Challenges to Using LLMs in Code Generation and Repair

Liliana Pasquale, Antonino Sabetta, Marcelo d'Amorim, Peter Hegedus, Mehdi Tarrit Mirakhorli, Hamed Okhravi, Mathias Payer, Awais Rashid, Joanna C. S. Santos, Jonathan Spring, Lin Tan, Katja Tuma

LLMs hold great promise in solving many challenges arising from software complexity, including the possibility of automating code generation and repair. Although we cannot deny the groundbreaking nature of LLM-based code repair, we must be realistic in positioning current results. This column explores the challenges in using LLMs for automated code generation and program repair.

Introduction

Joanna C. S. Santos, Mathias Payer

With the latest advances in LLMs, AI-based code development assistants are increasingly part of day-to-day software development. A recent study (https://tinyurl.com/3kub3awn) of 500 US-based developers showed that 92% use AI coding assistants for work and personal use. The increased productivity perceived by developers partly explains this fast, widespread adoption; AI helps them automate repetitive tasks so that they can focus on higher-level challenging tasks [1].

Peter Hegedus, Joanna C. S. Santos, Lin Tan

Automated program repair (APR) aims to generate source code to fix software defects and vulnerabilities automatically. Research on APR has advanced significantly with generative AI models. LSTM models achieved notable success in generating complex, syntactically correct code after training on extensive source code datasets. LLMs further improved automated program repair. Since they are pre-trained on an enormous amount of natural language text and source code, they also offer an out-of-the-box solution for code repair. Recent studies [2, 3, 10, 11] show that LLMs can fix issues in the code, such as defects, vulnerabilities, and code smells. In some cases, code repair is treated as a code generation task with a prompt explicitly instructing the model to fix a problem in a given location [2, 3].

Peter Hegedus

While LLMs, such as GPT-4, excel at fixing functional bugs in laboratory environments (i.e., on synthetic or isolated issues), their real-world application - especially when the task is fixing complex, security-related issues—remains limited [4].

LLMs generate vulnerable and incorrect code.

Awais Rashid

Software professionals are concerned about AI-generated code quality, correctness, and security and the need to scrutinise and validate such code [5]. This is particularly critical for program repair. The CrowdStrike case has highlighted how errors in a single patch can have a global impact, halting critical services.

Marcelo d'Amorim

There is evidence that LLMs can produce code containing security weaknesses even when the user of the LLM is not malicious [6]. Prevention and detection are two directions to mitigate this problem. For *prevention*, responsibly disclosing the weaknesses of an LLM to the public encourages the LLM maintainers to curate training datasets actively. Users must know the threats and limitations associated with the versions of the LLMs they are using. LLM maintainers are expected to care about public announcements about weaknesses in their LLM and will address them in subsequent releases. The LVE Repository (https://lve-project.org/) is a commendable global initiative in that direction. For *detection*, LLMs can be used to explain the weaknesses identified by third-party tools. Ideally, those explanations should describe the consequences of not taking some action to mitigate the weakness, i.e., counterfactual explanations are likely more helpful to users. Such explanations should help the distracted trained developer and help to train inexperienced developers.

Hamed Okhravi

Source code often must comply with many other requirements besides functionality. These may include soft and hard real-time constraints, power usage requirements (e.g., for embedded code), and side-channel resilience (e.g., for crypto code), as well as more generic non-functional requirements such as readability, maintainability, performance, portability, testability, and modularity. LLM-generated code rarely accounts for these requirements.

LLMs must also understand the underlying platforms to generate the correct code to fix specific bugs [9]. Platform-specific parameters may include Windows vs. Linux file handling, 32-bit vs. 64-bit code, Windows vs. POSIX threading API, network socket differences, or memory alignment. To successfully repair code, the LLM should be trained on all those platforms, and detailed platform information must be provided when prompting it to repair source code.

Joanna C. S. Santos, Antonino Sabetta

LLMs have a prompt input and output size threshold (e.g., GPT-4 can take up to 128,000 tokens and generate up to 16,384 tokens). Considering real software systems' sheer complexity and size, these thresholds are insufficient. As such, LLMs may miss the broader context of a project and can generate a limited repair size. Understanding the complete environment in which the code operates (e.g., configuration files, external dependencies, database structures, etc.) is crucial for code generation and repair.
However, despite improvements in token counts (e.g., Gemini 1.5 allows up to 1 million tokens), capturing sufficient relevant context may require more than an extra-large token count. Effective code repair requires context-aware reasoning. That is, it requires understanding the structure and purpose of the overall application so that repairs are generated following the codebase's security needs and technical constraints.

Mathias Payer

AI-based assistants must be sufficiently scoped to create correct code, especially in highly optimised environments. Although research has explored integrating LLMs into automated testing, the results only marginally improve on existing methods when incorporating the cost of LLM queries. A more promising application of LLMs is in generating test drivers to target

specific functionalities, as they can generate and refine drivers to improve code path coverage. While manually-written drivers often fall short, LLMs could fill these gaps and enhance API coverage. However, LLM-generated drivers may be flawed or incomplete, potentially leading to false positives and wasted resources.

Peter Hegedus

The reproducibility of the fixing process is a major challenge, as LLM results are non-deterministic. Since prompts can have a major impact on the results, instead of model training, one would need to invest effort into prompt engineering.

Katja Tuma

From experience assessing the effectiveness of LLMs in fixing security misconfigurations in Kubernetes-based applications [7], existing tools (Checkov, Datree, and KICS, to name a few) adopt different rules and security policies to identify security misconfigurations. These tools may produce both false positives and negatives. Some configurations (such as allowing network access to a container) might be flagged as insecure while they are required for the running application to perform its key functionalities (e.g., network monitoring), which can only become apparent at runtime when the applications are deployed in the cluster. It is up to the administrator to find the right configuration, but LLMs could be used to help. Keeping the human in the loop is essential: For infrastructure-as-code repair with LLMs, first, we need to distinguish between misconfiguration fixes that can (and should) be verified by humans and those that could potentially be automated with limited security risks. Second, we need to establish a common taxonomy of misconfigurations and robustness measures for more effective tool benchmarking and experimental validation. This could help associate a certain level of confidence in the LLM-generated fixes for certain types of misconfigurations and instead leave the (orders of magnitude smaller) remaining set of issues for humans to handle.

Insufficient training data and how to add software domain knowledge

Hamed Okhravi

Supervised approaches may be necessary for APR to succeed. To achieve this aim, LLMs must capture a solid notion of vulnerable and secure code to repair code successfully. However, realistic data for vulnerable and secure code samples is insufficient to apply supervised learning. The entire National Vulnerability Database (NVD) contains around 260K vulnerabilities at the time of writing. Consider further that not every reported vulnerability has an associated code sample available, and some vulnerabilities on NVD are too old to be relevant to modern code. As a result, there are often less than tens of thousands of vulnerable code samples on which to train an LLM. This is insufficient to ensure the LLM is properly trained to generate only secure code. Recent work in this domain suggests that enriching existing data with additional properties (context, syntax, and semantics) allows one to achieve better accuracy, precision, and recall in distinguishing between vulnerable and secure code [8].

Joanna C. S. Santos

Prior work [11] examined whether LLMs could repair their generated insecure code. Stark differences exist between the issues LLMs could repair for each programming language. For

example, for Python, LLMs can solve issues related to XML validation vulnerabilities but are less capable of solving issues related to the Use of a Broken or Risky Cryptographic Algorithm (CWE-327), Path Traversal (CWE-22), and Incorrect Permission Assignment for Critical Resource (CWE-732). We also observed that, overall, LLMs were more capable of repairing Python code than Java code. These results indicate open challenges in effectively using LLM to repair insecure code. LLMs are trained with popular languages, especially Python. Consequently, LLMs will struggle to repair insecure code for languages with fewer samples in their training data. Even in cases where the language is well-covered, a model generates repairs to insecure code based on historical data. Still, vulnerabilities and secure coding practices continually change as technology evolves. Thus, the precision observed today likely will not be the same tomorrow.

Lin Tan

Another important question is *whether adding more data to train deep learning (DL) models, including LLMs, is a promising direction to improve APR techniques.* Using increasingly large amounts of data has succeeded in tasks such as speaking a natural language, which may fundamentally differ from coding tasks. Babies learn to speak their mother tongue by mimicking and learning implicitly from what they hear. However, software developers do not simply learn programming and program repair by reading code and patches; *they also use* logic and reasoning by taking programming, algorithms, and data structure courses. Thus, while adding more data may improve LLMs for text and other modalities, it may not be the most effective approach for APR tasks. Adding explicit domain knowledge (including but not limited to type rules) to models may be a more effective approach [9]. On the other hand, models may not need to learn the same way humans do, and the most effective learning approaches for humans may not be the most effective ones for models, suggesting that more data could be more effective.

Recent DL-based program repair techniques provide conflicting results in this respect. For example, KNOD employs a domain-rule distillation technique to explicitly inject domain knowledge including types into the neural network decoding procedure [9]. Specifically, the domain-rule distillation technique (1) represents syntax and semantics as rules in first-order logic, and (2) uses these logic rules to refine the teacher-student probability distributions to guide the model to learn to follow these syntactic and semantic rules. This approach shows that adding domain knowledge explicitly improves the effectiveness of neural networks for program repair. Yet, other studies (e.g., [10]) suggest the opposite. They show that LLMs for code, *without or with fine-tuning*, outperform existing DL-based program repair techniques specially designed for APR to fix software defects. These generic LLMs for code are pre-trained with a vast amount of data but are not designed for APR. Since these LLMs are typically trained with more data than existing DL-based APR approaches, the finding suggests that more data could be more effective for improving LLM-based program repair. The next relevant open questions are (1) *whether we have more data for DL models to improve automated program repair and code generation and (2) how to add domain knowledge to LLMs effectively.*

Limited LLMs Accountability and Overreliance

Peter Hegedus

Another major challenge with LLM-based code repair is the validation of the fixes they produce. It is not always easy to determine if an LLM-generated patch is genuinely good, meaning humans still play an essential role in verifying the correctness of the generated patches.

Marcelo d'Amorim

A challenge is avoiding hallucinations, which can be especially detrimental to inexperienced developers, who may not realise incoherences in the discourse.
The complementary problem of vulnerability repair can be even more challenging in practice if we consider the possibility of developers accepting plausible patches recommended by an LLM. The possibility of introducing bugs or other vulnerabilities when repairing code is well-known in software engineering, but security weaknesses can be more consequential. Developers need to validate the security patches that automated tools generate. However, for small single-hunk patches ---which are prevalent--- the human cost of reviewing may well dominate the cost of writing the patch. So, the benefit of automated repair in that context is questionable. It is therefore important (1) to focus automated repair on multiple hunk patches, (2) to develop tools capable of explaining the repairs, and (3) to ensure developers validate these patches.

Jonathan Spring

Developers need a robust development environment to place more trust in the outputs of an LLM. That means good specification and documentation of the application programming interface of the project or module, adequate unit tests, adequate integration tests, repeatable build processes, appropriate program verification techniques to detect specific common classes of vulnerabilities, appropriate testing to check parsing and error handling, and so on. An organisation should have these tools established and working well before moving to automated code repair.

However, there are some critical tasks an LLM cannot do. An LLM cannot take ownership of maintaining a software product that is out of support or end-of-life. An LLM cannot automatically write in interoperable, open standards for communication and data formats. Free and open standards will help others (using an LLM tool or not) repair your code after you move on to another project.

With or without LLM assistance, a software vendor should meet the goals of CISA's Secure by Design initiative. When a software vendor offers a product on which the engineers use LLM-based code repair, the vendor should provide software transparency and vulnerability management. A system owner or acquisitions team should still ask for a Software Bill of Materials and ask the vendor about their vulnerability disclosure and reporting practices. Vendors should still pledge the organisational work to make software secure by design.

If we demand that software is secure by design, tools such as LLMs for code repair can help software developers meet that demand.

Awais Rashid

Several open questions surround the quality of LLM outputs. Would we see situations where the computer (LLM) says “no repair is needed” when one is required or where it hallucinates one? Similar questions arise about the repair itself. Who and how will scrutinise and validate the repair so it does not introduce undesirable side effects, such as impacting other software functionalities or introducing security weaknesses or vulnerabilities?

There is an expectation that the developer's role will change – from the *driver* who writes the code to a *navigator* who will check and validate the driver’s work, that is, the LLM’s. However, we also know that automation and reliance on tools erode skills. I am reminded of a problem with my car: The hazard lights kept coming on when parked, draining the battery. Neither the small handheld diagnostic computer (with the repair person) nor the extensive diagnostic rig at the garage could replicate the issue or isolate the fault. The problem kept recurring until a different repair person came out to recharge the battery, used the same handheld diagnostic computer to no effect, gave it some thought and then noted that it was likely a faulty burglar alarm. He isolated it, and the problem was solved. Even if we use LLMs for code repair, we need skilled software engineers to understand, scrutinise and validate the outcomes.

Liliana Pasquale

LLMs can generate code that no longer satisfies system requirements or introduces vulnerabilities. Despite this, their growing power has led software engineers to increasingly depend on them, sometimes overly. This overconfidence becomes concerning as developers rely on LLMs for coding and program repair, where accuracy is critical. Existing AI coding assistants should identify the criticality of software development tasks and configure the reliance that developers can place on them accordingly. For example, LLMs can still be useful for several applications where errors can be tolerated. Thus, developers can entirely rely on LLMs to automate simple and repetitive programming tasks in non-critical applications. More complex programming tasks of non-critical applications could require the supervision of a senior software engineer to review the code generated automatically. New and large programming tasks, especially for critical applications, may require using LLMs only to oversee software development activities, such as generating test cases or performing code reviews.

Mehdi Tarrit Mirakhorli

Code repair generated by LLMs, while often functional, provides no guarantees that the repaired code is free of vulnerabilities, meets specific safety criteria or truly addresses the underlying requirements. This lack of assurance can be problematic, especially in critical systems where correctness, security, and performance are non-negotiable. One idea is to use LLMs to generate test cases and validate the repaired or synthesised code. However, a stronger idea is to provide proof of correctness. Since proofs equate with programs, one can deliver an LLM-based approach to generate proofs of correctness automatically using similar programs. We discussed the foundation of shifting towards certified code repair, where LLMs are integrated with formal verification techniques [12]. Based on the theory that proofs can equate with programs, we can think of generating proofs as a task similar to generating code. This theoretical foundation suggests that with appropriate training and fine-tuning, LLMs can be guided to produce not only code repairs but also formal proofs that guarantee the correctness of the generated solutions. In such a transformative approach, along with the code

fix, the LLM generates a formal proof that certifies the repaired code satisfies a set of predefined safety, correctness properties, security policies, or design rules. A lightweight verification tool can independently check the proof, ensuring the code fix meets the necessary safety criteria before deployment.

Certified code repair (or synthesis) is foundational for enabling AI autonomously and developing secure and trustworthy systems. Pre-LLMs and through my NSF CAREER award, I focused on the challenges of realising such a foundational approach where software engineers could focus on key engineering tasks (1) creativity and (2) design, then collaborate with a design synthesis tool to generate low-level code that correctly implements their design choices. While we are closer to such an idea today, there are challenges to achieving it for modern large-scale systems. For instance, generating formal proofs for code repairs can be computationally expensive, especially for large and complex systems. Proof generation requires rigorous formalisation of the code's properties and behaviour, and ensuring that these properties hold under all conditions can be time-consuming. Also, modern software has many third-party dependencies, adding to the complexity of generating proof of correctness. Fine-tuning LLMs on datasets that include examples of formal methods, symbolic reasoning, and proof-generation tasks can help bridge this gap. Integrating language models with formal proof engines could also enhance their capabilities in proof generation.

Opportunities for Software Testing

Joanna C. S. Santos

LLMs cannot simply be used off the shelf as a foolproof tool to solve the insecure code repair problem. LLMs should ***enhance*** classic APR techniques rather than fully ***replace*** them. Such a hybrid approach has been shown by prior work to help in generating tests [14]. In that context, LLMs generated more diversified inputs to increase test coverage for an underlying search-based software-testing approach.

Mathias Payer

Two key areas are certainly human-in-the-loop code completion and the generation of unit tests and fuzzers. Automated testing, particularly fuzzing, has experienced a meteoric rise in popularity, mirroring the growth of large language models (LLMs) in computer science. Despite its conceptual simplicity, fuzzing effectively uncovers bugs by randomly probing a wide range of inputs to expose program vulnerabilities. A promising application of LLMs is generating test drivers to target specific functionalities [15], as they can create and refine drivers to improve code path coverage. While manually written drivers often fall short, LLMs could fill these gaps and enhance API coverage. However, LLM-generated drivers may be flawed or incomplete, potentially leading to false positives and wasted resources.

A promising use case of LLMs is in the bug-fixing process [3]. After a fuzzer detects a bug and generates test inputs to reproduce it, an LLM could assist the developer by iteratively suggesting patches to address the underlying vulnerability. The fuzzer could then explore the patched code to uncover any lingering weaknesses of the patch. This iterative approach, alternating between fuzzers and LLMs, may lower developer involvement and reduce the costs of producing a complete patch. A hybrid approach combining fuzzers, LLMs, and developers

could be a promising future direction for integrating LLMs into the bug discovery and remediation cycle. As it neither increases costs nor produces false positives, this approach is likely the most interesting angle for LLMs, but it will require careful customisation and optimisation.

However, while LLMs offer significant potential for enhancing fuzzing, the baseline approach without LLMs is already highly optimised, and the cost of querying LLMs must be carefully balanced against the potential benefits. LLMs trained on source code and specifications may improve mutation operators and driver generation, but some challenges, such as false positives, remain.

Conclusion

Awais Rashid

"*Many people expect advances in artificial intelligence to provide the revolutionary breakthrough that will give order-of-magnitude gains in software productivity and quality. I do not.*" wrote Fred Brooks Jr. in *No Silver Bullet*, his seminal 1986 essay tackling essential and accidental complexity in software engineering [15].
Will LLMs for code repair tasks alleviate essential complexity or exacerbate accidental complexity? Unless we systematically address issues such as correctness, verifiability and explainability, LLMs will likely add accidental complexity – potentially order-of-magnitude – to the task of program repair, thus eroding any gains they may provide.

There are several open questions about the quality of LLM outputs. Would we see situations where the computer (LLM) says "no repair is needed" when one is required or where it hallucinates one? Similar questions arise about the repair itself. Who and how will scrutinise and validate the repair so it does not introduce undesirable side effects, such as impacting other software functionalities or introducing security weaknesses or vulnerabilities? Time will tell. Let us know what your experience and opinions are.

References

[1] A. Ziegler, E. Kalliamvakou, X. Li, A. Rice, D. Rifkin, S. Simister, E. Aftandilian, "Productivity Assessment of Neural Code Completion". In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS)*, ACM, New York, NY, USA, pp. 21-29, 2022, doi: 10.1145/3520312.3534864.

[2] H. Joshi. J.C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, I. Radiček, "Repair is Nearly Generation: Multilingual Program Repair with LLMs." In *Proceedings of the 37th AAAI Conference on Artificial Intelligence and 35th Conference on Innovative Applications of Artificial Intelligence and 13th Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*, pp.5131-4140, 2023. Doi: 10.1609/aaai/v37i4.25642.

[3] H. Pearce, B. Tan, B. Ahmad, R. Karri, B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models". In *Proceedings of the2023 IEEE Symposium on Security and Privacy (SP),* pp. 2339-2356, IEEE Computer Society, 2023. Doi: 10.1109/SP46215.2023.10179324.

[4] Z. Ságodi, G. Antal, B. Bogenfürst, M. Isztin, P. Hegedűs, and R. Ferenc, “Reality Check: Assessing GPT-4 in Fixing Real-World Software Vulnerabilities”. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*, pages 252-261. ACM, 2024. Doi: 10.1145/3661167.3661207.

[5] J. H. Klemmer, S. A. Horstmann, N. Patnaik, C. Ludden, C. Burton Jr., C. Powers, F. Massacci, A. Rahman, D. Votipka, H. R. Lipford, A. Rashid, A. Naiakshina, S. Fahl, “Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns”, In *Proceedings of 31st ACM Conference on Computer and Communications Security (CCS)*, pp. 2726-2740, ACM, 2024. Doi: 10.1145/3658644.3690283.

[6] S. Hamer, M. d'Amorim and L. Williams, "Just Another Copy and Paste? Comparing the Security Vulnerabilities of ChatGPT Generated Code and StackOverflow Answers," in 2024 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, 2024, pp. 87-94, doi: 10.1109/SPW63631.2024.00014.

[7] F. Minna, F. Massacci, and K. Tuma, “Analyzing and Mitigating (with LLMs) the Security Misconfigurations of Helm Charts from Artifact Hub,” 2024. [Online]. Available: *arXiv preprint arXiv:2403.09537*.

[8] S. Shimmi, A. Rahman, M. Gadde, H. Okhravi, M. Rahimi, “VulSim: Leveraging Similarity of Multi-Dimensional Neighbor Embeddings for Vulnerability Detection,” In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security’24*), Philadelphia, PA, USA, August 14-16, 2024, pp. 1777--1794, 2024. [Online]. Available: https://www.usenix.org/system/files/usenixsecurity24-shimmi.pdf

[9] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, X. Zhang. “Knod: Domain Knowledge Distilled Tree Decoder for Automated Program Repair.” In *Proceedings of the 45th International Conference on Software Engineering (ICSE),* pp. 1251-1263, IEEE, 2023. Doi: 10.1109/ICSE48619.2023.00111.

[10] N. Jiang, K. Liu, T. Lutellier and L. Tan, "Impact of Code Language Models on Automated Program Repair," In *Proceedins of the 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, 2023, pp. 1430-1442, doi: 10.1109/ICSE48619.2023.00125.

[11] M. Siddiq, L. Casey, J. C. S. Santos, “FRANC: A Lightweight Framework for High Quality Code Generation.” To appear in *Proceedins of the24th International Conference on Source Code Analysis and Manipulation (SCAM’24),* IEEE, 2024. [Online]. Available: https://arxiv.org/abs/2307.08220.

[12] M. Fazelnia, M. Mirakhorli, H. Bagheri, “Translation Titans, Reasoning Challenges: Satisfiability-Aided Language Models for Detecting Conflicting Requirements.” In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024) – New Ideas and Emerging Results*, ACM, pp. 2294-2298, 2024. Doi: 10.1145/3691620.3695302.

[13] C. Lemieux, J. P. Inala, S. K. Lahiri, S. Sen, “Codamosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models.” In *Proceedings of the* 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 919-931, IEEE. Doi: 10.1109/ICSE48619.2023.00085.

[14] Y. Lyu, Y. Xie, P. Chen, and H. Chen, “Prompt Fuzzing for Fuzz Driver Generation.” In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS’24), pp. 3793-3807, ACM, 2024. Doi: 10.1145/3658644.3670396.
[15] F. P. Brooks Jr., “No Silver Bullet–Essence and Accident”, The Mythical Man-Month, Essays on Software Engineering, Anniversary Edition, pp. 177-203, Addison-Wesley, 1995. Doi: 10.1109/MC.1987.1663532