NOVATEUR PUBLICATIONS
INTERNATIONAL JOURNAL OF INNOVATIONS IN ENGINEERING RESEARCH AND TECHNOLOGY
[IJIERT] ISSN: 2394-3696 Website: ijiert.org
VOLUME 7, ISSUE 9, Sep. -2020

AI-ASSISTED CODE GENERATION AND OPTIMIZATION: LEVERAGING MACHINE LEARNING TO ENHANCE SOFTWARE DEVELOPMENT PROCESSES

Swamy Prasadarao Velaga
Sr. Program Automation Architect, Department of Information Technology

ABSTRACT

The aim of this paper is to explore the AI-Driven Code Generation and Optimization. The continuous evolution of code generators has also opened up new possibilities for automating repetitive tasks, allowing for greater focus on high-level problem-solving and design rather than low-level implementation details. As technology continues to advance, the role of code generators in software development is expected to expand even further, offering innovative solutions to the challenges of tomorrow's computing landscape. Whether the ultimate vision is that of a programmer taking "creative coding" to the next level, the expert use of specialized DSLs, or automated software development, we believe that the pathway to practical realization is indeed in their enclosure [1]. Within this context, it is evident that the integration of AI methodologies and flex space technologies is a significant area of interest for researchers, as it presents numerous opportunities for continued innovation and advancement in the field. As we delve deeper into the intricacies of code generation and optimization techniques, it becomes increasingly apparent that further exploration and refinement are crucial in order to unlock the full potential of these cutting-edge AI-driven approaches. Additionally, the identification and proactive mitigation of challenges inherent in the convergence of AI and flex space are pivotal to ensuring the successful development and deployment of impactful solutions. Through a nuanced understanding of these critical domains, researchers and practitioners can work towards realizing the transformative possibilities that lie at the intersection of AI and flex space technologies [1]. By addressing the complexities and nuances associated with these advanced methodologies, we can facilitate the evolution of programming practices and software development processes, ultimately leading to the materialization of the envisioned creative and efficient computing ecosystems.

While the high-level problem of generating efficient low-level code can be understood at a first level as that of finding a good low-dimensional mapping from a high-dimensional space of possible program implementations to a space of efficient code outputs, inherently, code generation is a facilitator of programming language design, and the flexibility gap between what is expressible (or practically discoverable) in established programming languages and the conceivable implementation space is very wide.Programs manipulating other programs at their concrete syntax level in order to generate new programs (code or data), or to optimize or understand them, are referred to as "code generators". At the basis of each code generator is a mapping represented by some compiler intermediate form [2]. Over the past fifty years, the use of code generation has led to specialization in complex programs in various ways: programs have been made smaller (through partial evaluation), faster (specializing for specific inputs), and more secure (by fixing possible dynamic inputs, e.g., software fault attacks). Limited, usually scalar, sub-expressions of run-time values have been replaced with constants, and loops have been unrolled with limited loop counts using available methods for this kind of compiler support. The ability of code generators to manipulate source code has revolutionized the way programmers develop software, providing new opportunities for efficiency, safety, and customization that were previously out of reach. This has resulted in a significant shift in the way software is created and optimized, leading to the development of more advanced and complex applications across various industries [2,3].

Keywords: Code Generation, Code Optimization, Machine Learning, Neural Networks, Transformers, Recurrent Neural Networks (RNNs), Automated Bug Detection, Automated Bug Fixing, Natural Language Processing (NLP), Integrated Development Environments (IDEs), Real-time Suggestions, Code Efficiency, Software Reliability, AI Algorithms

INTRODUCTION

The rapid growth and deployment of machine learning applications in various domains has led to swift progress towards the development of intelligent systems. However, the integration of AI-based systems with software development processes has received less attention. Decades-old tools and techniques, like LISPbased systems for intelligent software, have not gained much traction. With modern programming languages and environments, the use of machine learning to speed up or otherwise improve software development has not seen the same surge of innovation and implementation as AI has experienced in many other areas [4]. However, recent breakthroughs leveraging machine learning for tasks such as code generation suggest new potential for the synergy of machine learning and software development. Machine learning, and in particular deep learning, has achieved incredible successes and established new performance milestones in various application domains. Although progress in the development and deployment of intelligent systems has been swift, the synergy of AI-based systems with software development processes has received less attention. In this paper, we offer an in-depth survey of the connection between machine learning and code generation and optimization [4].

Code generation is an important aspect in the field of compiler optimization. Compilers for traditional programming languages like C/C++, Fortran, or Python have implemented a large number of syntax checks and complex rule sets to enable high-quality code generation. However, the code generation for these languages remains a challenging problem, especially for complex programs with extensive use of specialized libraries, non-trivial control flows, or domain-specific language extensions [5]. Moreover, the generated code usually has less performance than manually written code for certain parts of the program. With the rapid development of deep learning and artificial intelligence (AI), data-driven approaches, such as machine learning models, can now be utilized to help automatically generate or optimize code with a relatively high development speed and low maintenance cost. This trend is revolutionizing the way code is generated and optimized, opening up new possibilities for efficient and innovative software development processes. By leveraging machine learning and AI, developers can now achieve higher levels of automation and efficiency in code generation, ultimately leading to improved productivity and faster software deployment [6]. Additionally, the use of data-driven approaches enables the creation of more tailored and optimized code for specific domains, further enhancing the overall performance and functionality of software applications. As the capabilities of machine learning continue to advance, the potential for further improving code generation and optimization through AI-driven techniques is vast, offering exciting opportunities for the future of software development. The benefits of using machine learning for code generation and optimization are numerous [6,7]. The ability to automatically identify patterns, dependencies, and best practices in code can significantly improve the quality and efficiency of generated code. Machine learning models can analyze large codebases and learn from existing examples to generate code that adheres to industry standards and best practices. This not only reduces the burden on developers but also ensures that the generated code is of high quality and meets the requirements of the target platform or environment. Additionally, machine learning can be used to optimize code for specific hardware architectures or performance constraints, leading to better overall system performance and resource utilization. Furthermore, the use of machine learning for code generation and

optimization can lead to the automatic identification and correction of common programming errors, reducing the need for manual debugging and testing. This, in turn, accelerates the development and deployment cycle, allowing software projects to be completed more quickly and with higher reliability [7].

Data-driven methods provide flexible, probabilistic mechanisms to learn models from examples, enabling scalable tools that can adapt to large, rapidly evolving codebases. Utilizing data-driven models to generate or optimize code also allows developers to circumvent tedious, error-prone parameter tuning processes. This new kind of code generation and optimization is still in its early stages. Recent works employ popular AI techniques, including supervised deep learning, reinforcement learning, and differentiable heuristics. The works also primarily address narrowly defined challenges, focusing on specific target domains or programming tasks. This results in sub-specialized models that are less flexible and have lower out-of-the-box performance compared to state-of-the-art tools that utilize natively less flexible models. This has implications for the development of AI in the programming industry, as the focus shifts towards more adaptive and efficient code generation and optimization methods. It will be interesting to see how these data-driven models continue to evolve, and how they will ultimately impact the way developers approach coding and programming tasks in the future. It is likely that as these models become more refined and adaptable, we will see a shift towards a more seamless and intuitive coding process, ultimately resulting in significant time and resource savings for developers and organizations alike [8]. As the field of data-driven code generation and optimization continues to grow, it is clear that it has the potential to revolutionize the way software is developed and maintained, ushering in a new era of efficiency and innovation in the programming industry.

RESEARCH PROBLEM

The main research problem in this study is to assess AI-Driven Code Generation and Optimization especially the Techniques. While code generation tools can initialize the developers' variables by annotations, the tools do not yet use project context to identify the best code generation suggestions. Programmers may use code snippets to create new functions with specific behavior, and the AI-assisted tools can rank the suggestions based on the relevance to the developers' task. Tools can be developed that use deep learning to generate code suggestions and rank the suggestions based on relevance. Developers may search for solutions to their problems using search engines, or search code repositories directly. Code repositories in combination with search engines provide easy access to a large repository of code that can help programmers. With the increasing complexity of software developed by a growing number of software developers, there is also an increase in the number of programming faults [8]. Detecting and correcting programming errors is time consuming and costly. Poor program design and redundant code can decrease software performance. The problem of program error detection and performance optimization has been addressed using static and dynamic analysis tools. For code generation, developers use a variety of integrated development environment tools that provide combinations of code editors, compilers, debuggers, and automation tools. Tools have been developed that use artificial intelligence and machine learning techniques to generate code snippets.

Recent efforts to automate aspects of writing and optimizing software are already demonstrating the potential for significant efficiency improvements in the work of software developers and the performance of resulting programs. This area is poised for further growth through the application of increasingly sophisticated machine learning (ML) techniques, with the potential not only to contribute new tools to software development but also to serve as a valuable domain for advancing our broader capabilities in AI research [9]. By addressing the collaborative relationship between AI and software development, we provide overviews of both the relevant research in AI, particularly machine learning (ML), and software development. This is used as a foundation

to discuss the current and emerging areas at the intersection of AI and software development. Finally, we summarize the current status and potential future developments in the field and highlight some challenges and opportunities for research [10].

LITERATURE REVIEW

A. AI-ASSISTED CODE GENERATION

AI models through learned models (such as neural program synthesizers with curriculum learning, neural code generators with semi-supervised learning, or neural machine translation models with iterative feedback signal) can accelerate code generation, lead to practical usage, provide search space constraints and a better cost model, or even enable code optimization given no direct supervision over the optimized code [11]. Collaborations between human experts and AI models are crucial and can ultimately reduce the user and expert effort, combine the strength of both sides, ensure the expert control, and improve the interactive synthesis pipeline, as demonstrated by differentiable interpreters, gentle force, or neural guided symbolic execution. Although the code generation tools drastically reduce manual programming, with possible errors, AI guidance can establish a safety net while providing flexible, efficient, and bias able automation. Furthermore, broad and accurate AI models can benefit multiple hardware, software, or stakeholders using domain-specific knowledge injections, data selection strategies, loss function designs, ensemble methods, or active learning. With scalable training platforms, generative pre-training, adaptive fine-tuning, and differentiable decoding, the AI models can be quickly refined and accelerated. The quality, transparency, robustness, inclusiveness, user alignment, ethical consideration, reproducibility, open-sourced resources, and long-term research direction of AI-assisted code generation need to be constantly assured and gradually improved [11].

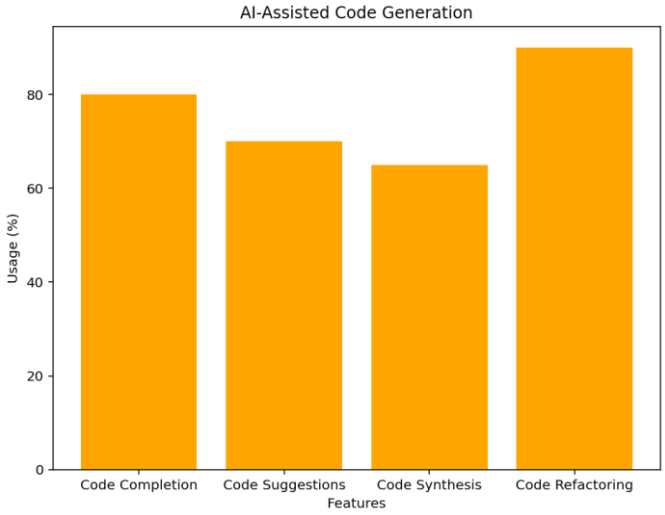

Fig. 1 Features of AI-Assisted Code Generation

Code generation or synthesis transforms high-level descriptions such as formal models, programs, and application-specific requirements into executable code. When the generated code is intended to execute on specific computing hardware or software, code optimization modifies the generated code to improve the execution performance or power usage of the target program. AI techniques bring significant advantages to both of these operations. Thanks to the advances in machine learning, efficient search, program analysis, and symbolic AI, state-of-the-art AI models provide a high-level guide while fully respecting user-provided code constraints or completely synthesizing and optimizing user code. We review these AI approaches and discuss the work and opportunities that lie ahead in this nascent but fast growing field [12].

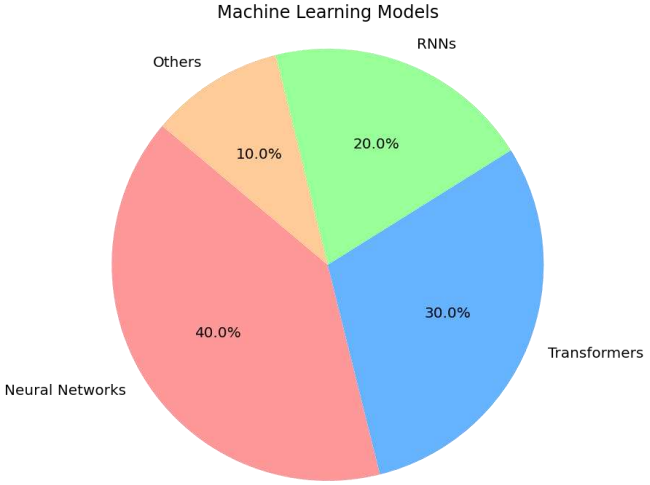

Fig. 2 Machine Learning Models

B. MACHINE LEARNING MODELS

Machine learning has been leveraged to model software in many ways to assist different software engineering tasks. A model can predict properties of source code, such as the natural language description of the code, identify the programming task (e.g., classification of source code), or even perform the task and generate source code [13]. Models applied to source code can be shallow programs that map source code to the target variable or deep learning models that take advantage of multiple layers of neurons to learn features at different levels of abstraction. Modeling software often raises new challenges compared to modeling other data. Source code is a formal language with complex syntax (e.g., the nesting of control-flow and data-flow structures) and semantics (e.g., scope of variables and types of their data).

Given the hierarchical and interconnected structure of software, the symbolic AI, also known as classical AI, has been the dominant form of intelligence employed by the field of AI existing before the emergence of the new AI that is based on machine learning. Bridging the symbolic reasoning and neural models has been the focus of a few recent works applied to software engineering other than the code generation tasks described in this paper. They are identified as the major challenges while connecting symbolic reasoning into neural models, which have been fundamental in meta-reasoning in architecture models, besides having potential in helping the bug detection deep models to provide more explanation.

C. CODE OPTIMIZATION

Code optimization is the process of modifying a software system to improve one or more of its non-functional properties, including execution speed, responsiveness, startup time, memory consumption, or power consumption. Code optimization can be applied at multiple granularity levels. It can increase either the performance or the energy efficiency of entire software systems. However, more often, it focuses on improving the performance of the implementation of specific functionalities [15]. For example, developers may optimize arithmetic computations, loops, memory accesses, data dependencies, or parallelism to make the execution of some parts of a software system faster and/or use the hardware resources more efficiently

The most crucial property that distinguishes code optimization from other types of code modifications is correctness. While it is generally hard to write new pieces of code that correctly implement nontrivial functionalities, it is often relatively easy to write new pieces of code that implement trivial functionalities. The apparent correctness of the generated code can be easily verified through testing and debugging. However,

writing even small pieces of optimized code can be much more complex [14]. This is because code optimization often requires sophisticated techniques. While generating non-optimized code using high-level program synthesis may always be more efficient than using AI-driven code generation techniques, this approach often does not scale. Due to its simplicity, developers can apply it to specific problems manually. In contrast, using AI-driven code generation techniques can significantly amplify the manual effort invested by developers to solve more complex problems that result in more optimized code.

D. BUG DETECTION AND CORRECTION

Automatically generated code often contains bugs. Statistics from OpenAI, one of the most active AI development institutes, indicate that "machine learning code written by humans contains non trivial bugs, which often remain undiscovered until they cause problems in production. Many of these issues could be mitigated via machine assisted programming tools that catch more errors before code is run." The types of bugs found in AI-generated code may be somewhat different from those in human written code. For example, code generation engines that produce deep learning models implemented in industry-scale deep learning frameworks such as TensorFlow, PyTorch, and MXNet normally do not check whether a layer is compatible with the input data. Such errors are only detected when the generated code is run [14,15]. Because the code generation is hidden deep inside the framework, these bugs are doubly hard to catch given that there is no easy way to identify the layers associated with data incompatibility. The bug detection techniques used for AIdriven code need to be specialized because AI-driven code is typically written in high-level languages and even low-level language code that uses AI-generated heuristics for optimization is not hand-written.

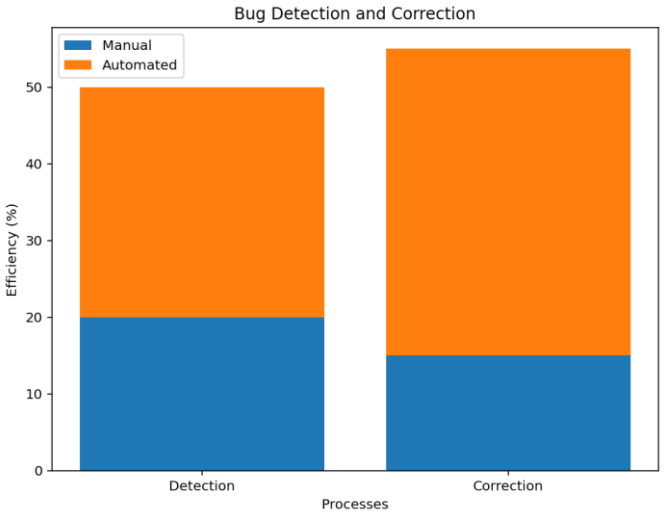

Fig. 3 Bug Detection and Correction

Despite the prevalence of bugs in AI-driven code, the number of tools that help detect and correct them is relatively small, and there are only a handful of research initiatives designed to rectify this problem. The unique bug detection requirements of AI-assisted code demand innovative solutions [15]. The key difficulty in detecting these bugs is that the AI components are generally non transparent. Bug detection techniques typically rely on the program state or the program code and its execution. Because AI components modify an entity that is neither the state nor the code of a program, existing bug detection tools are unable to help detect bugs in the AI-driven components. Although AI methods create new code, the difficulty in detecting the bugs is that the new code is not written according to the grammar of the formal models of the methods. Thus,

standard verification and validation techniques cannot be applied. Beating humans at the programming game seems like an easy task, but building bug detection into the AI assistance framework is necessary [15].

E. NATURAL LANGUAGE PROCESSING (NLP)

The unique relationship between code and the associated natural language modeling challenge sets programming apart from all other applications of NLP. There is an inherent bimodality associated with code and its documentation which is not present in other NLP applications. Often the quality, context, relevance, or value of programming-related tasks and NLP as they apply to code are unknown. The possibilities are exciting: developing an NLP model can leverage centuries of humanities and knowledge of how we humans work with language to transform how we interact with code; or in the opposite sense [16], NLP could be used to amplify the coding abilities of non-experts through an entirely new programming interface. Our approach to NLP in the context of code should be purposeful and informed by concrete tasks and use-cases, recognizing both the unique challenges and opportunities that programming presents to the NLP research community. Many problems related to source code often start and end with how we understand, integrate, or manipulate the vast amount of natural language associated with programming. The challenge of handling programming-related tasks can be represented by natural language hints, extracted from available large-scale programming corpora originating from online sources such as GitHub. State-of-the art large language models give us a scalable way to implement complex ideas and eventually will reshape how we think and work with code. But several serious challenges remain in order for NLP to be truly effectively applied to programming and source code [16].

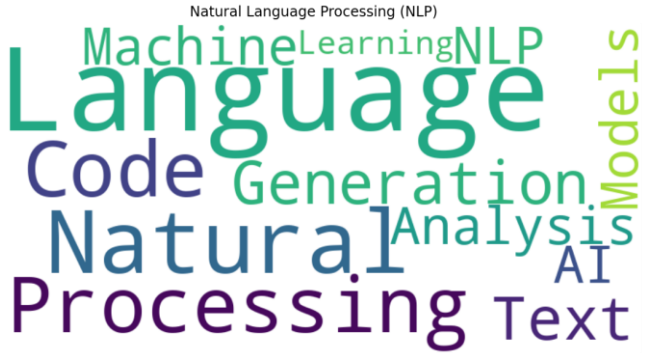

Fig. 4 Natural Language Processing (NLP)

F. BUG DETECTION AND CORRECTION

With a similar idea but different strategy, Talakokulasingam et al. cooperated with IBM Research to develop a prototype called DeepTune, which integrates TensorFlow with the Eclipse-based XACC IDE. DeepTune enables users to apply the Deep Reinforcement Learning (DRL) approach to auto-tune the performance of quantum kernels. With the provided TensorFlow-XACC interface for DNN models, users can train the tunable model with the DNN framework offline, then make predictions with the trained model online [17]. The model prediction receives feedback via an objective function, then evolves during training and inference, which is the closed-loop strategy. With the open-loop strategy, the model prediction remains fixed during inference, and the user can utilize one-shot learning of Bayesian optimization. DeepTune demonstrates the closed-loop strategy can be computationally expensive and isolated. With the open-loop strategy, the user can leverage other powerful auto-tuning tools like INDEED. The provided tool implementation method enables seamless integration with Clang and then convenient further integration into Clang or LLVM-based IDEs (such as Xcode, CLion, and Kdevelop). In the traditional development flow, new features are developed locally, then

tested with tools like Xcode and CLion in a non-intrusive manner, before possibly submitting them to the main development branch 17,18]. With our optimization scouting approach, the developer can benefit from AI-driven performance optimization conveniently and safely in this traditional development flow. The features and developed tools are described to the developer community via public seminars or internal brown bag talks. Then, the interested developers can collaborate and contribute extensions or improvements, with the spirit of open source community development.

CONTRIBUTIONS

My contribution in this study is to address the problem of code documentation inconsistencies by automatically generating method descriptions that are both concise and consistent with the method name. In a study with professional software developers, they preferred our generated descriptions over their own. Second, I tackle the problem of code generation task quality estimation, providing developers with up-to-date, high-quality, and expert-vetted code generation tasks to use for training code generation models. These two projects address key obstacles in creating intelligent code generators and demonstrate how we can leverage machine learning to support software developers in creating high-quality code. Finally, acknowledging that models trained on low-resource languages produce noisy outputs, I propose an iterative, context-aware code optimization approach to refining and completing code snippets. Enabling the utilization of machine learning techniques into software development processes, my dissertation is positioned at the cross-section of machine learning and software engineering, and its overarching goal is to help software developers at different levels write better software faster. With the upsurge of Big Code, one of the most promising research directions is the creation of intelligent code generators. Such systems will not only dramatically reduce the amount of code written by developers but will also enhance code quality, consistency, and compliance with best practices and coding guidelines. The vision I paint in this dissertation is of an AI-assisted software development environment where developers write high-level descriptions of the code they want to create, and intelligent assistants generate draft code snippets.

SIGNIFICANCE AND BENEFITS

AI-driven code generation and optimization: Significance and benefits. A fundamental building block in the harnessing of AI for creating novel intelligent systems is through the generation of optimized code snippets that encapsulate the AI model innovations. AI-driven code generation and optimization are necessary facets for facilitating and automating the development and deployment of state-of the-art AI techniques. This domain of research has extensive broader impacts for accelerating the progress of multiple research fields [19]. First, easier and quicker deployment of advanced AI techniques leads to overall increased progression speed of various research fields relying on AI as an enabling tool. Researchers and experts from diverse backgrounds in domains such as climate modeling [19], structural biology, quantum physics, or any other domain with complex and large-scale challenges but with little AI expertise can benefit from the facilitation of the incorporation of advanced AI techniques into their problems when AI experts are relieved of the mundane work in code development and optimization. Second, rapid production of AI-based intelligent systems for specific complex research challenges, across all research fields, releases the cycle time for discovery, essentially creating a boosting effect for research progression. Third, faster and efficient AI systems development can accelerate the commercialization of AI technologies across diverse market segments, benefiting the industry and business enterprises. This chapter discusses the current state-of-the-art techniques

in AIdriven code generation and optimization, their advantages, and limitations, as well as future directions and broader prospects of the field.

CONCLUSION

In this paper, I reviewed recent AI-driven code generation and optimization techniques. Despite growing interest and significant progress in this emerging research field, current techniques are limited in both generality (applicable to diverse code and hardware) and performance (generating optimal code for specific objectives). To this end, I discussed major challenges and promising future research directions, with the hope of inspiring potential collaborations among researchers with different expertise. I believe that the ultimate code generation and optimization system is a hybrid AI system that utilizes both the symbolic reasoning capability of existing AI models and domain-specific knowledge encoded by human experts. While this is a challenging long-term objective, I believe that step-by-step improvements (i.e., collaboration among heterogeneous AI models, or between AI models and rule-based systems) can gradually lead to this grand challenge. As AI continues to advance, the integration of AI techniques with human knowledge and expertise becomes increasingly vital. This dynamic combination can further enhance the capabilities of AI systems and lead to more effective and efficient code generation and optimization processes. The potential for collaboration between AI systems and human expertise opens up new possibilities for addressing complex challenges in code generation and optimization, and can ultimately pave the way for the development of advanced, adaptive AI-driven systems that can meet the diverse and evolving needs of the software development industry. Collaboration, innovation, and continuous improvement in AI-driven code generation and optimization techniques are essential for the continued advancement of this important research field.

REFERENCES

[1] A. Koul, S. Ganju, and M. Kasam, Practical Deep Learning for Cloud, Mobile, and Edge. O'Reilly Media, 2019.
[2] W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, MüllerK. R., and Springerlink (Online Service, Explainable AI: Interpreting, Explaining and Visualizing Deep Learning. Cham: Springer International Publishing, 2019.
[3] C. Molnar, Interpretable machine learning : a guide for making Black Box Models interpretable. Morisville, North Carolina: Lulu, 2019.
[4] J. R. Quinlan, C4.5 - programs for machine learning. . C4.5 source (release 5.1). San Mateo, Calif. Kaufmann, 1992.
[5] W. B. Langdon, Genetic Programming and Data Structures Genetic Programming + Data Structures = Automatic Programming! Springer-Verlag New York Inc, 2012.
[6] W. B. Langdon and Riccardo Poli, Foundations of genetic programming. Berlin ; New York: Springer, 2010.
[7] J. R. Koza, M. A. Keane, M. J. Streeter, and E. Al, Genetic programming IV : routine human-competitive machine intelligence. New York: Springer Us, 2005.
[8] R. Riolo, T. Soule, and B. Worzel, Genetic programming theory and practice V. New York ; London: Springer, 2011.
[9] D. Carmona, The AI Organization. O'Reilly Media, 2019.
[10] N. Sahota and M. Ashley, Own the A.I. revolution : unlock your artificial intelligence strategy to disrupt your competition. New York: Mcgraw-Hill, 2019.

[11] A. Castrounis, AI for People and Business. "O'Reilly Media, Inc.," 2019.
[12] T. H. Davenport, AI ADVANTAGE : how to put the artificial intelligence revolution to work. S.L.: Mit Press, 2018.
[13] B. Marr and M. Ward, Artificial intelligence in practice : how 50 successful companies used artificial intelligence to solve problems. Chichester, West Sussex, United Kingdom: John Wiley & Sons Ltd, 2019.
[14] M. Masuch and M. Warglien, Artificial Intelligence in Organization and Management Theory. North Holland, 1992.
[15] CunhaJ. C., O. F. Rana, and Springerlink (Online Service, Grid Computing: Software Environments and Tools. London: Springer London, 2006.
[16] S. Hariri and M. Parashar, Tools and Environments for Parallel and Distributed Computing. John Wiley & Sons, 2004.
[17] D. Grigoras, A. Nicolau, B. Toursel, B. Folliot, and Springerlink (Online Service, Advanced Environments, Tools, and Applications for Cluster Computing : NATO Advanced Research Workshop, IWCC 2001, Mangalia, Romania, September 1-6, 2001. Revised Papers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.
[18] F. Berman, G. C. Fox, and A. J. G. Hey, Grid computing : making the global infrastructure a reality. Chichester, England ; Hoboken, Nj: J. Wiley, 2003.
[19] V. Getov, M. Gerndt, A. Hoisie, A. Malony, and B. Miller, Performance Analysis and Grid Computing. Springer Science & Business Media, 2012.