COMPUTER RESEARCH AND MODELING
2024 VOL. 16 NO. 7 P. 1715–1726
DOI: 10.20537/2076-7633-2024-16-7-1715-1726

Ки&М

SPECIAL ISSUE

UDC: 004.42

A survey on the application of large language models in software engineering

N. Salem[1,a], A. Hudaib[1,b], K. Al-Tarawneh[1,c], H. Salem[2,d], A. Tareef[1,e], H. Salloum[2,f], M. Mazzara[2,g]

[1]King Abdullah II School for Information Technology, University of Jordan,
Amman, Jordan
[2]Innopolis University,
1 Universitetskaya st., Innopolis, 420500, Russia

E-mail: [a] NAD9220478@ju.edu.jo, [b] ahudaib@ju.edu.jo, [c] Khawla_t@mutah.edu.jo, [d] H.salem@innopolis.ru,
[e] a.tareef@ju.edu.jo, [f] h.salloum@innopolis.university, [g] m.mazzara@innopolis.ru

Received 26.10.2024, after completion — 19.11.2024
Accepted for publication 25.11.2024

Large Language Models (LLMs) are transforming software engineering by bridging the gap between natural language and programming languages. These models have revolutionized communication within development teams and the Software Development Life Cycle (SDLC) by enabling developers to interact with code using natural language, thereby improving workflow efficiency. This survey examines the impact of LLMs across various stages of the SDLC, including requirement gathering, system design, coding, debugging, testing, and documentation. LLMs have proven to be particularly useful in automating repetitive tasks such as code generation, refactoring, and bug detection, thus reducing manual effort and accelerating the development process. The integration of LLMs into the development process offers several advantages, including the automation of error correction, enhanced collaboration, and the ability to generate high-quality, functional code based on natural language input. Additionally, LLMs assist developers in understanding and implementing complex software requirements and design patterns. This paper also discusses the evolution of LLMs from simple code completion tools to sophisticated models capable of performing high-level software engineering tasks. However, despite their benefits, there are challenges associated with LLM adoption, such as issues related to model accuracy, interpretability, and potential biases. These limitations must be addressed to ensure the reliable deployment of LLMs in production environments. The paper concludes by identifying key areas for future research, including improving the adaptability of LLMs to specific software domains, enhancing their contextual understanding, and refining their capabilities to generate semantically accurate and efficient code. This survey provides valuable insights into the evolving role of LLMs in software engineering, offering a foundation for further exploration and practical implementation.

Keywords: large language model, natural language processing, software development life cycle

Citation: *Computer Research and Modeling*, 2024, vol. 16, no. 7, pp. 1715–1726.

Introduction

Human beings possess a remarkable ability to utilize language as a means of self-expression and interpersonal connection. This capacity begins to develop in early childhood and continues to evolve throughout an individual's life. In contrast, machines lack this inherent capability for understanding and communication, unless they are equipped with advanced artificial intelligence (AI) algorithms. The quest to enable machines to read, write, and communicate in a manner akin to human beings has been a significant challenge and aspiration within the field of AI [Chowdhary, Chowdhary, 2020; Hadi et al., 2023]. Artificial intelligence encompasses the creation of systems that can replicate human cognitive abilities [Zhao et al., 2023]. In the 18th century, the philosopher Denis Diderot posited a thought-provoking idea: if a parrot could respond to any inquiry, it could be deemed intelligent [Liu et al., 2023a]. Although Diderot referred to living beings, his concept highlighted the intriguing notion that a highly intelligent entity might exhibit human-like behavior. This idea was further expanded by Alan Turing in the 1950s, who proposed what is now known as the Turing Test. This test remains a pivotal concept in AI, focusing on determining whether machines can exhibit intelligence comparable to that of humans [Li et al., 2023]. In AI discourse, the term "agent" frequently arises. Agents serve as the foundational components of AI systems. An agent can perceive its environment through sensors, make decisions, and take actions using actuators [Weng, 2023].

The investigation at hand reviews recent advancements in LLMs and their profound implications for the AI community. It delves into various aspects of LLMs, including pre-training, adaptation, utilization, and evaluation of capabilities, while emphasizing their significance and potential future directions. This review highlights the powerful capabilities of pre-trained models, particularly in addressing diverse natural language processing (NLP) tasks. Moreover, it articulates the central theme of model scaling, which enhances LLM performance and unveils specialized capabilities not evident in smaller models. The technological evolution in the LLM domain has revolutionized AI algorithm development, leading to significant advancements that have garnered substantial attention in both academia and industry. As progress continues, the discussion will address persisting challenges and future research directions, delineating the imperative for ongoing exploration and innovation in LLM studies. In Fig. 1 an evolution process of the four generations of language models from the perspective of task-solving capacity.

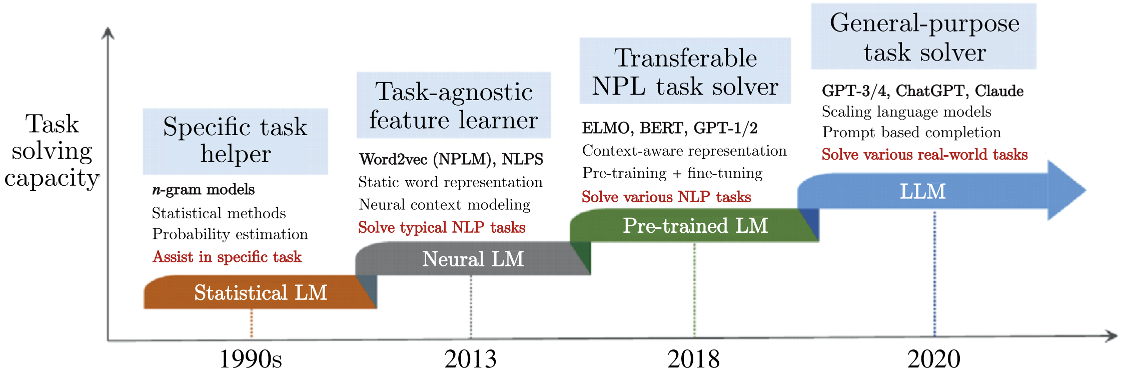

Figure 1. Evolution process of the four generations of language models (LM) from the perspective of task-solving capacity [Zhao et al., 2023]

The exploration of LLMs in NLP tasks reveals transformative benefits across multiple domains. With extensive prior training on diverse datasets, LLMs demonstrate formidable capabilities in understanding and generating human language, thereby excelling in language understanding and generation tasks. Specifically, the significantly expanded LLM, which owes its success to its large model size, is optimized for linguistic tasks due to its extensive parameter volume. This enables it to accurately capture and reproduce nuanced linguistic features across tasks such as language translation, sentiment analysis, question answering, and text generation. Furthermore, applications such as login software, virtual assistants, and translation systems have been revolutionized by LLMs, promising more natural and engaging user experiences.

The scale of LLM impacts heralds a new era in AI algorithm development, identifying key areas beyond traditional NLP tasks that warrant innovation and progress. Notable applications, including language translation, sentiment analysis, and question answering, exemplify the significant contributions of larger LLMs across diverse fields. Nevertheless, substantial research remains to address the challenges that persist within the LLM domain as we strive towards new frontiers of discovery.

Large language models have emerged as pivotal components in various applications. In language translation, they dramatically enhance the performance of translation software by accurately capturing subtle linguistic features and generating distinctive translations. Similarly, in sentiment analysis, larger language models exhibit a profound understanding of the emotional tone within texts, leading to improved results in sentiment analysis tasks. Additionally, in question-answering scenarios, LLMs demonstrate enhanced comprehension of nuanced inquiries, thereby increasing overall accuracy. Their capacity to generate contextually relevant text has facilitated advancements in applications such as chatbots, virtual assistants, and content creation, representing a significant leap in the functionality of AI in natural and industrial applications.

AI-driven chatbots, such as ChatGPT, are poised to revolutionize healthcare by improving patient outcomes through enhanced communication between patients and healthcare professionals. Utilizing natural language processing, these chatbots facilitate more accessible dissemination of information regarding patient care and treatment options. For instance, there is an ongoing proposal for a database focused on repurposing COVID-19 drugs through NLP techniques [Thawkar et al., 2023]. Figure 2 shows the applications of LLMs in research directions, options, and downstream fields [32].

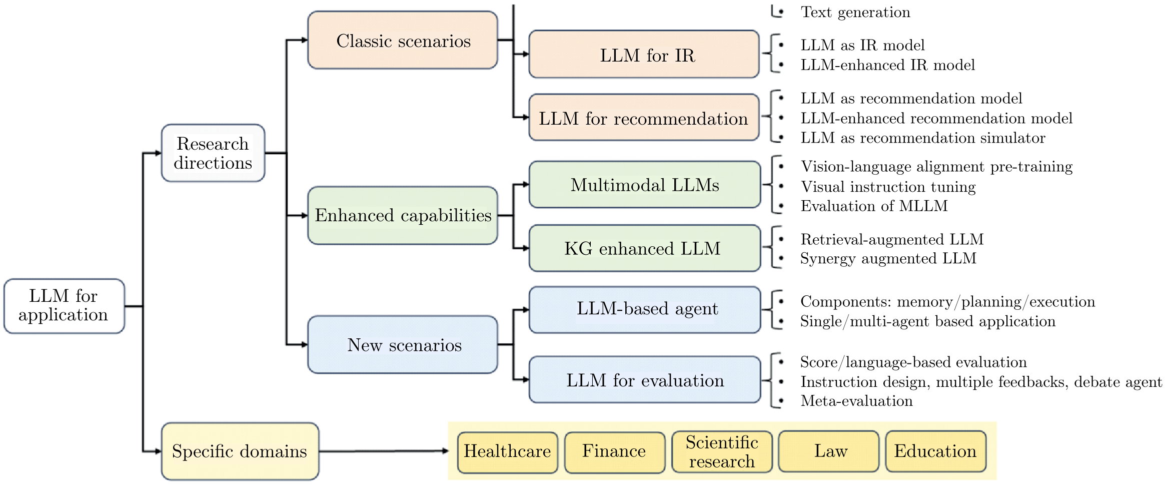

Figure 2. Applications of LLMs in research directions, options, and downstream fields [Zhao et al., 2023]

In summary, larger language models possess a distinct advantage in generating coherent and contextually appropriate text. This proficiency renders them invaluable for various applications, including chatbots, virtual assistants, and content creation. Their capability to capture intricate linguistic details enhances their performance in natural language tasks, ultimately facilitating a deeper understanding of language dynamics. As they continue to process vast datasets, these models improve their contextual understanding, allowing for the generation of creative and engaging content across multiple domains.

The methodology

The methodology employed involved a thorough investigation of 31 pertinent Software Engineering (SE) papers that utilized LLMs. An automated search strategy incorporating keywords related to both SE and LLMs was implemented. Zhang [Zhang et al., 2023b] investigates the utilization of LLMs within Software Engineering, exploring their practical applications throughout the development process. This section categorizes existing studies in SE that employ LLMs into four main phases of the SDLC: software requirements and design, software development, software testing, and software maintenance. Each phase encompasses various code-related tasks, such as fault localization and program repair.

Software requirements are comprehensive descriptions of conditions or capabilities necessary for users or system components, typically documented to ensure that developed software meets user expectations. Software design involves defining the structure, components, and functionalities of a software system. In the software requirements generation phase, automated processes derive formal descriptions from unstructured data sources like code comments. LLMs exhibit significant promise in automating this process, as demonstrated by Xie et al. [Xie et al., 2023a], who employed few-shot learning techniques for generalization. This study highlights recommendations for future research aimed at enhancing LLM efficacy in specification generation, including the exploration of hybrid methodologies that integrate traditional techniques with LLMs. In software specification repair, LLMs like ChatGPT are employed to address errors in formal declarations of software requirements or behaviors. Hasan et al. [Hasan et al., 2023] evaluated ChatGPT's capability in repairing specifications in the Alloy language, comparing its performance against existing automated repair methods. The study revealed ChatGPT's effectiveness in addressing unique errors while identifying areas for improvement in consistency. Effectively categorizing software requirements is crucial and encompasses both functional and non-functional dimensions. Functional requirements delineate specific tasks the software should perform, whereas non-functional requirements encompass broader attributes such as performance and security. Hey et al. [Hey et al., 2020] introduced NoRBERT, a BERT-based approach that effectively classifies both types of requirements through transfer learning. By labeling the functional section of the NFR dataset into categories such as Function, Data, and Behavior, NoRBERT achieved impressive F1-scores of up to 92 %. Khan et al. [Khan et al., 2023] explored LLMs, particularly XLNet, for identifying and categorizing non-functional requirements, yielding promising results in enhancing stability and user satisfaction in mobile banking applications. Overall, the findings indicate that transfer learning models are highly effective in precisely identifying and categorizing non-functional requirements within software development projects.

Subahi et al. [Subahi, 2023] proposed utilizing BERT to map non-functional requirements, addressing sustainability issues in software engineering. Their methodology connects requirements with aspects of green software sustainability, leveraging BERT's pre-training alongside domain-specific enhancements during the software development stage. GUI layouts involve the strategic arrangement and organization of elements such as widgets, images, and icons within a graphical user interface. The objective is to establish a user-friendly interface by considering spatial relationships, usability, and aesthetics. Brie et al. [Brie et al., 2023] investigated whether an LLM-based system could enhance the design process of GUI layouts without introducing irrelevant results. Instigator, an LLM-based system, was evaluated in a controlled study involving 34 participants. The system parses code from over 100 000 websites to extract GUI elements, creating a dataset of over 500 000 elements. The results indicated that Instigator effectively returned relevant GUI layouts based on user instructions while also revealing positive user perceptions regarding usability, trustworthiness, and efficiency. Despite some limitations in natural language understanding, Instigator offers significant benefits for practitioners in generating quality GUI layouts efficiently.

Software development involves utilizing computer programming languages, methodologies, and tools to manifest user requirements and functional necessities within computer programs. This section explores the progressions in code generation facilitated by LLMs, including notable examples like AlphaCode and CodeGen, which are instrumental in software development. It discusses three primary categories of advancements in LLMs for code generation:

Requirement-guided code generation involves LLMs generating code snippets based on natural language requirements. For instance, Li et al. [Li et al., 2023] introduced AceCoder, which employs requirement-guided generation combined with example retrieval to enhance code comprehension and implementation. Similarly, ClarifyGPT [Mu et al., 2023] and TurduckenGen [Yang et al., 2023] address user requirement ambiguity and syntactic constraints in code generation, respectively.

Inspired by human programming, this approach involves iteratively refining code based on execution results. Zhang et al. [Zhang et al., 2023a] present a technique named Self-Edit, aiming at improving code quality for competitive programming tasks by leveraging execution outcomes of code generated by LLMs. The core of their approach is a fault-aware code editor capable of editing and refining the generated code. Extensive evaluations demonstrate the substantial efficacy of their method, which enhances code quality by integrating execution results into the code and employing specialized code editors.

Additionally, Dong et al. [Dong et al., 2024] introduce a self-collaboration framework aimed at improving the problem-solving abilities of LLMs through collaborative and interactive methods. They explore how ChatGPT can facilitate collaborative code generation within software development workflows. Using their proposed framework, they create a basic team comprising three different roles of ChatGPT, designed to collaborate on code generation tasks. Through extensive experiments, they demonstrate the effectiveness and adaptability of their self-collaboration framework. They argue that enabling models to form teams and work together on complex tasks is a significant advancement towards automating software development.

Developers perform empirical evaluations to assess LLMs' attention alignment with human programmers and their efficacy in code generation tasks. Kou et al. [Kou et al., 2023] conduct an empirical investigation on the alignment of attention between LLMs and human programmers. This study involves creating a programmer attention dataset by manually annotating crucial words and phrases in programming tasks from HumanEval. Twelve attention calculation methods across three categories were implemented to capture attention. The alignment between model and programmer attention was scrutinized through quantitative comparisons and a user study involving 22 real programmers. Findings reveal consistent misalignment between LLMs' and programmers' attention. SHAP (SHapley Additive exPlanations), a perturbation-based method, exhibits the best alignment and is preferred by participants. The study concludes with implications and future research prospects for enhancing interpretation and performance of LLM-based code generation models. Nonetheless, challenges such as non-determinism in code generation tasks, particularly in semantic consistency, are noted, underscoring potential issues in ensuring code correctness and consistency. Searching for code is like exploring a vast library filled with books on software development. It's a crucial task where developers hunt down specific sections of code within extensive collections. This helps them find particular functionalities, understand how features are implemented, and even discover examples they can adapt for their projects. Think of it as using a combination of methods, tools, and techniques to navigate through this library efficiently. The goal is to locate, retrieve, and make use of code snippets effectively within software projects.

When it comes to existing LLMs designed for code search, they generally fall into three main categories [Xie et al., 2023b].

Improving the semantics of the query text

This involves refining how we understand and express the query. It's about enhancing the clarity and depth of meaning in the text, ensuring it accurately captures the intended message. This process involves extracting semantic information from the text, tailored to the context and more closely aligned with the functions or concepts embodied in the code.

In 2022, Li et al. [Li et al., 2022a] presented Code Retriever, a system designed to improve code search capabilities. By training on a large dataset of code-text pairs, Code Retriever learns semantic representations of code functions. It transforms inputs into vectors, refining them through training with contrastive losses. Code Retriever performs well in diverse code search tasks across multiple programming languages, exhibiting high performance even under resource constraints.

Introducing more efficient training techniques

This requires a holistic approach, encompassing both static and dynamic characteristics, along with evaluating model performance across different programming languages. The aim is to enhance the performance of LLMs in source code search tasks by introducing training methods that delve into proficient encoding representation learning.

Salza et al. [Salza et al., 2022] revealed that transfer learning significantly enhances neural network performance in code search, especially with limited training data. Leveraging the abundance of code on platforms like GitHub, transfer learning can benefit various source code analysis tasks. While models like BERT, designed for NLP, can be applied to code problems, they require extensive pre-training and fine-tuning. In cases with small training sets, a LUCENE model can perform similarly or better in code search. However, combining LUCENE with a pre-trained model on filtered search candidates yields the best results. The study also noted that BERT may struggle with short sequences common in code search queries, suggesting exploration of structured code features like ASTs. Optimizing preprocessing and exploring cross-lingual pre-training could further enhance model performance. Finally, understanding BERT's attention mechanisms in processing code may lead to improved model architectures for code search and related tasks.

Empirical investigations into the practical effectiveness of LLMs

Although code translation has been extensively studied in natural languages, it has received relatively little attention in the context of programming languages. In programming languages, code translation entails leveraging data from one language to improve a model's performance in another language. Transfer learning with LLMs has shown promising outcomes in enhancing various software engineering domains, including code summarization.

LLMs greatly enhance developer productivity in high-resource programming languages by utilizing both labeled and unlabeled code samples. However, many programming languages lack sufficient resources, hindering the benefits of LLMs for users of such languages. Cross-lingual transfer, though extensively studied for natural languages, remains underexplored for programming languages.

Baltaji et al. [Baltaji et al., 2023] present experiments with a transformer-based LLM across multiple programming languages to address questions regarding cross-lingual transfer effectiveness, source language selection, and predictive characteristics of language pairs. The study reveals practical insights, such as Kotlin and JavaScript being highly transferable source languages. Overall, it demonstrates effective learning transfer across various programming languages.

Code summarization automates the process of generating natural language summaries for code snippets provided by developers. This technology generates summaries of code snippets in natural language, improving understanding and streamlining software maintenance. These summaries provide higher-level explanations of software systems, leveraging the extensive training of LLMs on textual data to infer and generate them effectively.

Sun et al. [Sun et al., 2023] assess ChatGPT's proficiency in code summarization by contrasting it with NCS, CodeBERT, and CodeT5 models. They prompt ChatGPT to generate comments for code and evaluate its performance using METEOR and BLEU metrics. ChatGPT achieves lower scores compared to the baseline models, notably with CodeT5 exhibiting superior results. Nevertheless, ChatGPT showcases a more profound comprehension of code semantics and delivers more comprehensive descriptions. However, it encounters challenges with programming language features and intricate code logic in comparison to CodeT5.

Code edit prediction is the process of foreseeing the modifications or adaptations needed to transition a code from one version to another. This endeavor centers on predicting the alterations developers will implement when refining code, such as transitioning from version 1 to version 2 or 3. It holds significant importance in software development, particularly during code refactoring or when incorporating new features. Gupta et al. [Gupta et al., 2023] address the challenge of predicting code edits, which is essential for tasks such as bug fixing and feature addition. They propose a method named Grace (Generation conditioned on Associated Code Edits), which enhances pre-trained LLMs by incorporating knowledge of relevant prior edits. By leveraging the generative capabilities of LLMs and conditioning code generation on past edits, their goal is to capture developers' intents more accurately. They assess two popular LLMs, Codex and CodeT5, in both zero-shot and fine-tuning scenarios. Their experiments reveal that Grace notably enhances LLM performance, leading to 29 % and 54 % more accurately-edited code suggestions compared to state-of-the-art symbolic and neural approaches across two datasets.

Software testing involves running a program or system to uncover errors or assess its attributes and capabilities, ensuring it delivers the expected outcomes.

In software engineering, fault localization is crucial as it assists developers in identifying the exact location of bugs for faster debugging. Researchers have investigated the use of LLMs, such as BERT and GPT, for this purpose. While LLMs have demonstrated remarkable effectiveness in numerous software engineering tasks, their application to Fault Localization (FL) remains relatively limited. FL entails identifying the specific code element responsible for a bug within a potentially extensive codebase. Nonetheless, harnessing LLMs for FL shows potential in improving both performance and interpretability for developers.

Mohsen et al. [Mohsen et al., 2023] present a phased bug localization method to overcome current limitations. The approach comprises three primary phases: raw data preparation, package classification, and source code recommendation. It takes a bug report and the source code of previous versions of the target system as input. Various details from the bug report, including summary, description, stack traces, and fixed source code files, are utilized and restructured during the raw data preparation phase. The package classification phase aims to identify the package containing the source code to be modified, thereby reducing the time required to locate the relevant files. Bidirectional Encoder Representations from Transformers (BERT) are applied in both the package classification and source code recommendation phases. Experimental findings illustrate that the proposed approach surpasses existing methods in terms of TOP-N rank and Mean Reciprocal Rank (MRR) evaluation metrics.

Vulnerability detection concentrates on identifying potential security vulnerabilities in software. This procedure is essential for protecting against malicious attacks and ensuring prompt patching of reported security flaws before they can be exploited. Steenhoek et al. [Steenhoek et al., 2023] conduct an empirical study to examine how learning models perform in identifying software vulnerabilities. They analyze and replicate nine learning-based Vulnerability Detection (VD) approaches, including two LLM-based methods. This research offers valuable insights into the current capabilities and effectiveness of LLMs in the field of vulnerability detection.

Test generation encompasses the creation of a set of test cases to evaluate the functionality of newly developed or updated software applications. Within this domain, unit test generation is a specialized focus area that revolves around generating test cases designed specifically for individual code units. In a detailed comparison conducted by Tang et al. [Tang et al., 2023] between ChatGPT and the advanced SBST tool EvoSuite, several notable findings emerge. ChatGPT successfully generates unit test cases for all 207 Java classes, with 69.6 % of them compiling and executing without manual intervention. However, certain cases encounter compilation errors due to ChatGPT's incomplete understanding of the entire project. SpotBugs detection reveals 403 potential defects among 204 test cases, most of which are low-priority issues. Nevertheless, the test suite generated by ChatGPT violates coding style conventions, particularly in indentation, indicating a need for improvement in code consistency and maintainability.

Graphical User Interface (GUI) testing is essential to ensure the accuracy and reliability of a mobile app's interface. This involves verifying that interactions with elements such as buttons and text boxes yield the expected results. GUI testing can be conducted manually or automatically, often employing metrics such as error detection to evaluate performance. Its primary goal is to guarantee the resilience and reliability of the app's interactive elements. Liu et al. [Liu et al., 2023b] present GPTDroid, a framework that transforms the GUI testing task into a question-answer format, utilizing LLMs as human-like testers. GPTDroid simplifies the process by feeding the application's GUI information to LLMs, allowing them to generate suitable testing scripts and provide feedback based on the obtained execution results. Evaluation of GPTDroid on 93 apps from Google Play demonstrates significant success, showcasing a 32 % enhancement in activity coverage. Additionally, GPTDroid identifies 53 previously undiscovered bugs on Google Play, with 35 of them being confirmed and addressed.

Natural Language Processing (NLP) testing entails assessing and scrutinizing the performance, precision, and resilience of NLP systems, encompassing tasks such as text generation and language comprehension. The objective is to verify the efficacy of these systems in comprehending and processing natural human language. Liu et al. [Liu et al., 2022] discuss the challenges encountered by modern question-answering (QA) systems due to their wide range of topics and task formats, complicating test collection and labeling tasks, consequently affecting their quality. To tackle this issue, the authors introduce QATest, a fuzzing framework based on metamorphic testing theory. The framework aims to automatically generate tests with oracle information for various QA systems. Additionally, they propose N-Gram coverage and perplexity priority to enhance testing efficiency and generate more tests capable of detecting erroneous behaviors based on question data features. Evaluation conducted on four QA systems illustrates that tests generated by QATest effectively detected hundreds of errors, while the testing criteria have enhanced both the diversity of testing and the efficiency of fuzzing.

Patch correctness assessment

Numerous contemporary program repair methodologies heavily rely on test suites created by developers to evaluate the accuracy of generated patches. However, these test suites often encompass only a fraction of the program's behavioral spectrum, resulting in an incomplete specification. Consequently, repair techniques may encounter the challenge of patch overfitting, wherein patches pass existing test suites but fail to generalize to other potential test scenarios. This limitation significantly hampers the practical applicability and acceptance of such repair methods in real-world contexts. Therefore, ensuring the correctness of patches becomes imperative for developers to effectively identify and discard overfitting patches post-generation.

Tian et al. [Tian et al., 2023] investigated the feasibility of employing representation learning models and supervised learning algorithms to statically predict the accuracy of patches generated by Program Repair (PR) tools. Their objective was to provide insights for enhancing the quality of repair candidates. Initially, they explored various distributed representation learning techniques, such as BERT, to capture the similarity between faulty and patched code fragments. Subsequent experiments focused on selecting threshold similarity scores to identify potentially incorrect patches produced by Automated Program Repair (APR) tools. They developed a framework named Leopard to forecast patch correctness, utilizing machine learning classifiers such as Decision Trees, Logistic Regression, Naïve Bayes, Random Forest, XGBoost, and Deep Neural Networks (DNN) with embeddings derived from BERT, Doc2Vec, and CC2Vec. Leopard, particularly with XGBoost and DNN utilizing BERT embeddings, demonstrated promising performance in predicting patch correctness, achieving high metrics such as Recall and F-Measure. Furthermore, they introduced Panther, an enhanced version of Leopard that combines learned embeddings with engineered features to improve classification accuracy. Utilizing SHAP (SHapley Additive exPlanations), they analyzed the rationales behind prediction outcomes to gain insights into identifying patch correctness. The study suggests integrating this approach with APR tools to efficiently explore a vast patch space in future endeavors.

Code review is a pivotal stage in software development aimed at ensuring the integrity of code. It involves developers meticulously scrutinizing code to assess factors such as logic, functionality, and style. Recent advancements in LLMs, notably the T5 model, have spurred innovations in automating code review processes. These advancements primarily seek to enhance code quality and review efficiency by focusing on tasks such as data collection, pre-training, and performance assessment. Li et al. [Li et al., 2022b] introduced CodeReviewer, a Transformer-based model inspired by the T5 architecture and initialized with parameters from CodeT5. CodeReviewer is designed to automate the code review process, thereby ensuring code quality. They gathered a comprehensive dataset from GitHub encompassing nine programming languages. Using this dataset, they pre-trained CodeReviewer on four tasks tailored for code review scenarios, focusing on comprehending code differences and generating relevant review comments. The model was evaluated on both the training dataset and a processed dataset, demonstrating superior performance compared to T5 and CodeT5-base. Overall, CodeReviewer enhances code review tasks, presenting promising advancements in automating and improving code quality assessment.

Bug reproduction is an essential phase in the software bug-fixing process, which entails recreating the environment where the bug originally occurred. This process involves a collaborative effort wherein developers gather information from users, including screenshots, logs, and issue descriptions. Despite its importance in software maintenance, reproducing bugs can be challenging, particularly when replicating the client-side context. To overcome these challenges, there is a need for a human-centered approach in research and tool design to assist developers effectively in bug reproduction.

Various automated test generation techniques have been devised to aid developers in test creation, with a primary emphasis on enhancing coverage or generating exploratory inputs. Nonetheless, these techniques frequently fail to fulfill semantic goals, such as accurately replicating bugs from bug reports. Kang et al. [Kang, Yoon, Yoo, 2023] emphasize the importance of bug reproduction, highlighting that approximately 28 % of tests added to open-source repositories stem from issue reports. Existing techniques primarily address program crashes due to challenges in transforming bug report semantics into test oracles. In response to this challenge, they propose LIBRO, a framework that harnesses LLMs capable of performing code-related tasks. Although LLMs cannot directly execute faulty code, LIBRO incorporates post-processing steps to discern effective LLM outputs and rank them based on validity. Evaluations conducted on the Defects4J benchmark indicate that LIBRO can generate failure-reproducing test cases for 33 % of the analyzed cases, recommending bug-reproducing tests as top candidates for 149 bugs. Additionally, LIBRO's performance when compared against post-training bug reports suggests its potential to significantly enhance developer efficiency by automatically generating tests from bug reports.

Software testing is essential in software development; however, test cases often lag behind production code changes, resulting in increased maintenance costs and the potential for bugs. To address this issue, Hu et al. [Hu et al., 2023] introduced CEPROT (Co-Evolution of Production-Test Code), a method designed to automatically update outdated test cases. CEPROT consists of two stages: identifying obsolete tests and updating them. The method utilizes two datasets sourced from Java projects with extensive unit tests for its training process. During training, CEPROT learns from both positive and negative samples, where positive samples necessitate updating due to production code changes, while negative samples do not. When compared to various baseline models such as KNN, SITAR, LSTM, and NMT, CEPROT achieves impressive precision, recall, and F1 scores of 98.3 %, 90.0 %, and 94.0 %, respectively, in identifying obsolete test cases.

Conclusion

Researchers have extensively applied LLMs across numerous software engineering (SE) tasks, spanning 43 different areas. These tasks encompass a wide spectrum, ranging from traditional ones like program repair to more intricate tasks such as fuzzing and GUI testing. While LLMs demonstrate proficiency in foundational SE tasks, including software testing and development, there is a need for more targeted efforts to address domain-specific challenges. This could involve designing specialized LLMs or integrating them into existing research workflows. Despite their widespread application in software testing and development, the utilization of LLMs in software requirements remains relatively underexplored, presenting a promising area for future exploration and research within the SE domain.

References

Baltaji R., Pujar S., Mandel L., Hirzel M., Buratti L., Varshney L. Learning transfers over several programming languages // arXiv preprint. — 2023. — arXiv:2310.16937

Brie P., Burny N., Sluÿters A., Vanderdonckt J. Evaluating a large language model on searching for gui layouts // Proceedings of the ACM on Human-Computer Interaction. — 2023. — Vol. 7, No. EICS. — P. 1–37.

Chowdhary K., Chowdhary K. R. Natural language processing // Fundamentals of artificial intelligence. — 2020. — P. 603–649.

Dong Y., Jiang X., Jin Z., Li G. Self-collaboration code generation via ChatGPT // ACM Transactions on Software Engineering and Methodology. — 2024. — Vol. 33, No. 7. — P. 1–38.

Gupta P., Khare A., Bajpai Y., Chakraborty S., Gulwani S., Kanade A., Tiwari A. GrACE: generation using associated code edits // arXiv preprint. — 2023. — arXiv:2305.14129

Hadi M. U., Qureshi R., Shah A., Irfan M., Zafar A., Shaikh M. B., Mirjalili S. A survey on large language models: Applications, challenges, limitations, and practical usage // Authorea Preprints. — 2023.

Hasan Md. R., Li J., Ahmed I., Bagheri H. Automated repair of declarative software specifications in the era of large language models // arXiv preprint. — 2023. — arXiv:2310.12425

Hey T., Keim J., Koziolek A., Tichy W. F. Norbert: Transfer learning for requirements classification // 2020 IEEE 28th international requirements engineering conference (RE). — 2020. — P. 169–179.

Hu X., Liu Z., Xia X., Liu Z., Xu T., Yang X. Identify and update test cases when production code changes: A transformer-based approach // 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2023. — P. 1111–1122.

Kang S., Yoon J., Yoo S. Large language models are few-shot testers: Exploring LLM-based general bug reproduction // 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). — 2023. — P. 2312–2323.

Khan M. A., Khan M. S., Khan I., Ahmad S., Huda S. Non functional requirements identification and classification using transfer learning model // IEEE Access. — 2023.

Kou B., Chen S., Wang Z., Ma L., Zhang T. Is model attention aligned with human attention? An empirical study on large language models for code generation // arXiv preprint. — 2023. — arXiv:2306.01220

Li J., Zhao Y., Li Y., Li G., Jin Z. Acecoder: Utilizing existing code to enhance code generation // arXiv preprint. — 2023. — arXiv:2303.17780

Li X., Gong Y., Shen Y., Qiu X., Zhang H., Yao B., Qi W., Jiang D., Chen W., Duan N. Coderetriever: A large scale contrastive pre-training method for code search // Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. — 2022a. — P. 2898–2910.

Li Z., Lu S., Guo D., Duan N., Jannu S., Jenks G., Sundaresan N. Automating code review activities by large-scale pre-training // Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2022b. — P. 1035–1047.

Liu R., Yang R., Jia C., Zhang G., Zhou D., Dai A. M., Yang D., Vosoughi S. Training socially aligned language models in simulated human society // arXiv. — 2023a. — https://arxiv.org/abs/2305.16960

Liu Z., Chen C., Wang J., Chen M., Wu B., Che X., Wang Q. Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions // arXiv preprint. — 2023b. — arXiv:2310.15780

Liu Z., Feng Y., Yin Y., Sun J., Chen Z., Xu B. QAtest: A uniform fuzzing framework for question answering systems // Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. — 2022. — P. 1–12.

Mohsen A. M., Hassan H., Wassif K., Moawad R., Makady S. Enhancing bug localization using phase-based approach // IEEE Access. — 2023.

Mu F., Shi L., Wang S., Yu Z., Zhang B., Wang C., Liu S., Wang Q. ClarifyGPT: Empowering LLM-based code generation with intention clarification // arXiv preprint. — 2023. — arXiv:2310.10996

Salza P., Schwizer C. Gu J., Gall H. C. On the effectiveness of transfer learning for code search // IEEE Transactions on Software Engineering. — 2022.

Steenhoek B., Rahman M. M., Jiles R., Le W. An empirical study of deep learning models for vulnerability detection // 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). — 2023. — P. 2237–2248.

Subahi A. F. BERT-based approach for greening software requirements engineering through non-functional requirements // IEEE Access. — 2023.

Sun W., Fang C., You Y., Miao Y., Liu Y., Li Y., Chen Z. Automatic code summarization via ChatGPT: How far are we? // arXiv preprint. — 2023. — arXiv:2305.12865

Tang Y., Liu Z., Zhou Z., Luo X. ChatGPT vs SBST: A comparative assessment of unit test suite generation // arXiv preprint. — 2023. — arXiv:2307.00588

Thawkar O., Shaker A., Mullappilly S. S., Cholakkal H., Anwer R. M., Khan S., Laaksonen J., Shahbaz Khan F. XrayGPT: chest radiographs summarization using medical vision-language models // arXiv. — 2023. — https://arxiv.org/abs/2306.07971

Tian H., Liu K., Li Y., Kaboré A. K., Koyuncu A., Habib A., Bissyandé T. F. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches // ACM Transactions on Software Engineering and Methodology. — 2023. — Vol. 32, No. 4. — P. 1–34.

Weng L. LLM-powered autonomous agents // 2023. — lilianweng.github.io

Xie D., Yoo B., Jiang N., Kim M., Tan L., Zhang X., Lee J. S. Impact of large language models on generating software specifications // arXiv preprint. — 2023a. — arXiv:2306.03324

Xie Y., Lin J., Dong H., Zhang L., Wu Z. Survey of code search based on deep learning // ACM Transactions on Software Engineering and Methodology. — 2023b. — Vol. 33, No. 2. — P. 1–42.

Yang G., Zhou Y., Chen X., Zhang X., Xu Y., Han T., Chen T. A syntax-guided multi-task learning approach for Turducken-style code generation // Empirical Software Engineering. — 2023. — Vol. 28, No. 6. — P. 141.

Zhang K., Li Z., Li J., Li G., Jin Z. Self-edit: Fault-aware code editor for code generation // arXiv preprint. — 2023a. — arXiv:2305.04087

Zhang Q., Fang C., Xie Y., Zhang Y., Yang Y., Sun W., Yu S., Chen Z. A survey on large language models for software engineering // arXiv preprint. — 2023b. — arXiv:2312.15223

Zhao W. X., Zhou K., Li J., Tang T., Wang X., Hou Y., Min Y., Zhang B., Zhang J., Dong Z., Du Y., Yang C., Chen Y., Chen Z., Jiang J., Ren R., Li Y., Tang X., Liu Z., Liu P., Nie J.-Y., Wen J.-R. A survey of large language models // arXiv preprint. — 2023. — arXiv:2303.18223