



Journal of Interdisciplinary Science & Technology

Explainable Artificial Intelligence (XAI) Techniques for the Software Development Lifecycle: A Phase-Specific Framework

Prof. Kayyum Shaikh¹, Prof. Anmol Budhewar¹, Yash Pable¹, Payal Attarde¹, Samrudh Badgujar¹, Chetan Desale¹

¹Department of Computer Engineering, Sandip Institute of Technology and Research Centre, Nashik, Maharashtra, India
kayyum.shaikh22@gmail.com, anmolsbudhewar@gmail.com, yashpable.nsk008@gmail.com, attardepayal@gmail.com
samrudhabadgujar007@gmail.com, desalechetan9997@gmail.com

Corresponding author: Prof. Kayyum Shaikh

DOI: 10.5281/zenodo.23299714

Abstract: Artificial Intelligence (AI) is rapidly being integrated into the Software Development Lifecycle (SDLC) to automate tasks ranging from defect prediction to requirements analysis. However, the “black-box” nature of these predictive models creates a significant barrier to trust, as developers cannot easily verify the reasoning behind AI-generated outputs. Current research disproportionately focuses on the maintenance phase, leaving practitioners without structured guidance for earlier lifecycle stages. To address this, we conducted a systematic literature review following PRISMA guidelines, detailing the rigorous search, filtering, and selection of current XAI-in-SE literature. We critically compare established foundational methodologies—including LIME, SHAP, and Counterfactuals—against recent advancements in inherent LLM interpretability and multi-agent orchestration. Building upon this analysis, we propose a unified, phase-specific conceptual framework that maps these techniques to the SDLC based on stakeholder technical proficiency. While the proposed architecture remains conceptual, it provides a standardized theoretical blueprint for designing transparent, auditable, and trustworthy AI-assisted software engineering pipelines.

Keywords: Explainable AI (XAI), Software Development Lifecycle (SDLC), PRISMA, Systematic Review, Algorithmic Transparency, SHAP, LIME

I. INTRODUCTION

The automation of software engineering tasks has accelerated rapidly with the integration of machine learning and large language models (LLMs). Today, complex, non-linear models drive applications from natural language requirement parsing to automated defect localization. However, the underlying technologies—like deep neural networks, graph neural networks (GNNs), and transformers—operate using internal logic that remains completely hidden from the end user [7], [18], [24]. This creates a serious practical problem. Developers struggle to debug poor recommendations, while project managers find it nearly impossible to audit these systems for fairness or bias [10], [29]. When looking at the financial and safety risks, this lack of transparency is alarming. If we consider mission-critical software—like aerospace or healthcare applications—even one unexplainable AI error during testing can eventually trigger massive failures in production.

Explainable Artificial Intelligence (XAI) has emerged to unbox these models, allowing humans to verify the reasoning

behind specific algorithmic outputs [11], [21], [33]. However, interpretability is highly contextual. An explanation that successfully clarifies a bug-prediction model for a tester might prove entirely useless to a business analyst trying to understand why a requirement was classified as ambiguous [6], [25].

This paper addresses the critical need for contextual explainability by systematically reviewing the literature and mapping primary XAI modalities directly to specific SDLC phases. Our core contributions are as follows:

- We conduct a systematic literature review following PRISMA guidelines, highlighting how current XAI research is heavily skewed toward specific phases of the software lifecycle.
- We build a taxonomy of existing XAI techniques, focusing on where they succeed and where they computationally break down when analyzing code.
- We design a conceptual framework that maps different XAI tools to specific SDLC phases, matching the technique to the technical skill level of the user.

- We outline a modular architecture to put this mapping into practice, illustrating how it works through three specific scenario walkthroughs.

Table 1: Systematic comparison of key surveyed literature

Reference	Year	Core SDLC Focus	Key Contribution	Noted Limitations
Ribeiro et al. [8]	2016	Testing	Introduced LIME for model-agnostic local surrogate explanations.	Vulnerable to instability across repeated runs.
Lundberg & Lee [9]	2017	Maintenance	Unified game-theoretic feature attribution via SHAP.	Computationally heavy for highly dimensional datasets.
Mothilal et al. [4]	2020	Deployment	Proposed diverse counterfactual explanations for actionable insights.	Search algorithms for feasible instances are resource-intensive.
Chazette et al. [6]	2022	Requirements	Argued XAI must be treated as a core non-functional software requirement.	Framework remains conceptual; lacks tooling details.
Fan et al. [15]	2023	Broad SDLC	Highlighted the interpretability challenges of applying LLMs across the SDLC.	High implementation overhead for legacy systems.
Hou et al. [16]	2024	Development	Systematized the shift from post-hoc methods to inherent LLM interpretability.	Restricted to neuro-symbolic and prompt-based code generation.
Zhang et al. [17]	2024	Testing	Established XAI transparency benchmarks for autonomous testing agents.	Multi-agent explanation tracking remains highly complex.

II. SYSTEMATIC REVIEW METHODOLOGY (PRISMA)

To ensure a rigorous, reproducible, and unbiased analysis of the intersection between XAI and software engineering, this study adopts the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology. The paper selection process progressed through four distinct phases: Identification, Screening, Eligibility, and Inclusion.

A. Identification

A comprehensive search was executed across four primary academic databases: IEEE Xplore, ACM Digital Library, SpringerLink, and ScienceDirect. The Boolean search string was constructed to capture the intersection of interpretability and software development: (“Explainable AI” OR “XAI” OR “Interpretability”) AND (“Software Engineering” OR “SDLC” OR “Software Maintenance” OR “Code Generation”). The initial search, spanning literature published between 2016 and 2026, yielded 412 preliminary records.

B. Screening and Eligibility

After the removal of 87 duplicate records, 325 papers underwent title and abstract screening. During this phase, papers were filtered based on strict inclusion criteria: the study had to directly apply a mathematical explainability technique to a specific software engineering task. Records focusing purely on AI model performance without an explainability component, or XAI papers applied to non-engineering

domains (e.g., medical imaging), were excluded. This screening eliminated 240 papers.

The remaining 85 papers underwent full-text eligibility assessment. The exclusion criteria at this stage removed papers lacking empirical validation, short workshop abstracts, and studies that did not clearly define the SDLC phase being targeted. This resulted in the exclusion of an additional 40 papers.

C. Quality Assessment and Data Extraction

We put the final 45 papers through a strict quality check. Our evaluation looked at three main things: whether the explainability setup could be reproduced, if the researchers actually tested their explanations on human developers, and how robust the core AI model was. From these studies, we pulled out specific details, including the exact machine learning models used, the XAI approaches applied, the target SDLC phase, the types of artifacts analyzed (like code or text), and any performance overhead they reported.

D. Inclusion and Quantitative Synthesis

Ultimately, 45 core papers were deemed eligible for deep synthesis. While this review heavily integrates recent advancements (2024–2025) to reflect the current state-of-the-art, foundational references (2016–2020) detailing the original mathematical proofs for LIME [8], SHAP [9], and

Counterfactuals [4] were deliberately retained as necessary theoretical anchors.

A quantitative analysis of the 45 included papers revealed a severe phase imbalance: 68% ($n = 31$) focused exclusively on the Maintenance and Evolution phase (primarily defect prediction and code-review automation). The Testing phase accounted for 15% ($n = 7$), and Deployment/Monitoring accounted for 9% ($n = 4$). Crucially, the early stages of the SDLC—Requirement Elicitation and Design—represented only 8% ($n = 3$) of the literature combined. This quantitative disparity empirically validates the necessity of the phase-spanning framework proposed in this paper.

III. STATE OF EXPLAINABLE SOFTWARE ENGINEERING

The trajectory of XAI has evolved from early intrinsic models (like decision trees) [14] to post-hoc, model-agnostic tools like LIME and SHAP. The software engineering domain is currently entering a third wave, characterized by the shift from isolated algorithmic demonstrations toward actual operational integration [2], [19], [26].

A. Taxonomy of XAI in Software Engineering

Before matching XAI methods to the software lifecycle, we need to clarify the types of explainability available for software artifacts.

- **Global versus Local Explainability:** Global methods try to explain the model's overall behavior across every possible input. Local methods look at just one specific decision at a time. Because software development moves quickly, developers generally rely on local explainability to debug specific issues right away.
- **Intrinsic versus Post-Hoc Interpretability:** Intrinsic models are built to be transparent from the start. But they usually struggle to handle complicated data structures like abstract syntax trees. Post-hoc tools work differently. They sit on top of complex black-box models and try to reverse-engineer how a decision was made after the fact.

B. Critical Comparison of Recent Literature (2023–2025)

While foundational papers successfully established post-hoc interpretability methods [8], [9], recent literature demonstrates a critical paradigm shift toward inherent interpretability and system-wide orchestration [20], [27].

For instance, the extensive survey by Fan et al. (2023) [15] criticizes the isolated application of SHAP and LIME, arguing that explainability must be elevated to a core non-functional requirement managed system-wide. This sharply contrasts with earlier, model-centric views. Expanding on this, Hou et al. (2024) [16] critically compare post-hoc methods against the inherent interpretability of modern LLMs in software engineering. They found that in the Development phase,

forcing post-hoc approximations onto code-generation LLMs introduces unacceptable latency, advocating instead for natively transparent prompting designs.

Most recently, literature synthesized by Zhang et al. [17] addresses the rise of autonomous, multi-agent SDLC orchestration. Their work highlights a critical gap in traditional XAI: while methods like LIME effectively explain single-node predictions, they struggle to explain the collaborative decision-making of AI agents spanning across requirements and testing concurrently. Maintaining transparency in these modern architectures requires dynamic, phase-aware routing—a finding that directly validates the conceptual architecture proposed in Section V of this paper.

IV. CRITICAL ANALYSIS OF XAI MODALITIES

A. Perturbation and Surrogate Models (Lime)

LIME isolates a single prediction and generates a localized dataset by perturbing the original input [8]. When analyzing a piece of code, LIME essentially hides different parts of it—like specific variables or operators—and watches how the AI's prediction shifts. It then builds a simple, easy-to-read model (like a linear regression) to explain that specific prediction [30]. LIME is fast and works with almost any model, which is great for interactive coding tools. On the downside, because it uses random sampling, the explanations can be unstable. If a developer runs LIME twice on the identical piece of code, they might get slightly different results, which quickly damages trust. Additionally, LIME relies on the assumption that the local decision logic is linear. In modern programming languages, code dependencies are highly non-linear, meaning LIME sometimes misses the real context.

B. Cooperative Game Theory (Shap)

SHAP relies on coalition game theory to distribute a model's output value among its input features [9]. SHAP measures how much each feature—whether that's a code metric, an AST node, or a single word—contributes to the final output by testing all possible combinations of those features. The major benefit here is mathematical certainty; SHAP provides rigid theoretical guarantees, specifically local accuracy, missingness, and consistency [5]. The downside is that it demands a lot of computing power [32]. Fortunately, optimized variants like TreeSHAP make it much easier to use on large codebases. A key advantage over LIME is that SHAP is deterministic, meaning developers get the exact same feedback every time they run it.

C. Counterfactual and Example-Based Explanations

Counterfactuals identify the minimal perturbation required to alter a model's decision [4]. Rather than showing abstract graphs that only point out what went wrong, a counterfactual gives a direct, actionable alternative that shows how to fix the

problem. For instance, the system might say: ‘If you lower the cyclomatic complexity by two and bump up test coverage by 5%, the code will pass the AI quality check.’ This is highly practical [23]. Finding these counterfactuals involves running optimization algorithms to locate the nearest acceptable solution. While this takes extra computing power, it provides a much better experience for the developer.

D. Rule and Concept Extraction

Rule extraction translates neural weights into deterministic Boolean logic [12]. While fidelity degrades in highly complex spaces, IF-THEN rules are perfectly suited for non-technical stakeholders who need clear, deterministic answers. Concept-based explanations go a step further by mapping basic features

(like syntax elements) to broader domain concepts, such as module coupling or class cohesion. This fits perfectly with high-level architectural planning [11], [31].

V. SYNTHESIZING THE PHASE-SPECIFIC FRAMEWORK

As visually summarized in Figure 1 and detailed in Table 2, the core contribution of this paper is the precise mapping of XAI modalities to SDLC phases. To optimize narrative flow and avoid redundancy with Table 2, this section focuses exclusively on the theoretical justification—specifically, why alternative XAI methods structurally fail in these specific engineering contexts.

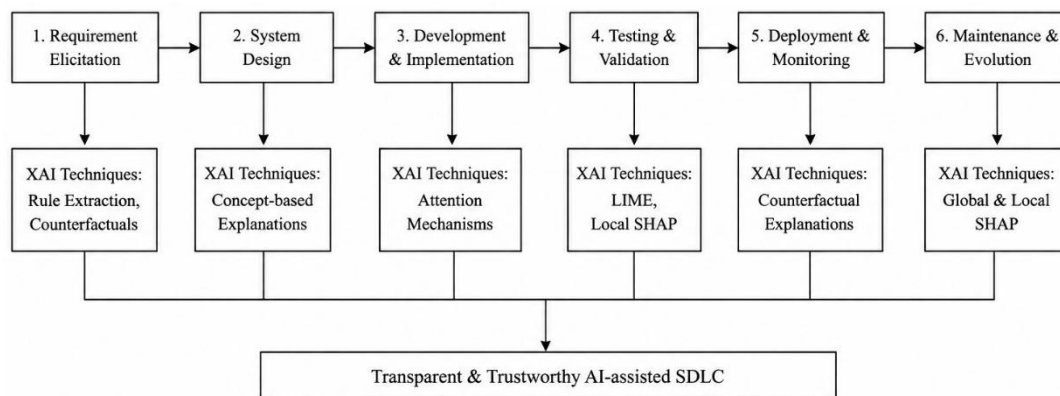


Figure 1: Phase-Specific XAI Framework for the SDLC

A. Requirement Elicitation (Rule Extraction)

In software projects today, NLP models process thousands of user stories to spot vague language or conflicting ideas. The people reading these alerts are usually business analysts and product owners who don’t have statistical backgrounds. **Why alternatives fail:** Tools like SHAP generate complex numerical grids and charts. An analyst gets no real value from a Shapley score saying the word ‘fast’ has a +0.4 impact on ambiguity. They need straightforward logic. Rule Extraction (for example, ‘IF vague quantifier THEN flag’) is absolutely necessary here [18], [30]. It turns AI math into the kind of business logic analysts deal with every day.

B. System Design (Concept-Based Explanations)

During system design, predictive models estimate how maintainable or secure a proposed architecture will be. **Why alternatives fail:** Local explainers like LIME tweak raw syntax or basic log data. But software architects don’t look at single tokens; they care about big-picture design patterns. Concept-based explanations work well here because they frame the AI’s logic around known architectural metrics, like module coupling or inheritance depth, rather than raw numbers.

C. Development and Implementation (Attention Mechanisms)

Modern coding environments rely heavily on LLMs like GitHub Copilot to auto-complete code. **Why alternatives fail:** Studies show [15] that trying to compute explanations after the fact using SHAP or LIME takes too long. A programmer writing code in real time cannot pause for five seconds to wait for a graph to load. Native Attention Mechanisms solve this by projecting visual heatmaps directly over the code as the model processes it [28]. This happens instantly, letting the developer see exactly which surrounding lines influenced the AI’s suggestion without breaking their flow.

D. Testing and Validation (Lime And Shap)

AI tools now predict where code is likely to fail, helping teams decide which unit tests to prioritize. **Why alternatives fail:** Simple rule extraction isn’t detailed enough to debug messy, complex scripts. QA engineers need the exact, quantitative scores that LIME and SHAP provide so they can link failures to specific variable changes or recent code churn [13], [24]. SHAP, for instance, lets a tester see exactly which file edit caused the risk score to jump.

E. Deployment and Monitoring (Counterfactual Explanations)

Anomaly detection systems watch over production environments by analyzing thousands of incoming telemetry streams. **Why alternatives fail:** When a live server goes down, a Site Reliability Engineer (SRE) is under severe pressure. They don't have time to study a dense feature-importance chart to guess what broke. They need immediate direction. Counterfactual explanations skip the abstract details

and suggest the exact rollback required to fix the issue (e.g., 'If memory allocation is raised to 4GB, the system will stabilize').

F. Maintenance and Evolution (Global Shap)

Estimating technical debt automatically helps ensure a codebase remains viable long-term. **Why alternatives fail:** Because LIME is mathematically unstable and gives slightly different answers each time, it is terrible for tracking trends over long periods. SHAP uses solid game theory, making it the only reliable choice for measuring how a repository decays over years and thousands of commits.

Table 2: SDLC phase-to-technique mapping architecture

SDLC Phase	Typical AI Implementation	Target Stakeholder	Assigned XAI Modality
Requirement Elicitation	NLP Ambiguity & Classification Models	Business Analysts, Clients	Rule Extraction, Counterfactuals
Architectural Design	Smell Detection & Maintainability Estimators	Software Architects	Concept-based Explanations
Development	Real-time Code Completion LLMs	Software Developers	Native Attention Mechanisms
Testing	Test-Failure & Defect Predictors	QA Engineers, Testers	LIME, Local SHAP
Deployment & Monitoring	Telemetry Anomaly Detection	Site Reliability Engineers	Counterfactual Explanations
Maintenance & Evolution	Technical Debt & Code Review Automation	Maintainers, Tech Leads	Global and Local SHAP

VI. PROPOSED CONCEPTUAL ARCHITECTURE

To ensure this theoretical framework can be systematically adopted, we propose a modular, conceptual pipeline architecture designed to decouple the predictive model from the explanation generation process.

A. Architectural Layers

The proposed system executes across four distinct conceptual layers:

- 1) **Input Layer:** Standardizes various SDLC artifacts—like text requirements, source code, or server logs—into the specific numerical formats the predictive models expect.
- 2) **Inference Layer:** Holds the main AI model (or models) that makes the actual prediction. This layer operates entirely as a black box, optimized purely for speed and accuracy.
- 3) **Routing and Explainability Layer:** This is the core control center of our framework. It looks at which SDLC phase the data came from (using metadata from the Input Layer) and dynamically sends the model's output to the correct mathematical explainer. This ensures that a business analyst is never confused by a LIME graph, and a tester isn't handed a vague concept abstraction.
- 4) **Presentation Layer:** Converts the raw mathematical data from the Routing Layer into a clear, visual format built specifically for the end user.

B. Case walkthrough A: Requirement Elicitation

Imagine the early stages of a project where an NLP model checks a new requirement ticket (e.g., 'The system shall process financial data quickly'). The Input Layer takes in the text, and the Inference Layer tags it with an 85% chance of being 'Ambiguous.' The Routing Layer sees that this is a non-technical context based on the ticket metadata. It actively ignores complex explainers like SHAP and triggers the Rule Extraction module instead. The Presentation Layer then prints a simple message in the ticket: "Flagged: IF requirement contains unquantified adjective ('quickly') THEN Ambiguous." The analyst immediately knows they need to define an exact latency number, showing how this setup supports non-technical users perfectly.

C. Case walkthrough B: Defect Prediction

Now consider the Maintenance Phase. A CI/CD pipeline runs an AI classifier to see if a new code commit might cause a bug. The Input Layer pulls features like cyclomatic complexity and the developer's historical commit data. The Inference Layer gives a 78% defect probability, causing the build to fail. Because this is an engineering context, the Routing Layer sends the data to the SHAP module. The Presentation Layer displays an interactive waterfall chart directly in the Pull Request. It shows the developer that "high cyclomatic complexity" was the main reason the code got flagged, giving them the exact insight needed to refactor the function.

D. Case walkthrough C: Incident Response

Finally, look at the Deployment phase. An anomaly detector watching Kubernetes pods notices a sudden latency spike in a microservice. The Inference Layer predicts the service will crash soon. Recognizing this as a high-stress deployment situation, the Routing Layer passes the data to the Counterfactual Explanation engine. Instead of drawing a graph to show that CPU spikes are to blame, the Presentation Layer gives the SRE a direct command on their dashboard: “Action Required: Reverting the recent configuration flag for batch-processing [False → True] will reduce crash probability from 92% to 14%.” The SRE can execute the rollback immediately to keep the system online.

VII. LIMITATIONS, ETHICS, AND VALIDATION STRATEGY

A. Conceptual Framework Limitations

A major limitation of this work is that the phase-specific routing framework is still a conceptual architecture. We have not yet validated it by deploying a live software prototype in a production environment. The processing power needed to dynamically route and calculate explanations (especially SHAP and Counterfactuals) inside real-time CI/CD pipelines is a real engineering challenge that needs more optimization work.

B. Proposed Empirical Validation Protocol

To move this framework from concept to an industry standard, researchers must run controlled tests with real software-engineering teams. We suggest a validation setup where developers are divided into control groups (using standard black-box AI tools) and experimental groups (using our XAI-routed architecture). Success should be measured with hard metrics, like how fast bugs are resolved and how quickly failed builds recover. At the same time, we need to survey developers to see if they actually trust the explanations.

C. Epistemological and Ethical Considerations

It is also important to remember the inherent limits of post-hoc explainability. As Rudin pointed out [14], post-hoc explanations are ultimately just approximations [29]. They explain the model’s logic, but not necessarily the real-world truth. In critical systems, it is always better to use models that are interpretable by design. Also, if a defect-prediction model picks up biases from old training data (for instance, unfairly flagging code from junior developers), the XAI tools will accurately explain why the model made the biased choice, but they will not fix the bias itself. XAI acts as a diagnostic window, not a cure [22], [33].

D. Regulatory Compliance and Automation Bias

Bringing AI into the SDLC intersects with new legal rules, like the European Union’s AI Act and the GDPR’s “Right to Explanation.” Implementing our proposed architecture helps software vendors keep clear logs of why their internal AI systems made specific engineering choices, which is crucial for compliance. However, teams must watch out for “automation bias”—the tendency for developers to blindly trust AI explanations just because the math looks rigorous, which can lead to sloppy manual code reviews.

VIII. CONCLUSION

We have shown that a single, one-size-fits-all approach to explainability simply does not work for the diverse tasks within the Software Development Lifecycle. Through our PRISMA-guided review, we measured the extreme phase imbalance in today’s research, exposing how upstream lifecycle phases are largely ignored. We compared older, foundational techniques against newer LLM interpretability methods, highlighting where they succeed and where they computationally break down. By mapping the mathematical realities of XAI techniques—from LIME and SHAP to rule extraction—to the specific needs of different stakeholders, we offer a realistic blueprint for transparent AI integration. Although our dynamic-routing architecture is currently conceptual, it sets the theoretical groundwork necessary for organizations to build predictive tools that are fully auditable, highly practical, and aligned with human reasoning. This ultimately supports a more accountable future for modern software engineering.

IX. REFERENCES

- [1] L. Arora, A. Raj, S. S. Girija, D. Pradhan, S. Kapoor, and A. Shetgaonkar, “Explainable Artificial Intelligence Techniques for Software Development Lifecycle: A Phase-specific Survey,” 2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC), 2025, doi: 10.1109/COMPSAC65507.2025.00321.
- [2] A. H. Mohammadkhani et al., “A Systematic Literature Review of Explainable AI for Software Engineering,” arXiv.org, Feb. 2023, doi: 10.48550/arxiv.2302.06065.
- [3] G. Vilone and L. Longo, “Classification of Explainable Artificial Intelligence Methods through Their Output Formats,” Machine Learning and Knowledge Extraction, vol. 3, no. 3, pp. 615-661, Aug. 2021, doi: 10.3390/make3030032.
- [4] R. K. Mothilal et al., “Explaining Machine Learning Classifiers through Diverse Counterfactual Explanations,” Proceedings of the 2020 Conference on Fairness,

Accountability, and Transparency, ACM, 2020, pp. 607-617, doi: 10.1145/3351095.3372850.

[5] A. Messalas et al., “Model-Agnostic Interpretability with Shapley Values,” 2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA), IEEE, 2019, pp. 1-7, doi: 10.1109/IISA.2019.8900669.

[6] L. Chazette et al., “Explainable software systems: from requirements analysis to system evaluation,” Requirements Engineering, 2022, doi: 10.1007/s00766-022-00393-5.

[7] S. Martinez-Fernandez et al., “Software Engineering for AI-Based Systems: A Survey,” ACM Transactions on Software Engineering and Methodology, vol. 3, no. 2, pp. 1-59, Apr. 2022, doi: 10.1145/3487043.

[8] M. T. Ribeiro, S. Singh, and C. Guestrin, “‘Why Should I Trust You?’: Explaining the Predictions of Any Classifier,” Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, doi: 10.1145/2939672.2939778.

[9] S. M. Lundberg and S.-I. Lee, “A Unified Approach to Interpreting Model Predictions,” Advances in Neural Information Processing Systems (NeurIPS), vol. 30, 2017.

[10] A. Adadi and M. Berrada, “Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI),” IEEE Access, vol. 6, pp. 52138-52160, 2018, doi: 10.1109/ACCESS.2018.2870052.

[11] F. K. Dosilovic, M. Brcic, and N. Hlupic, “Explainable Artificial Intelligence: A Survey,” 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, doi: 10.23919/MIPRO.2018.8400040.

[12] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, “A Survey of Methods for Explaining Black Box Models,” ACM Computing Surveys, vol. 51, no. 5, pp. 1-42, 2019, doi: 10.1145/3236009.

[13] S. Wattanakriengkrai et al., “Predicting Defective Lines Using a Model-Agnostic Technique,” IEEE Transactions on Software Engineering, vol. 48, no. 5, pp. 1480-1496, 2022, doi: 10.1109/TSE.2020.3023177.

[14] C. Rudin, “Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead,” Nature Machine Intelligence, vol. 1, pp. 206-215, 2019, doi: 10.1038/s42256-019-0048-x.

[15] A. Fan et al., “Large Language Models for Software Engineering: Survey and Open Problems,” 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023.

[16] X. Hou et al., “Large Language Models for Software Engineering: A Systematic Literature Review,” ACM Transactions on Software Engineering and Methodology, 2024.

[17] J. Zhang, M. Harman, L. Ma, and Y. Liu, “Machine Learning Testing: Survey, Landscapes and Horizons,” IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 1-36, 2022.

[18] A. Barredo Arrieta et al., “Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI,” Information Fusion, vol. 58, pp. 82-115, 2020.

[19] T. Miller, “Explanation in artificial intelligence: Insights from the social sciences,” Artificial Intelligence, vol. 267, pp. 1-38, 2019.

[20] A. Holzinger, “From Machine Learning to Explainable AI,” 2018 World Symposium on Digital Intelligence for Systems and Machines (DISA), 2018.

[21] D. Gunning et al., “XAI—Explainable artificial intelligence,” Science Robotics, vol. 4, no. 37, 2019.

[22] J. Ji et al., “AI Alignment: A Comprehensive Survey,” arXiv preprint arXiv:2310.19852, 2023.

[23] C. J. Cai et al., “The effects of example-based explanations in a machine learning interface,” Proceedings of the 24th International Conference on Intelligent User Interfaces, 2019.

[24] W. Samek, T. Wiegand, and K.-R. Müller, “Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models,” ITU Journal: ICT Discoveries, 2017.

[25] Q. V. Liao, D. Gruen, and S. Miller, “Questioning the AI: Informing Design Practices for Explainable AI User Experiences,” Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, 2020.

[26] P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis, “Explainable AI: A Review of Machine Learning Interpretability Methods,” Entropy, vol. 23, no. 1, p. 18, 2020.

- [27] Y. Dong et al., “A Survey on In-context Learning,” arXiv preprint arXiv:2301.00234, 2023.
- [28] Z. Niu, G. Zhong, and H. Yu, “A review on the attention mechanism of deep learning,” *Neurocomputing*, vol. 452, pp. 48-62, 2021.
- [29] F. Doshi-Velez and B. Kim, “Towards A Rigorous Science of Interpretable Machine Learning,” arXiv preprint arXiv:1702.08608, 2017.
- [30] C. Molnar, “Interpretable Machine Learning: A Guide for Making Black Box Models Explainable,” 2020.
- [31] I. H. Witten et al., “Data Mining: Practical Machine Learning Tools and Techniques,” Morgan Kaufmann, 2016.
- [32] H. Nori, S. Jenkins, P. Koch, and R. Caruana, “InterpretML: A Unified Framework for Machine Learning Interpretability,” arXiv preprint arXiv:1909.09223, 2019.
- [33] R. Dwivedi et al., “Explainable AI (XAI): Core Ideas, Techniques, and Solutions,” *ACM Computing Surveys*, 2023.