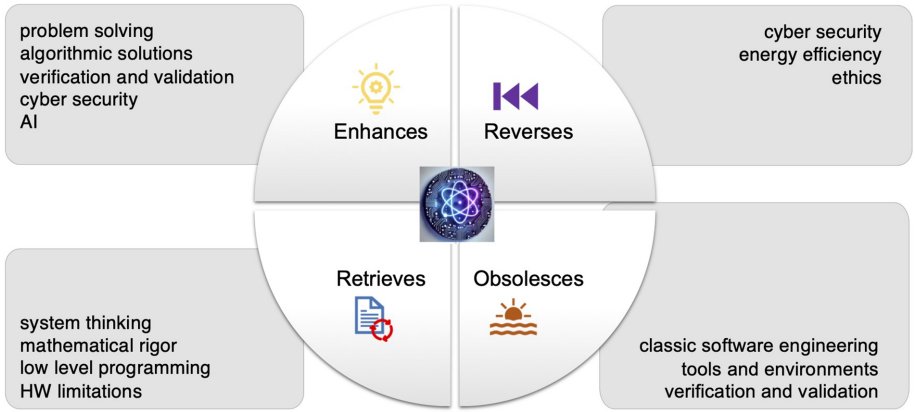

Fig. 9. The disruptive impact of quantum software engineering.

researchers and practitioners with a unique set of unexplored issues, capabilities, and opportunities that are driven by the principles of quantum mechanics, such as entanglement and superposition. Quantum computing dramatically reshapes the software engineering skyline: It requires new approaches to address the unique challenges of quantum software and leverage the specific strengths of classic software engineering solutions [5].

The tetrad in Figure 9 visualizes the disruptive impact of quantum software engineering. Quantum software engineering *enhances* problem solving, algorithmic solutions, artificial intelligence, verification and validation, with its potential to demolish the NP barrier, and pave the way to radically new algorithms and solutions. It enhances software security with the new frontiers of quantum cryptography.

Quantum software engineering *retrieves* system thinking and mathematical rigor in software engineering to handle the complexity that quantum software inherits from the principles of quantum mechanics. It retrieves both low-level programming at the quantum gate level, and the need to consider hardware constraints and limitations due to the number of qubits, the gate fidelity, the error rates, and the accessibility of qubits.

Quantum software engineering *obsolesces* the classic common practice of software engineering that relies on assumptions about the deterministic behavior of Turing machines and cannot handle quantum-specific features such as superposition and entanglement. For instance, classic software engineering manages concurrency and parallelism via threads, processes, and distributed computing, while quantum computing realizes parallelism by leveraging quantum entanglement and superposition. Similarly, the probabilistic nature of quantum computing drives out of practice classic debugging and testing techniques. It *obsolesces* new verification and validation approaches for artificial intelligence systems and artificial intelligence-based applications, by pushing the limits of complexity.

When pushed to the extreme, quantum software engineering *reverses* software security by invalidating the core assumptions of many current encryption algorithms. It also reverses energy efficiency with the huge energy demand of quantum computers. It pushes ethical concerns and technical wisdom to the limit by moving beyond classic computability borders. Overall, the inherent complexity of quantum computing and hence quantum software engineering pushes innovations to the limit, and the limited access of the quantum software engineering community to quantum hardware may limit the innovations of quantum software engineering.

8.1 State of the Art and Trends

Over the past years, the quantum software engineering community has introduced many methodologies and tools to address specific challenges in the engineering of quantum algorithms and applications. Although the research community has explored different phases similar to classic software development, from requirements engineering to architecture and design, modeling, programming, testing, and debugging, there is no well-defined and widely recognized quantum software development lifecycle or practice yet. In this section, we discuss the state-of-the-art and current trends in the different phases of the software lifecycle, and we discuss the main trends of quantum software engineering for artificial intelligence and artificial intelligence for quantum software engineering.

8.1.1 Requirements Engineering.

State of the Art. Requirements engineering is critical for quantum software engineering, and the limited attention to quantum requirements engineering compared to the work on quantum testing, debugging, and architecture is mainly due to the lack of commercially relevant applications. The current studies on quantum requirements engineering highlight the importance of considering hardware-related constraints such as the number of available qubits, the depth of quantum circuits, and the amount of noise in NISQ, with some preliminary studies on applying classic requirements engineering practices such as use cases and goal modeling for quantum.

Challenges and Trends. Quantum requirements engineering faces unique challenges, such as distinguishing and separating requirements to be handled with classic computing from those to be addressed with quantum computing, and systematically analyzing constraints related to quantum hardware. Murillo et al.'s paper [133] in this special issue briefly mentions that quantum requirements engineering will inevitably deviate from classic requirement engineering approaches.

In summary, the quantum community calls for quantum requirements engineering solutions that (i) deal with all the different requirements engineering aspects including requirements elicitation, requirements specification and modeling, requirements analysis, (ii) explicitly address the inevitable evolution of quantum hardware in requirements, and (iii) enable complex requirements analysis about the optimal allocation of classic and quantum computing resources to address different requirements.

8.1.2 Architecture.

State of the Art. Quantum software architecture aims to design quantum software systems that efficiently execute quantum algorithms, manage quantum computing resources, and integrate with classic software systems. The current studies on quantum software architecture focus on quantum service-oriented computing, which adapts service-oriented computing principles of classic computing to quantum software architecture, to design hybrid quantum applications, and benefits from well-studied techniques such as DevOps to facilitate the continuous deployment of quantum software.

Challenges and Trends. Murillo et al.'s paper [133] in this special issue highlights important research directions in quantum software architectures, such as identifying all factors that influence architectural decision-making and designing design patterns to facilitate the development of hybrid quantum software systems. Murillo et al discuss the main challenges to the research community for exploiting the full potential of quantum service-oriented computing: standardized and platform-independent APIs to interact with quantum computers, demand and capacity management, and workforce training. Yue et al.'s paper [239] in this special issue discusses the need for a taxonomy

of constraints specific to quantum software architectural designs, a taxonomy of functional and extra-functional quality attributes, the integration of quantum software components into existing architectures, and the consideration of quantum hardware-specific properties in architectural designs.

We need taxonomies, design patterns, architectural design methodologies, and tools that extend and revisit classic software engineering architecture description languages to address the new characteristics of (hybrid) quantum applications.

8.1.3 Modeling.

State of the Art. The quantum software engineering community defined several models for quantum programs, with quantum circuit diagrams being the most popular one. Quantum circuit diagrams model the sequence of logical quantum gates applied to qubits, and abstract from both the mathematical representation and the implementation details of the gates. There are several approaches to analyze and optimize quantum circuits: optimize the depth of gates, reduce the number of gates, and minimize errors and noise by selectively using certain types of quantum gates, given that the specific quantum hardware is known during the design phase. Several studies suggest model-driven engineering as a viable solution to raise the level of abstraction, reduce the complexity, and enable automation. Researchers proposed high-level modeling notations based on Unified Modeling Language, Business Process Model and Notation, quantum flow charts, and quantum decision diagrams.

Challenges and Trends. We need to raise the abstraction level of models, which is currently very low. Murillo et al.'s paper [133] in this special issue discusses the challenges of model-driven engineering for quantum computing about modeling quantum-specific constructs at a high level, and enabling intelligent code generation.

We need abstract models to (i) deal with the design and development of *complex* quantum software systems, (ii) enable standard communications and integration across classic and quantum computing paradigms, to facilitate the design and development of hybrid software systems, and (iii) facilitate automatic transformations to downstream artifacts such as quantum circuits, code, and tests.

8.1.4 Programming.

State of the Art. Current quantum programming languages (such as QSAM, Qiskit, Q#) require understanding quantum-specific concepts, such as superposition and entanglement, hardware limitations such as noise and the number of available qubits. We need quantum programming languages that abstract from quantum-specific concepts to reduce the complexity of programming and scale to large programs, and mature tools and libraries that go beyond the simple libraries currently available for quantum programming. So far, the main effort toward abstract programming concepts has focused on treating quantum registers as data-type encoding, defining new types to encapsulate qubit states and operations, and abstracting quantum states and operations.

Challenges and Trends. Murillo et al.'s paper [133] in this special issue discusses the challenges of defining quantum programming languages that effectively support the complexity of quantum circuits, enable circuit reuse and optimization, and raise the abstraction level to ease the cognitive load of quantum programmers. We need effective and mature approaches for optimizing quantum circuits, ensuring reusability, and facilitating integration with classic software systems. We need comprehensive quantum software libraries and frameworks to enable cross-platform development, determine the suitable level of abstraction, and efficiently scale to large quantum programs.

8.1.5 Testing and Debugging.

State of the Art. Classic testing strategies, such as assertions, bug patterns, and debugging techniques, barely adapt to the characteristics of quantum programming. Recent studies investigate how to adapt classic testing strategies, including coverage criteria, partition testing, combinatorial testing, search-based testing, fuzz testing, property-based testing, mutation testing, and metamorphic testing to quantum program testing, with encouraging but still preliminary results.

Challenges and Trends. The studies of approaches that extend classic testing and debugging to quantum programs only scratch the surface of a largely open area with many new challenges. Ramalho et al.'s [137] and Murillo et al.'s [133] papers in this special issue present the main challenges of testing quantum software systems. Ramalho et at. [137] discuss the challenges of adapting classic testing strategies, partition testing, structural testing, combinatorial testing, search-based testing, fuzz testing, property-based testing, mutation testing and metamorphic testing to quantum program testing, and present a conceptual model that summarizes the key concepts and challenges of quantum software testing and debugging. Murillo et al. [133] overview the challenges of test oracles, test scalability, test optimization, and transitioning from simulators to quantum computers, considering the noise of current quantum computers.

We need quantum-specific testing strategies that effectively (i) deal with the test oracle problem exacerbated by the no-cloning theorem in quantum computing, (ii) handle noise when executing test cases on quantum computers that are inherently noise and error prone, (iii) generate test cases from high levels abstraction, beyond quantum circuits, (iv) test hybrid quantum applications, (v) consider practical constraints such as limited quantum computing resources, (vi) produce scalable test cases, (vii) generate mutants that correspond to real bugs, and (viii) benefit from classic artificial intelligence by taking advantage of quantum artificial intelligence for classic software testing debugging.

8.1.6 Quantum Software Engineering and Artificial Intelligence.

State of the Art. Quantum for artificial intelligence leverages quantum computing to improve artificial intelligence algorithms, such as speeding up machine learning algorithms [20], implementing neural networks with quantum circuits [81], and applying quantum algorithms (*for instance*, quantum approximate optimization algorithm and variational quantum eigensolver) to solve complex optimization problems directly relevant to artificial intelligence tasks. Various approaches to *artificial intelligence for quantum software engineering* aim to power quantum software developers with large language model-based code assistants to lower the quantum programming learning curve. Some artificial intelligence approaches aim to mitigate noise in quantum software output to facilitate quantum software testing.

Challenges and Trends. A main open challenge is the use of artificial intelligence to generate quantum programs and optimize quantum circuits, as well as to educate and train the next generation of quantum software engineers. A complementary challenge is the definition of quantum algorithms to efficiently address search-based optimization tasks in software engineering, by suitably addressing the unavailability of sufficient and high-quality training data and the lack of large and real-world quantum applications. Murillo et al.'s [134] paper in this special issue discusses the challenge of artificial intelligence and hybrid artificial intelligence-quantum workflows for quantum circuit optimization and quantum error mitigation and correction.

The integration of artificial intelligence and quantum software engineering is a multidisciplinary effort, which is subject to advances in quantum hardware, innovations in quantum algorithms, the

availability of high-quality training data, and the advances of quantum software engineering in developing quantum applications.

8.2 Roadmap

— *What system thinking for quantum software engineering?* Quantum software engineering is complex and requires a holistic and interdisciplinary cross-layer vision. System thinking has been around since the seventh decade of the last century as a way of making sense of the world's complexity from a holistic rather than a partitioned viewpoint, the typical computational thinking that we often practice. We need a system thinking perspective to mitigate challenges of scoping, enable large-scale development, and manage structural change impacts [58, 152].

— *How to align quantum software engineering with classic software engineering?* We need to align quantum software engineering with classic software engineering to define hybrid quantum-classic software systems, a key trend in quantum software engineering, and to ensure that both computing paradigms can work together by optimally leveraging both paradigms and supporting the evolution of quantum hardware with smooth transitions.

— *How to assess the energy efficiency of quantum software systems?* Quantum computing is extremely energy demanding, and energy efficiency plays a key role for the large-scale usage of quantum software systems. We need an increase in efficiency that often comes from a *rebound effect* in terms of the usage of the primary source [4].

— *How to educate quantum software engineers?* We need new curricula to educate the next generation of quantum software engineers. We need to extend computer science curricula with the core curriculum foundations of quantum mechanics, linear algebra, and system thinking.

9 2030 Research Horizon

This special issue outlines the research frontier of software engineering toward 2030 and beyond. Here we summarize the key issues that challenge software engineering research and practice, and that we identify and discuss in this editorial. The many papers in this special issue flesh out the main challenges in different areas of software engineering from various perspectives, aiming to offer a broad viewpoint of the open challenges and mitigate the risks of biases. We identify the main challenges in seven hot areas:

— *Artificial Intelligence and Software Engineering*: Artificial intelligence is dramatically transforming the software engineering paradigms across the board, not only in tooling and automation, but also in developer collaboration and decision-making. We envision a dramatic paradigm shift in software engineering. The core challenge is:
What software engineering paradigm in the artificial intelligence era?
The detailed concerns are about *domain specific design patterns*, *contamination-free benchmarks*, *software engineering life cycles*, and *interpretability techniques*

— *Human Aspects and Software Engineering*: Human aspects in developing and using artificial intelligence systems shape the way artificial intelligence interacts with society, and raise ethical, educational, and inclusiveness issues. We envision a completely new role for both software engineers, who design and develop software systems, and humans, who use artificial intelligence-powered software systems. Artificial intelligence bots will be active teammates and not just tools for software engineers. Humans will be active elements of software ecosystems and not mere users of software systems. The core challenges are:
What role of software engineers?

What role of humans in artificial intelligence-powered ecosystems?
The detailed concerns are about *hybrid human–artificial intelligence teams*, *equitable and socially responsible artificial intelligence-powered software systems*, *transparency, user control, privacy, and adaptability*, *computational empathy*, *adaptability to evolving needs*, and *code of ethics*.

— *Sustainable Software Engineering*: Software and artificial intelligence systems increasingly impact sustainability. At the same time, artificial intelligence-powered software systems offer enormous opportunities to improve sustainability. We envision a major impact of both sustainability in software engineering, that is, *green* software engineering, and software engineering for sustainability, that is, the role of software engineering in sustainability. The core challenges are:
What sustainability in software engineering?
What software engineering for sustainability?
The detailed concerns are about *interdisciplinary collaboration*, *KPIs*, *education for sustainability*, and *ethical and sustainable artificial intelligence*.

— *Automatic Programming*: Automatic programming is shifting the focus from programming at scale to programming with trust. We envision artificial intelligence-powered system generating large-scale programs from artifacts in natural language as well as other human accessible means that describe humans' needs. The role of software engineers and developers will shift from designing and programming code to controlling the front-end artificial intelligence-powered programmer, and certifying the back-end final automatically generated programs. The core challenge is:
What automatic programming for generating useful and trustworthy software systems?
The detailed concerns are about *scalable automatic programming*, *integration of automatically generated code with human-designed and legacy code*, *transparent and trustworthy automatic code generation*, *auditing automatically generated code*, *trustworthy and ethical automatically generated code*.

— *Security and Software Engineering*: Artificial intelligence opens new opportunities for and poses a threat to security. On one side, there is increased automation via the use of large language models in core security technologies to protect critical digital infrastructures. This includes techniques for vulnerability detection as well as vulnerability remediation. On the other side, automatically generated code opens new challenges to security. We envision artificial intelligence-powered tools to protect software ecosystems under human control. The core challenges are:
What artificial intelligence-powered approaches to protect critical digital infrastructures?
How to prevent security breaches in automatically generated code?
The detailed concerns are about *combining large language models and neural networks with classic vulnerability analysis*, *trustworthy automatic bug detection and fix*, *vulnerability issues in automatically generated code*.

— *Verification and Validation*: Artificial intelligence offers opportunities and raises new challenges for verification and validation. On one side, artificial intelligence enhances automatic testing and verification. On the other side, both artificial intelligence-powered ecosystems and automatically generated code challenge quality engineers with new types of bugs. We envision autonomic testing tools that continuously check both functional and non-functional properties and automatically fix bugs that emerge while software ecosystems evolve and adapt. The main challenge is:
How to reveal functional and non-functional bugs that emerge in production in human-centric cyber-physical ecosystems?

The detailed concerns are about *automatically generating valid test suites*, *assessing automatically generated code*, *verifying non-functional properties*, *verifying cyber-physical ecosystems*, *verifying evolving and adaptive systems*, *verifying human centric software ecosystems*.

— *Quantum Software Engineering*: Quantum computing upsets the programming landscape with new challenges for engineering software for quantum clusters. Quantum computing violates the core hypothesis of all classic software engineering approaches and requires new approaches at every stage of the development process. We envision a new quantum engineering paradigm to efficiently program large quantum computers. The main challenge is as follows.

What paradigm for engineering quantum software?

The detailed concerns are about *system thinking for an explicit cross-layer vision*, *aligning quantum software with classic software engineering*, *assessing the energy efficiency of quantum software systems*.

We live in a fast-changing era, and any roadmap quickly ages. We aim to offer a living roadmap that adjusts according to progress and innovation. We will incrementally update the roadmap with a community effort that we will collect with targeted events and workshops, and we will share the updated roadmap in incremental ACM TOSEM special issues.

References

[1] Silvia Abrahão, John Grundy, Mauro Pezzè, Margaret-Anne Storey, and Damian Andrew Tamburri. 2025. Software engineering by and for humans in an AI era. *ACM Transaction on Software Engineering and Methodology,* 2030 Roadmap Special Issue, 1049-33X.

[2] Zeynep Akata, Dan Balliet, Maarten de Rijke, Frank Dignum, Virginia Dignum, Guszti Eiben, Antske Fokkens, Davide Grossi, Koen V. Hindriks, Holger H. Hoos, et al. 2020. A research agenda for hybrid intelligence: Augmenting human intellect with collaborative, adaptive, responsible, and explainable artificial intelligence. *Computer* 53, 8 (2020), 18–28. DOI: https://doi.org/10.1109/MC.2020.2996587

[3] Mohammad Alahmadi, Abdulkarim Khormi, Biswas Parajuli, Jonathan Hassel, Sonia Haiduc, and Piyush Kumar. 2020. Code localization in programming screencasts. *Empirical Software Engineering* 25, 2 (2020), 1536–1572.

[4] Blake Alcott. 2005. Jevons' paradox. *Ecological Economics* 54, 1 (2005), 9–21.

[5] Shaukat Ali, Tao Yue, and Rui Abreu. 2022. When software engineering meets quantum computing. *Communications of the ACM* 65, 4 (2022), 84–88. DOI: https://doi.org/10.1145/3512340

[6] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE '24)*. ACM, New York, NY, 185–196. DOI: https://doi.org/10.1145/3663529.3663839

[7] Saleema Amershi, Daniel S. Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi T. Iqbal, Paul N. Bennett, Kori Inkpen, et al. 2019. Guidelines for human-AI interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.), ACM, 3. DOI: https://doi.org/10.1145/3290605.3300233

[8] Wesley Assunção, Luciano Marchezan, Lawrence Arkoh, Alexander Egyed, and Rudolf Ramler. 2025. Contemporary software modernization: Strategies, driving forces, and research opportunities. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[9] Marco Autili, Martina De Sanctis, Paola Inverardi, and Patrizio Pelliccione. 2025. Engineering digital systems for humanity: A research roadmap. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[10] C. L. Baker, R. Saxe, and J. Tenenbaum. 2009. Action understanding as inverse planning. *Cognition* 113, 3 (2009), 329–349.

[11] 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.

[12] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, et al. 2020. Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion* 58, C (2020), 82–115. DOI: https://doi.org/10.1016/j.inffus.2019.12.012

[13] Christoph Becker. 2023. *Insolvent: How to Reorient Computing for Just Sustainability*. MIT Press.
[14] Christoph Becker, Stefanie Betz, Ruzanna Chitchyan, Leticia Duboc, Steve M. Easterbrook, Birgit Penzenstadler, Norbet Seyff, and Colin C. Venters. 2015. Requirements: The key to sustainability. *IEEE Software* 33, 1 (2015), 56–65.
[15] Christoph Becker, Ruzanna Chitchyan, Leticia Duboc, Steve Easterbrook, Birgit Penzenstadler, Norbert Seyff, and Colin C. Venters. 2015. Sustainability design and software: The Karlskrona manifesto. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Vol. 2, 467–476.
[16] Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, et al. 2022. A survey of field-based testing techniques. *ACM Computing Surveys* 54, 5 (2022), 1–39. DOI: https://doi.org/10.1145/3447240
[17] Stefanie Betz and Birgit Penzenstadler. 2025. With great power comes great responsibility: The role of software engineers. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.
[18] Stefanie Betz, Birgit Penzenstadler, Leticia Duboc, Ruzanna Chitchyan, Sedef Akinli Kocak, Ian Brooks, Shola Oyedeji, Jari Porras, Norbert Seyff, and Colin C. Venters. 2024. Lessons learned from developing a sustainability awareness framework for software engineering using design science. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–39.
[19] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative AI: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code'24)*. ACM, New York, NY, 54–61. DOI: https://doi.org/10.1145/3643795.3648396
[20] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (2017), 195–202.
[21] Christian Birchler, Sajad Khatiri, Pooja Rani, Timo Kehrer, and Sebastiano Panichella. 2025. A roadmap for simulation-based testing of autonomous cyber-physical systems: Challenges and future direction. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.
[22] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2022. Call me maybe: Using NLP to automatically generate unit test cases respecting temporal constraints. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. ACM, 19:1–19:11. DOI: https://doi.org/10.1145/3551349.3556961
[23] Michael Brooks. 2023. The race to find quantum computing's sweet spot. Nature 617, 7962 (May 2023), S1–S3. DOI:10.1038/d41586-023-01692-9
[24] Gro Harlem Brundtland. 1987. *Our Common Future World Commission on Environment and Developement*. UN World Commission on Environment and Development.
[25] Lola Burgueño, Davide Di Ruscio, Houari Sahraoui, and Manuel WImmer. 2025. Automation in model-driven engineering: A look back and ahead. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.
[26] Marcel Böhme, Eric Bodden, Tevfik Bultan, Cristian Cadar, Yang Liu, and Giuseppe Scanniello. 2025. Software security analysis in 2030 and beyond: A research roadmap. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.
[27] Coral Calero and Mario Piattini. 2015. *Green in Software Engineering*, Vol. 3, Springer.
[28] 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.
[29] Roberto Casadei, Gianluca Aguzzi, Giorgio Audrito, Ferruccio Damiani, Danilo Pianini, Giordano Scarso, Gianluca Torta, and Mirko Viroli. 2025. Software engineering for collective cyber-physical ecosystems. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.
[30] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. 2019. Studying the difference between natural and programming language corpora. *Empirical Software Engineering* 24, 4 (2019), 1823–1868. DOI: https://doi.org/10.1007/s10664-018-9669-7
[31] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, et al. 2022. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. arXiv:2208.08227. Retrieved from http://arxiv.org/abs/2208.08227
[32] Johan Cederbladh, Antonio Cicchetti, and Robbert Jongeling. n.d. A road-map to readily available early validation & verification of system behaviour in model-based systems engineering using software engineering best practices. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.
[33] Alexander Chatzigeorgiou, Iftekar Ahmed, Haipeng Cai, Mauro Pezzè, and Denys Poshyvanyk. 2024. Artificial intelligence for software engineering: The journey so far and the road ahead. *ACM Transactions on Software Engineering and Methodology* 34 (2024), 9.
[34] 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.

[35] Daihang Chen, Yonghui Liu, Mingyi Zhou, Yanjie Zhao, Haoyu Wang, Shuai Wang, Xiao Chen, Tegawende Bissyande, Jacques Klein, and Li Li. 2025. LLM for mobile: An initial roadmap. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[36] 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 *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 364–375.

[37] 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:2107.03374. Retrieved from http://arxiv.org/abs/2107.03374

[38] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: A new approach for generating next test cases. arXiv:2002.12543. Retrieved from https://arxiv.org/abs/2002.12543

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

[40] Yuxing Cheng, Yi Chang, and Yuan Wu. 2025. A survey on data contamination for large language models. arXiv:2502.14425. Retrieved from https://arxiv.org/abs/2502.14425

[41] 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. arXiv:2108.01585. Retrieved from https://arxiv.org/abs/2108.01585

[42] Emilia Cioroaica, Thomas Kuhn, and Thomas Bauer. 2018. Prototyping automotive smart ecosystems. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. IEEE Computer Society, 255–262. DOI: https://doi.org/10.1109/DSN-W.2018.00072

[43] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. 2021. Counterfactual explanations for models of code. arXiv:2111.05711. Retrieved from https://arxiv.org/abs/2111.05711

[44] Jane Cleland-Huang, Ankit Agrawal, Michael Vierhauser, Michael Murphy, and Mike Prieto. 2022. Extending MAPE-K to support human-machine teaming. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '22)*. Bradley R.Schmerl, MartinaMaggio, and JavierCámara (Eds.), ACM/IEEE, 120–131. DOI: https://doi.org/10.1145/3524844.3528054

[45] Luis Cruz, Xavier Franch, and Silverio Martínez-Fernández. 2025. Innovating for tomorrow: The convergence of SE and green AI. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[46] 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.

[47] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171, C (2024), 107468. DOI: https://doi.org/10.1016/j.infsof.2024.107468

[48] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2018. Explainable software analytics. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '18)*. ACM, New York, NY, 53–56. DOI: https://doi.org/10.1145/3183399.3183424

[49] Eliot Zackrone, Beiza Eken, Michal Ernst, Mauro Pezzè, Davide Molinelli, and Alberto Martin-Lopez. 2025. Tratto: A neuro-symbolic approach to deriving axiomatic test oracles. In *Proceedings of the ACM ISSTA International Symposium on Software Testing and Analysis*.

[50] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley R. Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, et al. 2010. Software engineering for self-adaptive systems: A second research roadmap. In *Proceedings of the International Seminar on Software Engineering for Self-Adaptive Systems II*. Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw (Eds.), *Lecture Notes in Computer Science, Vol. 7475*, Springer, 1–32. DOI: https://doi.org/10.1007/978-3-642-35813-5_1

[51] Santos de Souza Ronnie, Felipe Fronchetti, Sávio Freire, and Rodrigo Spinola. 2025. Software fairness debt: Building a research agenda for addressing bias in AI systems. *ACM Transaction on Software Engineering and Methodology 2030 Roadmap Special Issue,* 1049-331X.

[52] 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. Retrieved from https://arxiv.org/abs/2009.08525

[53] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu Lahiri. 2022. TOGA: A neural method for test oracle generation. In *Proceedings of the International Conference on Software Engineering (ICSE '22)*. ACM. Retrieved from https://www.microsoft.com/en-us/research/publication/toga-a-neural-method-for-test-oracle-generation/

[54] Leticia Duboc, Birgit Penzenstadler, Jari Porras, Sedef Akinli Kocak, Stefanie Betz, Ruzanna Chitchyan, Ola Leifler, Norbert Seyff, and Colin C. Venters. 2020. Requirements engineering for sustainability: An awareness framework for designing software systems for a better tomorrow. *Requirements Engineering* 25, 4 (2020), 469–492.

[55] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, 530–541. DOI: https://doi.org/10.1145/3377811.3380364

[56] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, et al. 2022. Toy models of superposition. arXiv:2209.10652. Retrieved from https://arxiv.org/abs/2209.10652

[57] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

[58] Jay W. Forrester. 1987. Lessons from system dynamics modeling. *System Dynamics Review* 3, 2 (1987), 136–149.

[59] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, 416–419. DOI: https://doi.org/10.1145/2025113.2025179

[60] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. 39, 2 (2013), 276–291.

[61] Alessio Gambi, Shreya Mathews, Benedikt Steininger, Mykhailo Poienko, and David Bobek. 2024. The flexcrash platform for testing autonomous vehicles in mixed-traffic scenarios. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. Maria Christakis and Michael Pradel (Eds.), ACM, 1811–1815. DOI: https://doi.org/10.1145/3650212.3685299

[62] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. 2025. The current challenges of software engineering in the era of large language models. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[63] 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 *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 761–773. DOI: https://doi.org/10.1109/ASE56229.2023.00109

[64] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[65] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Communications of the ACM* 62, 12 (2019), 56–65.

[66] Michael Grieves and John Vickers. 2017. *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*. Springer International Publishing, Cham, 85–113. DOI: https://doi.org/10.1007/978-3-319-38756-7_4

[67] 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.

[68] 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.

[69] Achim Guldner, Rabea Bender, Coral Calero, Giovanni S. Fernando, Markus Funke, Jens Gröger, Lorenz M. Hilty, Julian Hörnschemeyer, Geerd-Dietger Hoffmann, Dennis Junger, et al. 2024. Development and evaluation of a reference measurement model for assessing the resource and energy efficiency of software products and components—Green software measurement model (GSMM). *Future Generation Computer Systems* 155 (2024), 402–418.

[70] Guoxiang Guo, Aldeida Aleti, Neelofar Neelofar, and Chakkrit Tantithamthavorn. 2024. MORTAR: Metamorphic multi-turn testing for LLM-based dialogue systems. arXiv:2412.15557. Retrieved from https://arxiv.org/abs/2412.15557

[71] Susmita Haldar and Luiz Fernando Capretz. 2024. Interpretable software maintenance and support effort prediction using machine learning. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE Computer Society, Los Alamitos, CA, USA, 288–289. DOI: https://doi.org/10.1145/3639478.3643069

[72] George Halkos and Eleni-Christina Gkampoura. 2021. Where do we stand on the 17 sustainable development goals? An overview on progress. *Economic Analysis and Policy* 70 (2021), 94–122.

[73] Junda He, Christoph Treude, and David Lo. 2025. LLM-based multi-agent systems for software engineering: Literature review, vision and the road ahead. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[74] 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.

[75] Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. 2024. Do large code models understand programming concepts? Counterfactual analysis for code predicates. In *Proceedings of the 41st International Conference on Machine Learning (ICML'24)*. Article 753, 11 pages.

[76] Soneya Binta Hossain and Matthew Dwyer. 2024. TOGLL: Correct and strong test oracle generation with LLMs. arXiv:2405.03786. Retrieved from https://arxiv.org/abs/2405.03786

[77] Hui Huang, Yingqi Qu, Hongli Zhou, Jing Liu, Muyun Yang, Bing Xu, and Tiejun Zhao. 2024. On the limitations of fine-tuned judge models for LLM evaluation. arXiv:2403.02839. Retrieved from https://arxiv.org/abs/2403.02839

[78] Sonja Hyrynsalmi, Sebastian Baltes, Chris Brown, Rafael Prikladnicki, Perez Gema Rodriguez, Alexander Serebrenik, Jocelyn Simmonds, Bianca Trinkenreich, and Yi Wang. 2025. Bridging gaps, building futures: Advancing software developer diversity and inclusion through future-oriented research. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[79] Victoria Jackson, Bogdan Vasilescu, Daniel Russo, Paul Ralph, Rafael Prikladnicki, Maliheh Izadi, Sarah D'Angelo, Sarah Inman, Anielle Lisboa, and André van der Hoek. 2025. The impact of generative AI on creativity in software development: A research agenda. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[80] Gunel Jahangirova and Valerio Terragni. 2023. SBFT tool competition 2023 – Java test case generation track. In *Proceedings of the 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 61–64. DOI: https://doi.org/10.1109/SBFT59156.2023.00025

[81] Weiwen Jiang, Jinjun Xiong, and Yiyu Shi. 2021. A co-design framework of neural networks and quantum circuits towards quantum advantage. *Nature Communications* 12, 1 (2021), 579.

[82] 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. DOI: https://doi.org/10.1109/TSE.2020.2982385

[83] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? arXiv:2310.06770. Retrieved from https://arxiv.org/abs/2310.06770

[84] 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 (ISSTA '14)*, 437–440.

[85] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the International Conference on Machine Learning*. PMLR, 5110–5121.

[86] Prabhjot Kaur, Samira Taghavi, Zhaofeng Tian, and Weisong Shi. 2021. A survey on simulators for testing self-driving cars. In *Proceedings of the 2021 4th International Conference on Connected and Autonomous Driving (MetroCAD)*, 62–70. DOI: https://doi.org/10.1109/MetroCAD51599.2021.00018

[87] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50. DOI: https://doi.org/10.1109/MC.2003.1160055

[88] Marcus Kessel and Colin Atkinson. 2025. Morescient GAI for software engineering. *ACM Transaction on Software Engineering and Methodology 2030 Roadmap Special Issue*.

[89] Jayden Khakurel, Birgit Penzenstadler, Jari Porras, Antti Knutas, and Wenlu Zhang. 2018. The rise of artificial intelligence under the lens of sustainability. *Technologies* 6, 4 (2018), 100. DOI: https://doi.org/10.3390/technologies6040100

[90] 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.

[91] Dipin Khati, Yijin Liu, David N. Palacio, Yixuan Zhang, and Denys Poshyvanyk. 2025. Mapping the trust terrain: LLMs in software engineering – Insights and perspectives. arXiv:2503.13793. Retrieved from https://arxiv.org/abs/2503.13793

[92] 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:2211.05286. Retrieved from https://arxiv.org/abs/2211.05286

[93] Jörg Kienzle, Gunter Mussbacher, Benoit Combemale, Lucy Bastin, Nelly Bencomo, Jean-Michel Bruel, Christoph Becker, Stefanie Betz, Ruzanna Chitchyan, Betty H. C. Cheng, et al. 2020. Toward model-driven sustainability evaluation. *Communications of the ACM* 63, 3 (2020), 80–91.

[94] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. 2024. Do LLMs generate test oracles that capture the actual or the expected program behaviour? arXiv:2410.21136. Retrieved from https://arxiv.org/abs/2410.21136

[95] Stefanie Kunkel, Frieder Schmelzle, Silke Niehoff, and Grischa Beier. 2023. More sustainable artificial intelligence systems through stakeholder involvement? *GAIA-Ecological Perspectives for Science and Society* 32, 1 (2023), 64–70.

[96] Dattatray Vishnu Kute, Biswajeet Pradhan, Nagesh Shukla, and Abdullah Alamri. 2021. Deep learning and explainable artificial intelligence techniques applied for detecting money laundering – A critical review. *IEEE Access* 9 (2021), 82300–82317. DOI: https://doi.org/10.1109/ACCESS.2021.3086230

[97] Christoph König, Daniel Lang, and Ina Schaefer. 2025. Sustainable software engineering: Concepts, challenges, and vision. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[98] Patricia Lago, Nelly Condori Fernandez, Iffat Fatima, Markus Funke, and Ivano Malavolta. 2024. The sustainability assessment framework toolkit: A decade of modeling experience. arXiv:2405.01391. Retrieved from https://arxiv.org/abs/2405.01391

[99] Patricia Lago, Sedef Akinli Koçak, Ivica Crnkovic, and Birgit Penzenstadler. 2015. Framing sustainability as a property of software quality. *Communications of the ACM* 58, 10 (2015), 70–78.

[100] Daniel J. Lang, Arnim Wiek, Matthias Bergmann, Michael Stauffacher, Pim Martens, Peter Moll, Mark Swilling, and Christopher J. Thomas. 2012. Transdisciplinary research in sustainability science: Practice, principles, and challenges. *Sustainability Science* 7, S1 (2012), 25–43.

[101] 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.

[102] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 919–931. DOI: https://doi.org/10.1109/ICSE48619.2023.00085

[103] Vanessa Levesque. 2019. Sustainability Methods & Perspectives. Retrieved from https://pressbooks.pub/sustainabilitymethods/

[104] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. arXiv:1711.09573. Retrieved from https://arxiv.org/abs/1711.09573

[105] 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 *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 208–219.

[106] Rui Li, Huai Liu, Pak-Lok Poon, Dave Towey, Chang Ai Sun, Zheng Zheng, Zhi Quan Zhou, and Tsong Chen. 2025. Metamorphic relation generation: State of the art and research directions. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue (April 2025).

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

[108] Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W. Eric Wong. 2025. Evaluating large language models for software testing. *Computer Standards & Interfaces* 93 (2025), 103942. DOI: https://doi.org/10.1016/j.csi.2024.103942

[109] J. C. R. Licklider. 1960. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics HFE-1* HFE-1, 1 (1960), 4–11. DOI: https://doi.org/10.1109/THFE2.1960.4503259

[110] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT prompt for code generation. arXiv:2305.08360. Retrieved from https://arxiv.org/abs/2305.08360

[111] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. arXiv:2305.01210. Retrieved from https://arxiv.org/abs/2305.01210

[112] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2024. On the reliability and explainability of language models for program generation. arXiv:2302.09587. Retrieved from https://arxiv.org/abs/2302.09587

[113] 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:2102.04664. Retrieved from https://arxiv.org/abs/2102.04664

[114] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. 2023. Improving requirements completeness: Automated assistance through large language models. arXiv:2308.03784. Retrieved from https://arxiv.org/abs/2308.03784

[115] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE '22)*. ACM, New York, NY, 168–172. DOI: https://doi.org/10.1145/3510454.3516829

[116] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4768–4777.

[117] Michael Lyu, Ray Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. 2025. Automatic programming: Large language models and beyond. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[118] Jing Ma, Xiaobo Che, Yanqiang Li, and Edmund M.-K. Lai. 2021. Traffic scenarios for automated vehicle testing: A review of description languages and systems. *Machines* 9, 12 (2021), 342. DOI: https://doi.org/10.3390/machines9120342

[119] Wei Ma, Shangqing Liu, Mengjie Zhao, Xiaofei Xie, Wenhan Wang, Qiang Hu, Jie Zhang, and Yang Liu. 2024. Unveiling code pre-trained models: Investigating syntax and semantics capacities. arXiv:2212.10017. Retrieved from https://arxiv.org/abs/2212.10017

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

[121] Zohar Manna and Richard J. Waldinger. 1971. Toward automatic program synthesis. *Communications of the ACM* 14, 3 (1971), 151–165. DOI: https://doi.org/10.1145/362566.362568

[122] 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. Retrieved from https://arxiv.org/abs/2206.08574

[123] Antonio Mastropaolo, Camilo Escobar-Velásquez, and Mario Linares-Vásquez. 2025. From triumph to uncertainty: The journey of software engineering in the AI era. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[124] Marshall McLuhan. 1977. Laws of the media. *ETC: A Review of General Semantics* 34, 2 (1977), 173–179. DOI: https://doi.org/10.20944/preprints202104.0526.v1

[125] Donella H. Meadows, Dennis L. Meadows, Jørgen Randers, and William W. Behrens III. 1972. The Limits to Growth - Club of Rome. Retrieved from https://policycommons.net/artifacts/1529440/the-limits-to-growth/2219251/

[126] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[127] Manoranjan Mishra, Sudarsan Desul, Celso Augusto Guimarães Santos, Shailendra Kumar Mishra, Abu Hena Mustafa Kamal, Shreerup Goswami, Ahmed Mukalazi Kalumba, Ramakrishna Biswal, Richarde Marques da Silva, Carlos Antonio Costa Dos Santos, et al. 2024. A bibliometric analysis of sustainable development goals (SDGs): A review of progress, challenges, and opportunities. *Environment, Development and Sustainability* 26, 5 (2024), 1–43.

[128] Ahmad Haji Mohammadkhani, Chakkrit Tantithamthavorn, and Hadi Hemmati. 2023. Explainable AI for pre-trained code models: What do they learn? When they do not work? arXiv:2211.12821. Retrieved from https://arxiv.org/abs/2211.12821

[129] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. 2025. Test oracle automation in the era of LLMs. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[130] 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. DOI: https://doi.org/10.1109/TSE.2018.2844788

[131] Ana Moreira, Patricia Lago, Rogardt Heldal, Stefanie Betz, Ian Brooks, Rafael Capilla, Vlad Coroama, Leticia Duboc, Joao Fernandes, Ola Leifler, et al. 2025. A roadmap for integrating sustainability into software engineering education. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[132] Noura El Moussa, Davide Molinelli, Mauro Pezzè, and Martin Tappler. 2021. Health of smart ecosystems. In *Proceedings of ESEC/FSE Ideas, Visions and Reflections (ESEC/FSE '21)*. DOI: https://doi.org/10.1145/3468264.3473137

[133] Jose Juan M. Murillo, Frank Johanna Barzen, Shaukat Ali, Tao Yue, Paolo Arcaini, Ignacio García Ricardo Pérez, Antonio Ruiz-Cortés, Antonio Brogi, Jianjun Zhao, Andriy Miranskyy, et al. 2025. Challenges of quantum software engineering for the next decade: The road ahead. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. arXiv:2404.06825. Retrieved from https://arxiv.org/abs/2404.06825

[134] Juan Manuel Murillo, Jose Garcia-Alonso, Enrique Moguel, Johanna Barzen, Frank Leymann, Shaukat Ali, Tao Yue, Paolo Arcaini, Ricardo Perez-Castillo, Ignacio García-Rodríguez de Guzmán, et al. 2025. Quantum software engineering: Roadmap and challenges ahead. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[135] 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, 5 (2024), 1215–1243. DOI: https://doi.org/10.1109/tse.2024.3379943

[136] Stefan Naumann, Markus Dick, Eva Kern, and Timo Johann. 2011. The GREENSOFT model: A reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems* 1, 4 (2011), 294–304.

[137] Higor A. de Souza, Neilson C. L. Ramalho, and Marcos L. Chaim. 2025. Testing and debugging quantum programs: The road to 2030. *ACM Transactions on Software Engineering and Methodology (TOSEM)* Issue TOSEM-2024-0251.

[138] 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.

[139] Curtis G. Northcutt, Anish Athalye, and Jonas Mueller. 2021. Pervasive label errors in test sets destabilize machine learning benchmarks. arXiv:2103.14749. Retrieved from https://arxiv.org/abs/2103.14749

[140] Wendkuuni C. Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F. Bissyande. 2024. LLMs and prompting for unit test generation: A large-scale evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. ACM, New York, NY, 2464–2465. DOI: https://doi.org/10.1145/3691620.3695330

[141] Shola Oyedeji, Ahmed Seffah, and Birgit Penzenstadler. 2018. A catalogue supporting software sustainability design. *Sustainability* 10, 7 (2018), 2296.

[142] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed random testing for java. In *Proceedings of the Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA '07)*. ACM, New York, NY, 815–816. DOI: https://doi.org/10.1145/1297846.1297902

[143] 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. arXiv:2407.08983. Retrieved from https://arxiv.org/abs/2407.08983

[144] David N. Palacio, Alejandro Velasco, Daniel Rodriguez-Cardenas, Kevin Moran, and Denys Poshyvanyk. 2023. Evaluating and explaining large language models for code using syntactic structures. arXiv:2308.03873. Retrieved from https://arxiv.org/abs/2308.03873

[145] 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.

[146] Pat Pataranutaporn, Ruby Liu, Ed Finn, and Pattie Maes. 2023. Influencing human-AI interaction by priming beliefs about AI can increase perceived trustworthiness, empathy and effectiveness. *Nature Machine Intelligence* 5, 10 (2023), 1076–1086. DOI: https://doi.org/10.1038/S42256-023-00720-7

[147] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2339–2356.

[148] Birgit Penzenstadler. 2013. Towards a definition of sustainability in and for software engineering. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 1183–1185.

[149] Birgit Penzenstadler. 2014. Infusing green: Requirements engineering for green In and through software systems. In *Proceedings of the International Workshop on Requirements Engineering for Sustainable Systems (RE4SuSy@ RE)*, 44–53.

[150] Birgit Penzenstadler. 2018. Sustainability analysis and ease of learning in artifact-based requirements engineering: The newest member of the family of studies (it'sa girl!). *Information and Software Technology* 95 (2018), 130–146.

[151] Birgit Penzenstadler, Stefanie Betz, Leticia Duboc, Norbert Seyff, Jari Porras, Shola Oyedeji, Ian Brooks, and Colin C. Venters. 2021. Iterative sustainability impact assessment: When to propose? In *Proceedings of the 2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS)*. IEEE, 5–6.

[152] Birgit Penzenstadler, Leticia Duboc, Colin C. Venters, Stefanie Betz, Norbert Seyff, Krzsztof Wnuk, Ruzanna Chitchyan, Steve M. Easterbrook, and Christoph Becker. 2018. Software engineering for sustainability: Find the leverage points! *IEEE Software* 35, 4 (2018), 22–33.

[153] Birgit Penzenstadler, Henning Femmer, and Debra Richardson. 2013. Who is the advocate? Stakeholders for sustainability. In *Proceedings of the 2013 2nd International Workshop on Green and Sustainable Software (GREENS)*. IEEE, 70–77.

[154] Birgit Penzenstadler and Andreas Fleischmann. 2011. Teach sustainability in software engineering? In *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 454–458.

[155] Anne-Kathrin Peters, Rafael Capilla, Vlad Constantin Coroamă, Rogardt Heldal, Patricia Lago, Ola Leifler, Ana Moreira, Joao Paulo Fernandes, Birgit Penzenstadler, Jari Porras, et al. 2024. Sustainability in computing education: A systematic literature review. *ACM Transactions on Computing Education* 24, 1 (2024), 1–53.

[156] Mauro Pezzè and Michal Young. 2007. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley.

[157] Johanna Pohl, Lorenz M. Hilty, and Matthias Finkbeiner. 2019. How LCA contributes to the environmental assessment of higher order effects of ICT application: A review of different approaches. *Journal of Cleaner Production* 219 (2019), 698–712.

[158] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. PyExplainer: Explaining the predictions of just-in-time defect models. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 407–418. DOI: https://doi.org/10.1109/ASE51524.2021.9678763

[159] Jari Porras, Colin C. Venters, Birgit Penzenstadler, Leticia Duboc, Stefanie Betz, Norbert Seyff, Saeid Heshmatisafa, and Shola Oyedeji. 2021. How could we have known? Anticipating sustainability effects of a software product. In *Proceedings of the 12th International Conference Software Business (ICSOB '21)*. Springer, 10–17.

[160] Ketai Qiu, Niccolò Puccinelli, Matteo Ciniselli, and Luca Di Grazia. 2025. From today's code to tomorrow's symphony: The AI transformation of developer's routine by 2030. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[161] Anichur Rahman, Md Sazzad Hossain, Ghulam Muhammad, Dipanjali Kundu, Tanoy Debnath, Muaz Rahman, Md Saikat Islam Khan, Prayag Tiwari, and Shahab S. Band. 2023. Federated learning-based AI approaches in smart healthcare: concepts, taxonomies, challenges and open issues. *Cluster Computing* 26, 4 (2023), 1–41. DOI: https://doi.org/10.1007/S10586-022-03658-4

[162] Neilson Ramalho, Higor Amario de Souza, and Marcos Lordello Chaim. 2025. Testing and debugging quantum programs: The road to 2030. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[163] Dezhi Ran, Mengzhou Wu, Wei Yang, and Tao Xie. 2025. Foundation model engineering: Engineering foundation models just as engineering software. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue, 1049-331X.

[164] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?": Explaining the predictions of any classifier. arXiv:1602.04938. Retrieved from https://arxiv.org/abs/1602.04938

[165] 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.

[166] Diana Robinson, Christian Cabrera, Andrew Gordon, Neil Lawrence, and Lars Mennen. 2025. Requirements are all you need: The final frontier for End-User software engineering. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[167] Daniel Rodriguez-Cardenas. 2024. Beyond accuracy and robustness metrics for large language models for code. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24)*. ACM, New York, NY, 159–161. DOI: https://doi.org/10.1145/3639478.3639792

[168] Daniel Rodriguez-Cardenas, David N. Palacio, Dipin Khati, Henry Burke, and Denys Poshyvanyk. 2023. Benchmarking causal study to interpret large language models for source code. arXiv:2308.12415. Retrieved from https://arxiv.org/abs/2308.12415

[169] Abhik Roychoudhury, Corina Pasareanu, Michael Pradel, and Baishakhi Ray. 2025. AI software engineer: Programming with trust. arXiv:2502.13767. Retrieved from https://arxiv.org/abs/2502.13767

[170] Daniel Russo, Sebastian Baltes, Niels van Berkel, Paris Avgeriou, Fabio Calefato, Beatriz Cabrero Daniel, Gemma Catolino, Jürgen Cito, Neil A. Ernst, Thomas Fritz, et al. 2024. Generative AI in software engineering must be human-centered: The Copenhagen manifesto. *Journal of System and Software* 216 (2024), 112115. DOI: https://doi.org/10.1016/J.JSS.2024.112115

[171] Jeffrey D. Sachs. 2012. From millennium development goals to sustainable development goals. *The Lancet* 379, 9832 (2012), 2206–2211.

[172] Jeffrey D. Sachs, Guido Schmidt-Traub, Mariana Mazzucato, Dirk Messner, Nebojsa Nakicenovic, and Johan Rockström. 2019. Six transformations to achieve the sustainable development goals. *Nature Sustainability* 2, 9 (2019), 805–814.

[173] Mozhgan Salimiparsa, Surajsinh Parmar, San Lee, Choongmin Kim, Yonghwan Kim, and Jang Yong Kim. 2023. Investigating poor performance regions of black boxes: LIME-based exploration in sepsis detection. arXiv:2306.12507. Retrieved from https://arxiv.org/abs/2306.12507

[174] Theresia Ratih Dewi Saputri and Seok-Won Lee. 2020. Addressing sustainability in the requirements engineering process: From elicitation to functional decomposition. *Journal of Software: Evolution and Process* 32, 8 (2020), e2254.

[175] Iqbal H. Sarker, Helge Janicke, Ahmad Mohsin, Asif Gill, and Leandros Maglaras. 2024. Explainable AI for cybersecurity automation, intelligence and trustworthiness in digital twin: Methods, taxonomy, challenges and prospects. *ICT Express* 10, 4 (2024), 935–958. DOI: https://doi.org/10.1016/J.ICTE.2024.05.007

[176] J. Schneider-Hufschmidt, T. Kühme, and U. Malinowski. 1993. *Adaptive User Interfaces: Principles and Practice*. North Holland, London.

[177] Lukas Schulte, Benjamin Ledel, and Steffen Herbold. 2024. Studying the explanations for the automated prediction of bug and non-bug issues using LIME and SHAP. *Empirical Software Engineering* 29, 4 (2024), 29. DOI: https://doi.org/10.1007/s10664-024-10469-1

[178] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. DOI: https://doi.org/10.1109/TSE.2023.3334955

[179] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824. DOI: https://doi.org/10.1109/TSE.2016.2532875

[180] Norbert Seyff, Stefanie Betz, Leticia Duboc, Colin Venters, Christoph Becker, Ruzanna Chitchyan, Birgit Penzenstadler, and Markus Nöbauer. 2018. Tailoring requirements negotiation to sustainability. In *Proceedings of the 2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 304–314.

[181] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 201–211. DOI: https://doi.org/10.1109/ASE.2015.86

[182] Jieke Shi, Zhou Yang, and David Lo. 2025. Efficient and green large language models for software engineering: Vision and the road ahead. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue (April 2025).

[183] Seung Yeob Shin, Fabrizio Pastore, Domenico Bianculli, and Alexandra Baicoianu. 2024. Towards generating executable metamorphic relations using large language models. arXiv:2401.17019. Retrieved from https://arxiv.org/abs/2401.17019

[184] 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:2411.10877. Retrieved from https://arxiv.org/abs/2411.10877

[185] 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:2502.04484. Retrieved from https://arxiv.org/abs/2502.04484

[186] Yongqiang Sun, Xiao-Liang Shen, and Kem Z. K. Zhang. 2023. Human-AI interaction. *Data and Information Management* 7, 3 (2023), 100048. Retrieved from https://doi.org/10.1016/J.DIM.2023.100048

[187] Joseph Tainter. 2003. A framework for sustainability. *World Futures* 59, 3–4 (2003), 213–223.

[188] Gaigai Tang, Long Zhang, Feng Yang, Lianxiao Meng, Weipeng Cao, Meikang Qiu, Shuangyin Ren, Lin Yang, and Huiqiang Wang. 2021. Interpretation of learning-based automatic source code vulnerability detection model using LIME. In *Proceedings of the 14th International Conference on Knowledge Science, Engineering and Management (KSEM '21)*. Springer-Verlag, Berlin, 275–286. DOI: https://doi.org/10.1007/978-3-030-82153-1_23

[189] Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. 2025.Apri. The future of AI-driven software engineering. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[190] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. 2019. Feature maps: A comprehensible software representation for design pattern detection. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 207–217. DOI: https://doi.org/10.1109/SANER.2019.8667978

[191] 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:2304.11938. Retrieved from https://arxiv.org/abs/2304.11938

[192] 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.

[193] Sergey Troshin and Nadezhda Chirkova. 2022. Probing pretrained models of source code. arXiv:2202.08975. Retrieved from https://arxiv.org/abs/2202.08975

[194] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. 2023. Large language models: The next frontier for variable discovery within metamorphic testing? In *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 678–682. DOI: https://doi.org/10.1109/SANER56733.2023.00070

[195] 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:2410.22663. Retrieved from https://arxiv.org/abs/2410.22663

[196] Keyon Vafa, Yuntian Deng, David M. Blei, and Alexander M. Rush. 2021. Rationales for sequential predictions. arXiv:2109.06387. Retrieved from https://arxiv.org/abs/2109.06387

[197] Aimee Van Wynsberghe. 2021. Sustainable AI: AI for sustainability and the sustainability of AI. *AI and Ethics* 1, 3 (2021), 213–218.

[198] Alejandro Velasco, Aya Garryyeva, David N. Palacio, Antonio Mastropaolo, and Denys Poshyvanyk. 2025. Toward neurosymbolic program comprehension. arXiv:2502.01806. Retrieved from https://arxiv.org/abs/2502.01806

[199] Alejandro Velasco, David N. Palacio, Daniel Rodriguez-Cardenas, and Denys Poshyvanyk. 2024. Which syntactic capabilities are statistically learned by masked language models for code? In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'24)*. ACM, New York, NY, 72–76. DOI: https://doi.org/10.1145/3639476.3639768

[200] Alejandro Velasco, Daniel Rodriguez-Cardenas, Luftar Rahman Alif, David N. Palacio, and Poshyvanyk Denys.2025. How propense are large language models at producing code smells? A benchmarking study. arXiv:2412.18989. Retrieved from https://arxiv.org/abs/2412.18989

[201] Colin Venters, L. M. S. Lau, Michael Griffiths, Violeta Holmes, Rupert Ward, Caroline Jay, Charlie Dibsdale, and Jie Xu. 2014. The blind men and the elephant: Towards an empirical evaluation framework for software sustainability. *Journal of Open Research Software* 2, 1 (2014), e8.

[202] Colin C. Venters, Norbert Seyff, Christoph Becker, Stephanie Betz, Ruzanna Chitchyan, Leticia Duboc, Dan McIntyre, and Birgit Penzenstadler. 2017. Characterising sustainability requirements: A new species. *Red Herring, or Just an Odd Fish, Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Society Track*, 20–28.

[203] Colin C. Venters, Rafael Capilla, Stefanie Betz, Birgit Penzenstadler, Tom Crick, Steve Crouch, Elisa Yumi Nakagawa, Christoph Becker, and Carlos Carrillo. 2018. Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software* 138 (2018), 174–188.

[204] Colin C. Venters, Rafael Capilla, Elisa Yumi Nakagawa, Stefanie Betz, Birgit Penzenstadler, Tom Crick, and Ian Brooks. 2023. Sustainable software engineering: Reflections on advances in research and practice. *Information and Software Technology* 164 (2023), 107316.

[205] Colin C. Venters, Caroline Jay, L. M. S. Lau, Michael K. Griffiths, Violeta Holmes, Rupert R. Ward, Jim Austin, Charlie E. Dibsdale, and Jie Xu. 2014. Software sustainability: The modern tower of Babel. In *Proceedings of the CEUR Workshop* 1216 (2014), 7–12.

[206] Colin C. Venters, Sedef Akinli Kocak, Stefanie Betz, Ian Brooks, Rafael Capilla, Ruzanna Chitchyan, Letícia Duboc, Rogardt Heldal, Ana Moreira, Shola Oyedeji, et al. 2021. Software sustainability: Beyond the tower of babel. In *Proceedings of the 2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS)*. IEEE, 3–4.

[207] 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* 50, 4 (2024), 911–936. DOI: https://doi.org/10.1109/TSE.2024.3368208

[208] Jun Wang, Li Zhang, Yanjun Huang, Jian Zhao, and Francesco Bella. 2020. Safety of autonomous vehicles. *Journal of Advanced Transportation* 2020, 1 (2020), 1–13.

[209] Qing Wang, Junjie Wang, Mingyang Li, and Yawen Wang. 2025. A roadmap for software testing in open-collaborative and AI-powered era. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[210] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022. ReCode: Robustness evaluation of code generation models. arXiv:2212.10264. Retrieved from http://arxiv.org/abs/2212.10264

[211] Shenao Wang, Yanjie Zhao, Xinyi Hou, and Haoyu Wang. 2025. Large language model supply chain: A research agenda. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[212] Wentao Wang, Nan Niu, Hui Liu, and Zhendong Niu. 2018. Enhancing automated requirements traceability by resolving polysemy. In *Proceedings of the 2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 40–51. DOI: https://doi.org/10.1109/RE.2018.00-53

[213] Zeng Wang, Minghao Shao, Jitendra Bhandari, Likhitha Mankali, Ramesh Karri, Ozgur Sinanoglu, Muhammad Shafique, and Johann Knechtel. 2025. VeriContaminated: Assessing LLM-driven verilog coding for data contamination. arXiv:2503.13572. Retrieved from https://arxiv.org/abs/2503.13572

[214] Cody Watson, Nathan Cooper, David Nader-Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology* 31, 2 (2022), 1–58. DOI: https://doi.org/10.1145/3485275

[215] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1480–1496. DOI: https://doi.org/10.1109/TSE.2020.3023177

[216] 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 (ASE '16)*. ACM, New York, NY, 87–98. DOI: https://doi.org/10.1145/2970276.2970326

[217] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 334–345. DOI: https://doi.org/10.1109/MSR.2015.38

[218] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark, and Kristina Lundqvist. 2017. Impediments for software test automation: A systematic literature review. *Software Testing* 27 (2017). Retrieved from https://api.semanticscholar.org/CorpusID:32631031

[219] Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, et al. 2025. Research directions in software supply chain security. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[220] Jonas Paul Winkler, Jannis Grönberg, and Andreas Vogelsang. 2019. Predicting how to test requirements: An automated approach. In *Proceedings of the 2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 120–130. DOI: https://doi.org/10.1109/RE.2019.00023

[221] Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. 2023. DeceptPrompt: Exploiting LLM-driven code generation via adversarial natural language instructions. arXiv:2312.04730. Retrieved from https://arxiv.org/abs/2312.04730

[222] 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 (ICSE '23)*. IEEE Press, 1482–1494. DOI: https://doi.org/10.1109/ICSE48619.2023.00129

[223] Chunqiu Steven Xia and Lingming Zhang. 2023. *Conversational automated program repair*. arXiv:2301.13246. Retrieved from https://arxiv.org/abs/2301.13246

[224] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for 0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. ACM, New York, NY, 819–831. DOI: https://doi.org/10.1145/3650212.3680323

[225] Mingxuan Xiao, Yan Xiao, Shunhui Ji, Yunhe Li, Lei Xue, and Pengcheng Zhang. 2025. ABFS: Natural robustness testing for LLM-based NLP software. arXiv:2503.01319. Retrieved from https://arxiv.org/abs/2503.01319

[226] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/3520312.3534862

[227] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A systematic evaluation of large language models of code. arXiv:2202.13169. Retrieved from http://arxiv.org/abs/2202.13169

[228] 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 *Proceedings of the 13th International Conference on Learning Representations (ICLR)*.

[229] Tingting Xu, Yun Miao, Chunrong Fang, Hanwei Qian, Xia Feng, Zhenpeng Chen, Chong Wang, Jian Zhang, Weisong Sun, Zhenyu Chen, et al. 2024. A prompt learning framework for source code summarization. arXiv:2312.16066. Retrieved from https://arxiv.org/abs/2312.16066

[230] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F. Bissyandé, and Shunfu Jin. 2024. CREF: An LLM-based conversational software repair framework for programming tutors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. ACM, New York, NY, 882–894. DOI: https://doi.org/10.1145/3650212.3680328

[231] Fengyu Yang, Guangdong Zeng, Fa Zhong, Wei Zheng, and Peng Xiao. 2023. Interpretable software defect prediction incorporating multiple rules. In *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 940–947. DOI: https://doi.org/10.1109/SANER56733.2023.00114

[232] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. ACM, New York, NY, 1607–1619. DOI: https://doi.org/10.1145/3691620.3695529

[233] Wei Yang, Yuan Yang, Wei Xiang, Lei Yuan, Kan Yu, Álvaro Hernández Alonso, Jesús Ureña, and Zhibo Pang. 2024. Adaptive optimization federated learning enabled digital twins in industrial IoT. *Journal of Industrial Information Integration* 41, 2024 (2024), 100645. DOI: https://doi.org/10.1016/J.JII.2024.100645

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

[235] Sixiang Ye, Zeyu Sun, Guoqing Wang, Liwei Guo, Qingyuan Liang, Zheng Li, and Yong Liu. 2025. Prompt alchemy: Automatic prompt refinement for enhancing code generation. arXiv:2503.11085. Retrieved from https://arxiv.org/abs/2503.11085

[236] 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.

[237] 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.

[238] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving ChatGPT for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726. DOI: https://doi.org/10.1145/3660783

[239] Tao Yue, Wolfgang Mauerer, Shaukat Ali, and Davide Taibi. 2023. Challenges and opportunities in quantum software architecture. *Software Architecture: Research Roadmaps from the Community*, 1–23.

[240] Wojciech Zaremba and Greg Brockman. 2021. OpenAI Codex. Retrieved from https://openai.com/blog/openai-codex/

[241] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2022), 1–36. DOI: https://doi.org/10.1109/TSE.2019.2962027

[242] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. 2024. AutoCodeRover: Autonomous program improvement. arXiv:2405.02213. Retrieved from https://arxiv.org/abs/2405.02213

[243] Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. 2025. LLM App Store Analysis: A Vision and Roadmap. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue (April 2025).

[244] 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.

[245] Tao Lin, Yaowen Zheng, Jingquan Ge, Jun Wang, Jacques Klein, Tegawende Bissyande, Yang Liu, Zhihao Lin, Wei Ma, and Li Li. 2025. Open-source AI-based SE tools: Opportunities and challenges of collaborative software learning. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue.

[246] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transaction on Software Engineering and Methodology* 2030 Roadmap Special Issue. DOI: https://doi.org/10.1109/TSE.2018.2887384

[247] 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. DOI: https://doi.org/10.1109/TSE.2018.2887384

Received 13 April 2025; revised 13 April 2025; accepted 16 April 2025