Universal Library of Innovative Research and Studies Research Article

ISSN: 3065-0003 | Volume 3, Issue 1

Open Access | PP: 01-07

DOI: https://doi.org/10.70315/uloap.ulirs.2026.0301001

Comparison of Traditional and AI-Mediated Approaches to Solving Engineering Problems

Burmistrov Aleksandr

Chief Technology Officer at Piogroup LTD, Krakow, Poland.
https://orcid.org/0009-0000-7576-3932

Abstract

This article looks at how older engineering methods and newer AI-based ones differ when they are used to solve software-engineering problems. Most of the examples come from recent empirical studies, several review papers, and work on deep-learning models. The aim of the article is to look at how the older metric-based techniques compare with the newer ideas built around machine learning, deep learning, and large language models. The main result show that the traditional methods still help because they are clear and fairly steady to work with, but they do not capture the actual semantic behaviour of code very well. The AI models — especially the hybrid setups and the transformer ones — tend to score higher and can be used in more parts of the development process. These systems also introduce their own difficulties. Many of them depend on fairly large datasets, and it is often hard to work out what led the model to a specific output, even when the result looks reasonable. Re-running the same experiment does not always produce the same behaviour either, which can complicate evaluation. In practice, teams sometimes have to reshape parts of their workflow simply to make the tools usable. The article will interest practitioners seeking to streamline their daily work with the new technology.

Keywords: *Traditional Engineering Methods, Software Defect Prediction, Deep Learning, Hybrid Models, Semantic Code Analysis, Large Language Models (LLMs), AI-Mediated Engineering, Software Quality, Machine Learning In Software Engineering, Engineering Workflow Transformation.*

INTRODUCTION

Engineering work has, for a long time, leaned on structured analytic routines and on metric sets designed by practitioners who knew the systems they were evaluating. In software engineering, this has usually meant things like static analysis, object-oriented complexity measures, and various process indicators or rule-driven checks. These methods remain in use largely because people know how to interpret them and because they behave predictably across different projects. As software systems have kept growing and getting more tangled with each other, the limits of those hand-made indicators have become harder to ignore. They often miss the bits of meaning or context that start to matter once a project gets big enough.

In the last few years, people have tried using AI to handle the same issues from another angle. Instead of relying only on whatever features were defined ahead of time, a lot of machine-learning and deep-learning models try to build their own representations from the data they are given. Some hybrid designs even mix the older statistical descriptors with learned semantic features and tend to produce stronger results than either of the two alone [1,2]. Large language models have pushed this further still, taking on tasks that would previously have required separate tools—formulating problems, drafting code, spotting bugs, generating tests, and assisting with maintenance work [3,4]. These systems frequently outperform earlier methods, but their use is tied up with questions about transparency and reproducibility, and with whether organisations actually have the data, infrastructure, or expertise needed to use them effectively [6–10].

Even with all this work going on, the literature still tends to treat the traditional methods and the AI-based ones as if they were separate topics. A lot of studies look at a single model on its own and mainly report accuracy or similar numbers, without asking how the approach itself shapes the way engineers think about the problem. Because of that, researchers do not have a clear sense of how the two methods compare in terms of interpretability, the practical limits they run into, or how smoothly they fit into the development lifecycles that teams actually use. This article

Citation: Burmistrov Aleksandr, "Comparison of Traditional and AI-Mediated Approaches to Solving Engineering Problems", Universal Library of Innovative Research and Studies, 2026; 3(1): 01-07. DOI: https://doi.org/10.70315/uloap.ulirs.2026.0301001.

attempts to close that gap by bringing together findings from recent empirical studies, reviews, and industry accounts to look at the two traditions side by side and explore what each one contributes—and what each one struggles with—in everyday software development.

METHODS AND MATERIALS

This study draws on a review of ten peer-reviewed papers published between 2021 and 2025, with each one reflecting a different line of work on traditional and AI-based approaches in software engineering. Abdu et al. looked at how semantic information from abstract syntax trees can be combined with older PROMISE metrics in a hybrid CNN–MLP setup for defect prediction [1]. Albattah and Alzahrani compared a range of machine-learning and deep-learning algorithms to see which of them performs best for defect prediction in practice [2]. Alenezi and Akour focused on AI tools used across the software-development lifecycle and discussed their benefits as well as the obstacles teams face when trying to adopt them [3]. Banh, Holldack, and Strobel examined how generative AI systems are starting to change everyday engineering work, including the way tasks are divided and how developer roles shift inside organisations [4]. Batool and Khan assessed several deep-learning models—CNNs, RNNs, LSTMs—to see how well they compare with traditional methods for predicting software faults [5].

Giray et al. carried out a broad survey of how deep learning has been used in defect prediction and paid particular attention to recurring methods, common problems in published work, and the state of reproducibility across the papers they examined [6]. Hasan and Mohi-Aldeen approached the field through a systematic review, gathering together the algorithms, datasets, and performance issues that appear most often in deep learning–based defect prediction research [7]. Hou et al. looked at a different part of the landscape by reviewing how large language models are being applied in software engineering, mapping their use in tasks such as generating code, debugging, writing tests, and supporting maintenance activities [8]. Pan, Lu, and Xu tested CodeBERT in a practical setting and focused on whether it actually picks up the semantic parts of code, as well as how often it does better than the older defect-prediction methods on the same problems [9]. Sikic and colleagues approached the issue differently. They built a graph-neural-network model that uses the structural connections already built into source code, and in their tests this allowed them to push classification accuracy higher [10].

These studies, together with the earlier ones, cover quite a wide range: traditional metric-based engineering work, different strands of machine learning and deep learning, hybrid setups, and the newer applications built around large language models and other generative systems. Viewed together, they show a field that is gradually broadening its predictive tools and its workflow support, while also revealing how engineers and automated systems now share parts of the work that used to be done entirely by hand.

Even with all of this research, some gaps are still easy to spot. A lot of the work looks either at the older metric-based techniques or at single AI models on their own, without really asking how these two ways of approaching a problem change the way engineers actually work. And quite a few papers stop at reporting accuracy or similar scores, without considering how the model fits into day-to-day development or what kinds of demands it places on the team that has to use it. Many evaluations stay within one task—generating code, finding bugs, debugging—without asking how these tools sit alongside the workflows that engineers already follow. Questions about transparency, reproducibility, integration problems, skill development, and the wider organisational effects of using AI show up only now and then. Because of this, even though the methods themselves are often described in detail, there is still no clear picture of how the traditional and AI-based approaches really differ in terms of how they work and what they require in practice. This study addresses that gap by placing the two approaches side by side and examining their respective strengths, limitations, and implications for engineering work.

RESULTS AND DISCUSSION

The studies reviewed here point to a fairly clear divide in how the two traditions operate. Older engineering methods rely on explicit metrics, step-by-step analytical procedures, and models whose behaviour can be explained with little difficulty. AI-based methods often perform better on prediction and generation tasks, largely because they do not depend on a preset list of features and instead pick up patterns directly from the data. They also manage to capture the kinds of semantic links in code that are difficult to describe in advance. Still, the strengths of these models come with their own complications. The practical and organisational issues they create—data demands, opaque behaviour, concerns about deployment—are quite different from the ones people are used to dealing with in more traditional settings.

Traditional defect-prediction work, on the other hand, still leans heavily on structural metrics such as size, coupling, cohesion, cyclomatic complexity, and a set of process-oriented indicators. These measures have been useful for steering testing and quality-assurance efforts, but their limits have become harder to ignore. Several studies point out that two pieces of code can look identical when reduced to metrics even though they behave very differently once executed [1]. Missing increments in loops, memory-management oversights, or subtle exception-handling problems are typical cases. Because these issues do not alter the values of the structural metrics, traditional models treat the faulty and corrected versions as the same. This is also why many standard machine-learning approaches make little progress once they rely only on those metrics, even when researchers try different algorithms or tuning strategies [2]. Figure 1 shows two such examples: the metric profiles match, but the behavioural differences are substantial—and in some cases, a real safety concern.

```
public void copy(Directory to, String src, String dest)
    throws IOException {
      IndexOutput os = to.createOutput(dest);
      IndexInput is = openInput(src);
      IOException priorException = null;

      try {
        is.copyBytes(os, is.length());
      } catch (IOException ioe) {
        priorException = ioe;
      }
      finally {
         IOUtils.closeSafely(priorException, os, is);
      }
    }
```

(a)

```
public void copy(Directory to, String src, String dest)
    throws IOException {
      IndexOutput os = null;
      IndexInput is = null;
      IOException priorException = null;
      try {
        os = to.createOutput(dest);
        is = openInput(src);
        is.copyBytes(os, is.length());
      } catch (IOException ioe) {
        priorException = ioe;
      }
      finally {
         IOUtils.closeSafely(priorException, os, is);
      }
    }
```

(b)

```
public class Buggy
{
    public static void main(String[] args)
    {
        int stop=0;
        int count=0;
        do
        {

         count++;
        } while (stop<20);
        String finish="Stop loop on the "+count;
        System.out.println(finish);
    }
}
```

(c)

```
public class Clean
{
    public static void main(String[] args)
    {
        int stop=0;
        int count=0;
        do
        {
            if (count<=stop)
            {stop++;}
         count++;
        } while (stop<20);
        String finish="Stop loop on the "+count;
        System.out.println(finish);
    }
}
```

(d)

Figure 1. Examples illustrating semantic differences that traditional code metrics cannot detect by Abdu et al [1].

Figure 1 illustrates two pairs of code examples that demonstrate how traditional static code metrics fail to capture meaningful semantic differences between buggy and corrected implementations. In the first pair (a–b), both versions of the copy method have nearly identical structural characteristics—such as lines of code, variable counts, and cyclomatic complexity—yet the buggy version mishandles resources and can produce a memory leak, while the corrected version safely encapsulates stream operations within the try block. The first example shows a pair of code fragments whose meaning diverges even though, at the level of traditional metrics, they look identical. In the second pair (c–d), the problem is different: a loop is supposed to terminate, but the buggy version leaves out the increment, so the loop never ends. Again, a metric-based model sees no difference because the two files produce the same feature vector.

The real difference between the code variants shows up only when they run, not in the metric vectors assigned to them. Traditional metrics simply do not recognise these kinds of semantic shifts, which is why their predictive power tends to flatten out; they can catch broad structural patterns but not the details that actually change program behaviour. To detect those, models built around structural or semantic cues—AST-based methods or various deep-learning approaches—are far better suited.

Data-driven techniques tackle the issue from another angle. Rather than depending on a fixed set of hand-crafted indicators, these models try to build their own representations from the code examples they see. In practice, they often pick up patterns—both structural and behavioural—that are hard to spell out manually. Giray et al. note that convolutional, recurrent, and graph-based models tend to outperform approaches built only on metrics, largely because they can account for the context around a code fragment and the deeper hierarchies that exist inside real programs [6]. Graph neural networks show this quite clearly. Program code already has a sort of graph shape to it, so these models can move through that structure and focus on pieces of the code where faults tend to show up [10]. Pan, Lu, and Xu saw something similar with transformer models like CodeBERT. Those models seem to benefit from dealing with natural-language hints and code patterns together, and that gives them an edge over the earlier approaches [9].

Hybrid approaches add a different point to the discussion. Abdu et al. show that when semantic features derived from ASTs are combined with PROMISE-style statistical descriptors in a CNN–MLP pipeline, the resulting model performs better—across F1, AUC, and several effort-aware measures—than models using only one of the two feature types [1]. The implication is fairly clear: each family of features captures something the other misses, and neither one provides a complete view of software behaviour on its own. They also provide rough estimates of the computational cost involved: building ASTs takes between 0.92 and 3.5 seconds per file, generating Word2Vec embeddings about 0.65–2.3 seconds, and training the joint model roughly 22–35 seconds. These figures suggest that, in practice, hybrid deep-learning systems remain feasible even for fairly large projects (Figure 2).

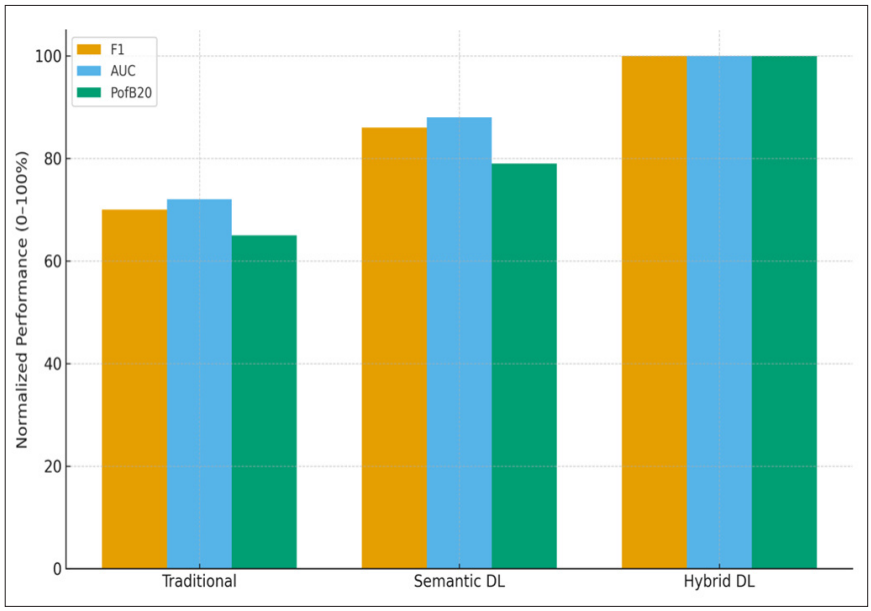

Figure 2. Comparative performance of traditional, semantic, and hybrid models

Figure 2 presents a normalized comparison of traditional, semantic deep-learning, and hybrid defect-prediction models across three commonly reported metrics: F1, AUC, and effort-aware PofB20 (an effort-aware evaluation metric widely used in software defect prediction research). Because the reviewed studies use heterogeneous datasets, experimental setups, and reporting conventions, exact numerical values cannot be combined directly. The bar heights are not meant to represent precise values. They come from a simple scaling of the performance ranges reported in several studies (Abdu et al.; Albattah & Alzahrani; Batool & Khan; Giray et al.). Looked at this way, the older metric-based models end up near the lower end of the scale — their results tend to be steady but the improvements they offer are fairly modest. The semantic deep-learning models sit somewhere in the middle of the scale. They usually show clearer gains than the older metric-based ones, although the results aren't completely uniform across studies. The hybrid models — the ones that mix engineered features with those learned from data — are used as the upper reference point and set to 100%. The intention behind this is simply to show the general trend that keeps appearing in the literature. It isn't meant to suggest that the numerical distance between the groups is precise or fixed in any strict sense.

AI tools now appear in parts of the engineering workflow that previously were not considered especially suitable for automation. Large language models, for instance, have been tested on a wide range of tasks: interpreting or rewriting requirements, generating tests, helping with parts of design, locating bugs, and supporting maintenance work. Hou et al. list more than fifty different uses of these models, ranging from pulling pieces out of formal specifications to fixing broken code and checking for security problems [8]. Part of their appeal is that the same model can work with natural language and with source code, without needing separate systems. But the results they produce still depend a lot on the datasets used for tuning or prompting. If the data are uneven, or if some cases are poorly represented, the outputs can shift in ways that aren't easy to predict. Table 1 gives a sense of how these issues show up in practice.

Table 1. Coverage of traditional vs. AI-mediated methods across the software engineering lifecycle

Lifecycle Stage	Traditional Methods	AI-Mediated Methods
Requirements Engineering	• Manual stakeholder interviews • Text-based requirement documents • Rule-based extraction	• LLM-based requirement summarization and rewriting • Automated extraction of functional/non-functional requirements • AI-generated traceability links
Design	• UML diagrams and architecture models • Expert-driven design reviews	• AI-assisted architecture suggestions • Pattern detection in design artefacts • Automated consistency and constraint checks
Coding	• Manual code implementation • Basic IDE autocomplete • Linters and static rule checks	• LLM-based code generation (functions/classes) • Semantic code completion (Copilot-like systems) • Automated refactoring and style harmonization
Testing	• Manually written test suites • Static/dynamic analysis tools • Manual coverage evaluation	• Automated test case generation • Bug prediction models (CNN, LSTM, GNN) • AI-driven fuzzing and vulnerability detection

Deployment	• Manual CI/CD configuration • Script-based rollouts	• AI-optimized pipeline configuration • Automated detection of deployment risks • Predictive rollout optimisation
Maintenance	• Manual debugging and patching • Log inspection • Human triage of issues	• AI-suggested code repair • Predictive maintenance & anomaly detection • LLM-based debugging guidance

Table 1 shows where the traditional techniques and the AI-based tools emerge in the various parts of the software-engineering process. The point is not just to list the tasks but to give a sense of how the reach of AI has been spreading beyond its usual role in defect prediction. The traditional techniques still lean heavily on manual checking and on artefacts or rules that people prepare themselves; these are familiar and easy to follow, but they take time and rely on fairly specific expertise. The AI-oriented methods deal with the same stages in a different way, adding automation and a layer of semantic interpretation that the older tools didn't have. Large language models, for example, are now being used to extract requirements or restate them when needed, and several deep-learning systems are being applied to automated testing and bug-prediction tasks. Some of the generative tools have begun showing up in coding and debugging tasks as well, and in a few cases even in parts of maintenance. In the table, the stages are separated to show where AI mostly works alongside existing practice — design and deployment tend to fall in that group — and where it begins to take over more routine pieces of the job, like low-level coding or test generation. Looking at the lifecycle this way helps make it clearer how AI affects not only model accuracy but also how engineering work is divided and organised.

At the practice level, AI-mediated engineering approaches deliver measurable productivity and quality benefits. Survey data from Alenezi and Akour show that 74% of practitioners report reduced coding time, 62% observe fewer post-release defects, and 45% note improved team resource allocation after adopting AI tools such as Copilot, IntelliCode, or AI-driven test automation [3] . Some tool-specific studies point to quite substantial improvements. For example, Snyk Code reports noticeably better early bug detection—around a 40% gain—and a marked reduction in manual review time, roughly by half. AlphaCode, working in a very different context, reaches a success rate of about 65% on competitive programming tasks and shortens prototyping cycles by close to 40% [10]. These kinds of jumps are not what traditional metric-based approaches typically deliver, which tend to improve performance only gradually.

The move toward AI-mediated techniques, however, brings its own set of complications. Giray et al. note that nearly half of the deep-learning studies they reviewed (48%) do not provide any reproducibility package, and this situation has not improved over time [6]. Traditional methods do not face this problem to the same degree because their assumptions and evaluation steps are usually straightforward to reconstruct. Additional difficulties emerge along the lifecycle: inconsistent data formats, skewed class distributions, a tendency toward overfitting, and practical issues when deploying or maintaining models. These are considerably more involved than the challenges posed by older analytical models [6].

When these tools are introduced in real engineering teams, a different layer of concerns appears. Only a minority of practitioners—about 40%—express full confidence in AI-generated outputs, and the reasons they give are mostly related to opacity and uncertainty about potential errors [3]. The survey numbers also point to practical issues. Many respondents mention skill gaps (58%) and the cost of getting these tools running (52%), both of which slow adoption. Banh et al. note that generative systems are already shifting how developers spend their time: instead of writing code from scratch, more of the work goes into reviewing and steering the system's suggestions. This raises questions about how skills evolve over time, how much teams end up depending on automated tools, and whether these new routines fit well with the workflows that organisations already use [4].

Another difficulty has to do with how scattered the current set of AI tools still is. Alenezi and Akour note that even though many of these tools do well on specific tasks—generation, prediction, testing—most of them work in isolation and do not pass information from one stage of the lifecycle to another [3]. By contrast, traditional approaches are usually part of larger frameworks that connect design, testing, and maintenance, which makes the overall workflow easier to keep together. Without some broader framework tying these tools together, whatever improvements they make tend to stay stuck in the single step they were built for. They do what they are meant to do there, but very little carries forward into the rest of the workflow.

Because of this, the difference between the older approaches and the AI-oriented ones is less straightforward than it seems at first. The traditional methods still give engineers processes they can check and follow, and they usually sit comfortably within the lifecycle structures teams already use. The AI models contribute something else entirely: they can reach higher accuracy, they pick up semantic or contextual cues that the older metrics ignore, and they often shorten pieces of work that normally take longer. But the trade-offs are real. These systems depend heavily on data, on whether an organisation can support them, and on people keeping a close eye on how they behave. For now, the setups that mix the two worlds — using engineered indicators alongside

features the model learns on its own — seem to be the most practical, adding to what the traditional techniques already do instead of replacing them.

CONCLUSION

Developers have not so much moved from one clear set of engineering methods to another as worked their way through a slow and uneven shift. Older techniques, the ones built around manually defined metrics and long-familiar analytical steps, are still used in many places simply because people know exactly what goes into them. Teams are also familiar with how these older tools tend to behave, which makes them easy to work with, but their shortcomings become noticeable once they are used on newer systems. Many of these systems operate in ways that cannot be captured by a small set of structural metrics, and the gap between what the tools measure and what the system actually does has only widened as software has grown in size and complexity.

More recent AI-based approaches come at the problem from a different direction. Rather than working with a fixed set of indicators, they try to learn patterns straight from the code and from the artefacts surrounding it. A lot of these patterns are difficult to spell out in fixed rules, which is part of the reason deep models and language-based systems often do better on prediction tasks. They can also help with routine work, such as suggesting where a problem might sit or narrowing down the part of the code that needs attention. But the advantages come with their own complications. A model's behaviour can shift quite a bit depending on the data it has seen, and it is often not obvious how it arrived at a particular output. Teams also run into the simple problem of trying to fit these tools into workflows that were set up long before anyone imagined using systems like this. On top of that, a number of studies mention trouble repeating results, especially when the information about how the model was trained or which datasets were involved is incomplete or only briefly described.

When the different results from the studies are put side by side, the picture that emerges is not one where a single approach clearly wins. The older methods still give engineers something steady to work from, and many rely on that familiarity when making decisions. The newer AI tools contribute something quite different — they notice patterns and contextual signals that the older metric sets never really captured. What seems to work best for now is not choosing one over the other but combining them in a way that lets each cover the gaps of the other. When engineered features are used along with what the models learn from data, the two parts tend to support each other rather than compete. It is less a matter of replacing one tradition with another and more of finding a mix that people can actually use in everyday practice.

Several directions for future research are becoming increasingly clear, although none of them have been explored in a consistent way. One of the more immediate needs is to understand how the various AI tools now used in software engineering can be linked so that information generated at one stage is not lost by the next. At the moment, code generators, defect predictors, and testing systems are typically introduced as separate additions to existing workflows, which means they rarely benefit from one another's outputs. Reproducibility remains a persistent point of friction in the literature. Quite a few deep learning papers show strong numerical results, but the data or model setups behind them are not always available in a form that others can actually use. Sometimes the dataset versions differ from what is described; in other cases, the training details are scattered across supplementary files or missing altogether. Without more stable conventions for sharing these materials, it becomes difficult to judge how much progress is being made or to repeat earlier experiments with any confidence.

Concerns about transparency continue to surface in practice, and they tend to slow down adoption more than the performance numbers would suggest. In settings where safety rules or compliance checks are tight, engineers often hesitate to bring in models whose reasoning cannot be inspected in a straightforward way. This hesitation is not surprising, since the consequences of relying on an opaque system can be significant. Currently, it is still unclear which interpretability techniques can actually be trusted in everyday engineering practice. The longer-term impacts of working with AI tools have received little systematic attention, and much of the available evidence is anecdotal or scattered. Looking more closely at these areas could help reveal how AI might contribute to a more continuous and integrated development process..

REFERENCES

1. Abdu, A., Zhai, Z., Abdo, H. A., Algabri, R., Al-masni, M. A., Muhammad, M. S., & Gu, Y. H. (2024). Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model. *Scientific Reports*, *14*(14771), 1-20. https://doi.org/10.1038/s41598-024-65639-4
2. Albattah, W., & Alzahrani, M. (2024). Software defect prediction based on machine learning and deep learning techniques: An empirical approach. *AI*, *5*(4), 1743–1758. https://doi.org/10.3390/ai5040086
3. Alenezi, M., & Akour, M. (2025). AI-driven innovations in software engineering: A review of tools and practices. *Applied Sciences*, *15*(3), 1-26. https://doi.org/10.3390/app15031344
4. Banh, L., Holldack, F., & Strobel, G. (2025). Copiloting the future: How generative AI transforms software engineering. *Information and Software Technology*, *183*(107751), 1-14. https://doi.org/10.1016/j.infsof.2025.107751

5. Batool, I., & Khan, T. A. (2023). Software fault prediction using deep learning techniques. *Software Quality Journal*, *31*(4), 1241–1280. https://doi.org/10.1007/s11219-023-09642-4

6. Giray, G., Bennin, K. E., Köksal, Ö., Babur, Ö., & Tekinerdogan, B. (2023). On the use of deep learning in software defect prediction. *Journal of Systems and Software*, *195*(111537). 1-26. https://doi.org/10.1016/j.jss.2022.111537

7. Hasan, A. T., & Mohi-Aldeen, S. M. (2024). Software Defect Prediction Based on Deep Learning Algorithms: A Systematic Literature Review. *Al-Rafidain Journal of Computer Sciences and Mathematics (RJCSM)*, *19*(1), 67-79. http://doi.org/10.33899/csmj.2025.156086.1160

8. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, *33*(8), 1–79. https://doi.org/10.1145/3695988

9. Pan, C., Lu, M., & Xu, B. (2021). An empirical study on software defect prediction using CodeBERT model. *Applied Sciences*, *11*(11), 4793. https://doi.org/10.3390/app11114793

10. Sikic, L., Kurdija, A. S., Vladimir, K., & Silic, M. (2022). Graph neural network for source code defect prediction. *IEEE Access*, *10*, 10402–10415. https://doi.org/10.1109/ACCESS.2022.3144598