DOI: 10.15514/ISPRAS-2026-38(6)-29



Artificial Intelligence for Software Engineering (AI4SE): A Review of Capabilities, Evaluation, and Verification Challenges

S.M. Avdoshin, ORCID: 0000-0001-8473-8077 <savdoshin@hse.ru>
E.Y. Pesotskaya, ORCID: 0000-0003-2129-4645 <epesotskaya@hse.ru>
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. Artificial intelligence has significantly reshaped contemporary software engineering practice, changing how software systems are designed, implemented, tested, and maintained. Recent advances in large language models and AI-assisted development tools have expanded the role of AI from supporting isolated tasks to participating in multiple stages of the software lifecycle. In many development settings, software is now created from high-level specifications and contextual requirements, while developers increasingly focus on supervision, validation, and coordination of generated outputs. Despite rapid growth in Artificial Intelligence for Software Engineering (AI4SE), the field remains fragmented, with no widely accepted framework connecting lifecycle stages, model capabilities, autonomy, context, and evaluation. This paper presents a review of AI4SE from a systems perspective and proposes an analytical model for structuring AI4SE analysis that links these dimensions within a single analytical framework. Based on this model, the study proposes a taxonomy of AI capabilities in software engineering, including code generation, transformation, analysis, testing, and interactive assistance, together with their practical limitations. The paper also examines recent developments in AI-assisted development workflows, formal verification of generated code, and multi-agent approaches to automated testing. Special focus is placed on evaluation challenges, including reproducibility, limited transferability of benchmark results to real development environments, and the decreasing usefulness of heavily optimized benchmark datasets. To support more balanced assessment of AI-assisted software engineering systems, the paper proposes a multi-dimensional utility model combining correctness, efficiency, maintainability, and operational risk. The analysis indicates a shift from isolated code-generation tools toward more integrated and partially autonomous development workflows. Reliable adoption of such systems will depend on stronger validation and verification practices, better integration of project context, and evaluation methods that more accurately reflect real software engineering conditions.

Keywords: Artificial Intelligence for Software Engineering (AI4SE); large language models; software engineering evaluation; AI-assisted software development; formal verification.

For citation: Avdoshin S.M., Pesotskaya E. Y. Artificial Intelligence for Software Engineering (AI4SE): A Review of Capabilities, Evaluation, and Verification Challenges. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 6, part 2, pp. 133-146. DOI: 10.15514/ISPRAS-2026-38(6)-29.

Acknowledgements. This research was supported in part by the computational resources of HPC facilities at HSE University.

Искусственный интеллект для программной инженерии (AI4SE): обзор возможностей, методов оценки и проблем верификации

С.М. Авдошин, ORCID: 0000-0001-8473-8077 savdoshin@hse.ru
Е.Ю. Песоцкая, ORCID: 0000-0003-2129-4645 <epesotskaya@hse.ru>
Национальный исследовательский университет «Высшая школа экономики»,
Россия, 101000, Москва, Мясницкая улица, д. 20.

Аннотация. Искусственный интеллект (ИИ) все заметнее влияет на программную инженерию, меняя подходы к разработке программного обеспечения. Современные большие языковые модели и инструменты разработки с поддержкой ИИ расширили применение искусственного интеллекта от выполнения отдельных вспомогательных задач до участия в различных этапах жизненного цикла программного обеспечения. Во многих современных процессах разработки программные системы все чаще создаются на основе высокоуровневых спецификаций и контекстных требований, тогда как роль разработчика в большей степени смещается в сторону контроля, валидации и координации полученных результатов. Несмотря на быстрое развитие направления Искусственный интеллект для программной инженерии (Artificial Intelligence for Software Engineering, AI4SE), эта область по-прежнему остается фрагментированной и не имеет единой модели, связывающей этапы жизненного цикла разработки, возможности моделей, уровни автономности, контекст применения и подходы к оценке. В работе представлен обзор методов AI4SE с системной точки зрения и предложена формальная модель, объединяющая эти измерения в рамках единого аналитического подхода. На этой основе предложена классификация возможностей искусственного интеллекта в программной инженерии, включая генерацию, преобразование, анализ, тестирование и интерактивную поддержку, а также рассмотрены связанные с ними практические ограничения. В статье также рассматриваются современные подходы к разработке с поддержкой ИИ, формальная верификация сгенерированного кода и мультиагентные подходы к автоматизированному тестированию. Особое внимание уделено методам оценки подобных систем, включая проблемы воспроизводимости результатов, ограниченную переносимость результатов бенчмарков на реальные условия разработки и снижение их информативности по мере широкого использования. Для более сбалансированной оценки систем программной инженерии с поддержкой ИИ предложена многомерная метрика полезности, объединяющая корректность, эффективность, сопровождаемость и уровень риска. Проведенный анализ показывает переход от изолированных инструментов генерации кода к более интегрированным и частично автономным процессам разработки. Надежное внедрение таких систем требует более строгих процедур валидации, более глубокой интеграции проектного контекста и методов оценки, лучше отражающих реальные условия программной инженерии.

Ключевые слова: искусственный интеллект для программной инженерии (AI4SE); большие языковые модели; методы оценки в программной инженерии; разработка программного обеспечения с поддержкой искусственного интеллекта; формальная верификация.

Для цитирования: Авдошин С.М., Песоцкая Е. Ю. Искусственный интеллект для программной инженерии (AI4SE): обзор возможностей, методов оценки и проблем верификации. Труды ИСП РАН, 2026, том 38, вып. 6, часть 2, стр. 133–146 (на английском языке). DOI: 10.15514/ISPRAS-2026-38(6)-29.

Благодарности. Исследование выполнено с использованием суперкомпьютерного комплекса НИУ ВШЭ.

1. Introduction

Artificial intelligence (AI) is increasingly influencing the way software systems are designed, implemented, tested, and maintained. Earlier applications of machine learning in software engineering (SE) were mainly associated with predictive and analytical tasks such as defect detection, effort estimation, recommendation systems, and project analytics [1-2]. The emergence of large language models (LLMs) and generative architectures has considerably expanded these

capabilities. Modern AI systems are now able to generate source code, assist in debugging, create tests and documentation, and support a broad range of engineering activities that previously required substantial manual effort [2-4].

This shift affects not only the set of available development tools, but also the organization of software engineering processes. In AI-assisted workflows, code increasingly functions as a derivative artifact produced from higher-level specifications, contextual information, and natural language descriptions. The role of the developer correspondingly shifts from direct implementation toward supervision, validation, and coordination of generated outputs. Recent studies describe this transition as an important change in software engineering practice and development workflows [4-5].

At the same time, current AI systems remain fundamentally probabilistic. Although large language models can generate syntactically valid and functionally plausible code, they do not provide guarantees of semantic correctness, reliability, security, or compliance with system-level constraints [5-6]. Existing evaluation methods are often limited to benchmark-oriented tasks and isolated code generation scenarios, making it difficult to assess how these systems behave in realistic development environments. Studies on code representation and semantic code search illustrate the importance of evaluating different aspects of code understanding and retrieval, while also highlighting that performance on specific tasks provides only a partial view of AI capabilities [7–9].

Recent research also shows the gradual transition from isolated AI assistants toward more integrated and partially autonomous development workflows. Contemporary AI systems increasingly combine generation, testing, execution, and feedback within iterative pipelines that support multiple stages of the software lifecycle. As a result, AI increasingly participates not only in local programming tasks, but also in broader engineering processes involving coordination, refinement, and validation [5, 10]. This evolution raises additional questions related to reproducibility, traceability, operational risk, and governance of AI-generated artifacts.

Recent Artificial Intelligence for Software Engineering (AI4SE) survey papers have examined the field from several complementary perspectives, including software engineering tasks, LLM architectures, agentic workflows, software lifecycle activities, and evaluation methodologies [3-4]. These studies provide comprehensive overviews of their respective topics and have substantially advanced understanding of AI4SE. However, they generally organize these perspectives independently rather than representing their interactions within a single analytical framework suitable for comparative analysis of AI-assisted software engineering systems. Accordingly, the objective of this study is not to replace existing surveys or taxonomies but to complement them through an analytical representation that makes relationships among these dimensions explicit.

Building on these surveys, this paper proposes a structured analysis of AI4SE from a system-level perspective. The study introduces an analytical model connecting lifecycle stages, AI model categories, autonomy levels, contextual factors, and evaluation criteria within a unified analytical framework. Based on this analytical model, the paper develops a functional classification of AI capabilities in software engineering and examines practical limitations associated with synthesis, transformation, analysis, testing, and interactive assistance tasks.

Unlike descriptive classifications, the proposed model explicitly represents interactions between software lifecycle stages, AI model classes, levels of autonomy, contextual factors, and evaluation criteria, enabling consistent analysis of AI4SE systems across different application scenarios.

Particular attention is given to repository-scale development workflows, formal verification of generated code, and evaluation of AI-assisted software engineering systems. The paper also discusses limitations of current benchmark practices, including restricted task scope, reproducibility issues, and limited transferability of benchmark results to real development settings. To support more balanced assessment of AI-assisted systems, a multi-dimensional utility model combining correctness, efficiency, maintainability, and operational risk is proposed.

The main contributions of the paper are as follows:

- a formal system-level model of AI4SE integrating lifecycle stages, autonomy, context, and evaluation;
- a system-level organization of AI capabilities and their practical limitations across software engineering tasks;
- an analysis of structural limitations in current benchmark and evaluation practices;
- a multi-dimensional utility model for assessing AI-assisted software engineering systems.

2. Research Methodology

This study follows a structured literature review methodology aimed at synthesizing current AI4SE research and informing the development of a system-level analytical model. The review was intended to identify recurring methodological patterns, relate AI approaches to software engineering activities, and synthesize observations that informed the analytical model proposed in this paper.

Publications were collected from IEEE Xplore, ACM Digital Library, Scopus, Web of Science, and arXiv. The search process used combinations of terms including AI for software engineering, large language models for code, AI-assisted programming, automated software engineering, AI testing, formal verification, and agentic software engineering. To improve coverage and reduce database bias, backward and forward citation analysis was additionally applied where appropriate [11].

The initial search identified 642 publications. After duplicate removal and title/abstract screening, 186 studies were selected for full-text review. The final analysis included 127 publications that satisfied the inclusion criteria described below.

The final dataset comprised peer-reviewed journal articles, conference papers from major software engineering and artificial intelligence venues, systematic reviews, benchmark studies, and a limited number of technical reports with empirical validation. Preference was given to peer-reviewed studies with clearly described methodologies, empirical validation, publicly available benchmarks, or reproducible experimental settings. Most included studies were published between 2020 and 2026, reflecting the rapid development of AI4SE following the emergence of large language models and generative AI.

To position the present study with respect to recent AI4SE survey literature, we conducted a comparative analysis of the analytical perspectives covered by the most comprehensive survey papers published between 2023 and 2026. This period was selected because it captures the emergence of large language model-based and agentic AI approaches that have substantially reshaped contemporary AI4SE research. The results of this comparative analysis are summarized in Table 1.

Table 1. Analytical perspectives covered by recent AI4SE survey literature.

Analytical dimension covered in recent surveys	**Representative studies**
AI4SE tasks and capabilities	[3, 12-14]
LLM architectures and code generation	[3, 13-14]
Agent-based software engineering	[12, 14]
Software lifecycle perspective	[15]
Evaluation benchmarks and performance metrics	[3, 14-15]
Context-aware AI-assisted software engineering	[12, 14]

Recent AI4SE surveys provide extensive coverage of individual aspects of AI-assisted software engineering, including task categories, model architectures, lifecycle activities, autonomy, context, and evaluation. The contribution of the proposed model is not the introduction of these dimensions as new categories, but their joint representation as a system configuration. This makes it possible to

examine how a particular combination of model class, autonomy level, lifecycle stage, and operational context affects the evaluation requirements of an AI4SE system. For example, two systems based on large language models and applied to code generation may require different validation strategies when one provides interactive assistance within an IDE and the other autonomously modifies a software repository. By explicitly representing these relationships, the model supports structured analysis of system configurations, their constraints, and associated quality trade-offs. It complements existing task-specific taxonomies by providing a common analytical basis for comparing AI4SE systems across different engineering settings, rather than ranking their performance.

2.1 Selection Criteria and Scope

The inclusion criteria were designed to balance coverage and relevance. The review prioritized studies that satisfied at least one of the following conditions: proposed AI methods applicable to software engineering tasks; evaluated large language models or generative systems on programming-related benchmarks; addressed software lifecycle activities such as coding, testing, maintenance, or deployment; examined evaluation practices, reproducibility, or benchmark design; discussed autonomous or agentic software engineering workflows.

Studies focused exclusively on general natural language processing tasks without software engineering relevance were excluded. Older publications were included selectively when they introduced widely adopted benchmarks, foundational models, or influential concepts that remain relevant to current AI4SE research. Papers lacking methodological description or empirical evaluation were also excluded unless they introduced broadly adopted conceptual frameworks or widely used software engineering tools.

The resulting dataset covers several interconnected research areas, including machine learning for software engineering, deep learning for source code analysis, automated testing, AI-assisted development environments, formal verification, and repository-scale agentic systems.

2.2 Classification Scheme

To organize the analysis, the reviewed studies were classified along five analytical dimensions: software lifecycle stage, AI method, level of autonomy, evaluation focus, and context.

The lifecycle dimension follows the standard decomposition of software engineering activities into requirements, design, coding, testing, deployment, and maintenance. This enables positioning of AI methods within specific engineering tasks and facilitates identification of areas where automation has reached different levels of maturity.

The second dimension describes the type of AI approach used in each study. The reviewed literature includes classical machine learning methods, deep learning models, transformer-based architectures, large language models, reinforcement learning systems, and hybrid neuro-symbolic approaches. These methods differ not only in their technical implementation but also in their ability to incorporate contextual information and generalize across software engineering tasks.

The third dimension reflects the degree of autonomy. Existing systems range from assistive tools operating under direct developer control to semi-autonomous workflows capable of performing sequences of engineering activities with limited supervision. Emerging agentic systems extend this concept by coordinating planning, generation, testing, execution, and refinement within iterative development pipelines.

The fourth dimension concerns evaluation. The review distinguishes between studies focused primarily on benchmark accuracy and those considering broader quality attributes such as maintainability, computational efficiency, robustness, reproducibility, and operational risk.

The fifth dimension captures contextual factors influencing AI-assisted software engineering, including repository scale, software architecture, application domain, availability of project

knowledge, and operational constraints. These factors increasingly influence the effectiveness of modern LLM-based and agentic software engineering systems.

Figure 1 summarizes the distribution of AI methods across software lifecycle stages and levels of autonomy. Current research remains concentrated in coding and testing tasks, whereas highly autonomous AI systems are comparatively less common and unevenly represented across the software lifecycle.

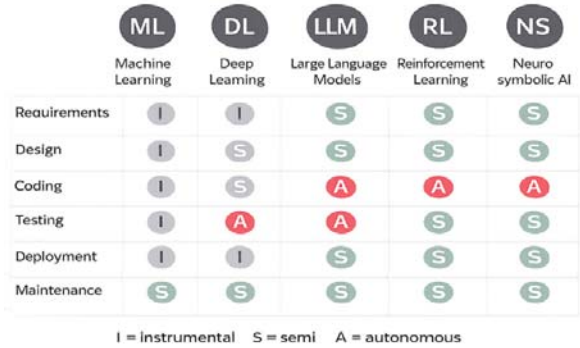


Fig. 1. AI methods across lifecycle stages and levels of autonomy.

Each reviewed study was independently assigned to one or more lifecycle stages, AI method categories, autonomy levels, contextual settings, and evaluation criteria based on its primary contribution and experimental focus. Since the objective of the review was conceptual synthesis rather than quantitative evidence aggregation, formal inter-rater agreement statistics were not calculated. Studies whose classification was not immediately clear were discussed jointly by the authors until consensus was reached. Although this classification provides a consistent descriptive organization of the reviewed literature, it does not explicitly represent interactions among contextual factors, autonomy, and evaluation criteria. These interactions are addressed by the analytical model introduced in the next section.

3. Analytical Model of AI4SE

The classification introduced in the previous section provides a consistent organization of current AI4SE research, but it remains descriptive. In practice, AI-assisted software engineering involves dependencies between software lifecycle activities, AI model capabilities, contextual constraints, levels of autonomy, and evaluation requirements. These dimensions influence one another and are therefore better understood as parts of a single analytical representation. Accordingly, the proposed AI4SE model describes each system as a configuration of five interacting analytical dimensions:

AI4SE = (S, M, A, C, E), where:

- **S** – the software lifecycle stages, including requirements, design, coding, testing, deployment, and maintenance;
- **M** – the AI model class, including machine learning, deep learning, large language models, reinforcement learning, and neuro-symbolic approaches;
- **A** – the level of autonomy, ranging from assistive tools to autonomous multi-agent systems;
- **C** – the operational context, including software architecture, application domain, repository characteristics, available project knowledge, and non-functional constraints;
- **E** – the evaluation criteria represented as a vector of quality attributes, including correctness, efficiency, maintainability, robustness, and operational risk.

The five dimensions define a system configuration rather than a set of independent attributes. A particular configuration is meaningful only when its components are compatible with one another. The lifecycle stage (S) and operational context (C) establish the engineering requirements and constraints, while the AI model class (M) characterizes the approach used to address them. The autonomy level (A) indicates the extent of independent execution supported by the configuration and affects the required degree of human oversight, validation, and risk control. Accordingly, evaluation criteria (E) should be selected in relation to the combined configuration of the system, rather than considered independently. For example, code generation may be assessed in terms of correctness and maintainability, whereas automated testing may require measures of coverage and fault detection. These relationships provide analytical guidelines for interpreting AI4SE configurations, without imposing universal constraints across different engineering contexts.

For example, repository-scale code generation assistance can be represented by the configuration (coding, LLM, semi-autonomous, repository context, correctness and maintainability evaluation). In contrast, automated defect prediction corresponds to a different configuration involving the testing or maintenance stage, classical machine learning, assistive autonomy, project-specific historical data, and evaluation focused on predictive performance. The proposed representation therefore characterizes AI4SE systems by their complete analytical configuration rather than by a single AI technique.

Errors in AI-assisted workflows are also configuration-dependent. In partially autonomous development pipelines, incorrect assumptions introduced during code generation may propagate through testing, integration, and maintenance. As the level of autonomy increases, evaluation, validation, and traceability become increasingly important for limiting error propagation and maintaining engineering reliability.

3.1 Functional Classes of AI Capabilities in Software Engineering

Within the proposed model, AI capabilities can be organized into five functional classes reflecting how AI systems interact with software artifacts during development.

Table 2 summarizes the main classes of AI capabilities identified from recent AI4SE surveys and benchmark studies [3, 10, 12-15], together with their typical limitations.

Table 2. Functional Classes of AI Capabilities.

Capability Class	Typical tasks	Limitations	References
Synthesis	Code generation from specifications	Hallucinations, incorrect solutions	[6, 16]
Transformation	Refactoring and code migration	Loss of semantic equivalence in complex logic	[17]
Analysis	Defect and vulnerability detection	Dependence on training data, false positives	[6, 10]
Testing	Generate test cases, support validation	Limited semantic coverage, unreliable oracles	[18-19]
Interaction	Q&A and documentation	Context window limitations	[20]

LLM-based systems can generate syntactically valid and plausible outputs that do not necessarily meet the semantic requirements of a task. This makes validation essential, particularly in software engineering activities where correctness and reliability are critical. In complex scenarios, incomplete context or incorrect assumptions may lead to errors that are difficult to detect, highlighting the importance of verification, testing, and traceability in AI-assisted development.

3.2 Application of the Proposed Model to Representative AI4SE Systems

The proposed model was applied to three representative systems: GitHub Copilot, SWE-agent, and EvoSuite. These systems were selected to illustrate different approaches to software engineering tasks, ranging from developer-assisted code generation to repository-level task execution and automated test generation. Their configurations are summarized in Table 3.

Table 3. Application of the proposed AI4SE model to representative systems.

System	Lifecycle stage	AI approach and autonomy	Operational context	Evaluation focus
GitHub Copilot	Coding	LLM-based; assistive	IDE, current code, available project context	Code correctness, relevance, developer productivity
SWE-agent [21]	Coding, testing, maintenance	LLM-based agent; multi-step task execution	Repository, files, tools, execution environment	Task completion, issue resolution, test results
EvoSuite [18]	Testing	Evolutionary algorithms; automated test generation	Java classes, execution environment	Structural coverage and test-suite properties

The comparison illustrates how the joint configuration of AI4SE systems reveals differences that cannot be captured by the AI approach alone. GitHub Copilot provides coding assistance within an IDE, whereas SWE-agent addresses repository-level tasks through multi-step interaction with files, tools, and the execution environment. EvoSuite, in contrast, focuses on automated testing. These differences are reflected across the model's dimensions, including lifecycle stage, autonomy, and operational context, and lead to distinct evaluation requirements. For interactive coding assistance, code correctness and developer productivity are relevant, while repository-level agents also require assessment of task completion, test outcomes, and the risks associated with changes across multiple development steps. For automated testing, structural coverage and test-suite properties are more relevant than code-generation productivity. The comparison shows how the proposed model links system configurations to their evaluation criteria. However, this analysis is descriptive and does not constitute a controlled assessment of system performance.

3.3 Context and Autonomy in AI-Assisted Development

Among the dimensions introduced above, context and autonomy have emerged as two of the most influential factors in recent AI4SE research.

Most current AI systems operate with incomplete access to project information. Although modern models can process substantially larger contexts than earlier architectures, they still do not maintain a complete representation of software structure, dependency history, or long-term design rationale. As a result, generated outputs may appear correct locally while remaining inconsistent with broader system requirements.

Context becomes especially important in repository-scale development tasks where successful generation depends not only on syntax, but also on architectural conventions, dependency relationships, testing infrastructure, and non-functional constraints.

Autonomy introduces additional complexity. Assistive systems typically operate under direct human supervision and therefore limit propagation of errors. In more autonomous workflows, generated artifacts may directly influence downstream development stages. Incorrect outputs can therefore accumulate through iterative feedback loops if validation mechanisms are insufficient.

Recent agentic systems extend this idea further by coordinating planning, implementation, testing, and refinement across multiple interconnected steps [5]. While such systems improve automation capabilities, they also introduce challenges related to traceability, reproducibility, and governance.

3.4 AI-Assisted Development Pipelines

The transition from isolated assistance tools toward integrated AI-driven workflows is one of the most visible changes in current software engineering practice.

Traditional development pipelines generally separate implementation, testing, validation, and maintenance into distinct stages coordinated primarily by developers. AI-assisted pipelines increasingly integrate these activities through iterative generation and feedback mechanisms.

A simplified AI-assisted pipeline typically includes:

1. specification or task definition;
2. generation of code or related artifacts;
3. execution and validation;
4. feedback and refinement;
5. integration into the development workflow.

This process changes the role of code within the engineering lifecycle. Code becomes an intermediate artifact subject to continuous refinement rather than a final manually constructed product.

At the same time, AI-driven pipelines remain strongly dependent on validation quality. Errors introduced during specification or generation may propagate through later stages if testing and verification are incomplete. This issue becomes more significant as workflows become increasingly autonomous.

For this reason, recent research has shown growing interest in combining generation with stronger validation mechanisms, including formal verification, repository-aware reasoning, automated testing, and iterative evaluation procedures [18-19].

Figure 2 illustrates one possible realization of the proposed analytical model, showing how software lifecycle stages, AI models, contextual information, autonomy, and evaluation interact within an iterative AI-assisted development process.

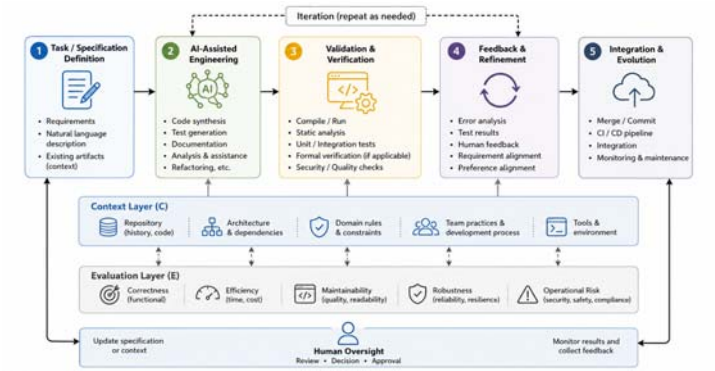


Fig. 2. AI-assisted development pipeline.

4. Evaluation and Benchmark Limitations in AI4SE

Evaluation remains one of the least stable areas of current AI4SE research. Although the number of benchmarks and reported performance results has increased rapidly in recent years, comparison

between studies remains difficult because datasets, prompts, execution settings, and reporting procedures often differ substantially.

Most existing benchmarks focus on relatively isolated programming tasks such as code completion, function synthesis, or short problem-solving scenarios. These benchmarks are useful for measuring selected capabilities of large language models, but they capture only a limited part of real software engineering activity. High benchmark performance therefore does not necessarily translate into robustness in large repositories, collaborative environments, or long-term maintenance workflows [8-9, 22].

Another limitation is that many evaluation approaches emphasize functional correctness while overlooking maintainability, computational efficiency, reproducibility, and operational risk. Recent benchmark studies have also started to evaluate execution efficiency of AI-generated code in addition to correctness, although such evaluation remains considerably less common than functional testing. Metrics such as pass@k or execution-based validation remain useful for estimating whether generated code satisfies a predefined test suite, but they do not fully capture broader software quality characteristics [6, 10].

Benchmark saturation has also become an increasing concern. As models are optimized directly or indirectly against widely used datasets, improvements in benchmark scores become harder to interpret. In some cases, high performance may reflect adaptation to benchmark structure or training-data overlap rather than meaningful improvement in reasoning or engineering capability [19, 23-24].

Reproducibility presents an additional challenge. Experimental outcomes often depend on prompt formulation, model version, decoding strategy, context length, and evaluation configuration. Even small changes in these parameters may produce substantially different results, while experimental settings are not always documented in sufficient detail [25].

4.1 Multi-Dimensional Utility Model

To address limitations of single-metric evaluation, this study proposes a multi-dimensional utility formulation:

$$U = \alpha C + \beta E + \gamma M - \delta R$$

where:

- U denotes the overall utility;
- C represents correctness;
- E denotes efficiency;
- M reflects maintainability;
- R corresponds to operational risk;
- $\alpha, \beta, \gamma, \delta$ are weighting coefficients dependent on the application domain.

The formulation represents trade-offs between evaluation dimensions rather than reducing software quality to a universal score. It uses a selected subset of the evaluation criteria in (E), namely correctness, efficiency, maintainability, and operational risk. Other criteria, such as robustness, may be assessed separately or incorporated when relevant to the application context.

Correctness concerns whether generated software satisfies specified requirements and expected behavior. Efficiency covers computational cost and developer productivity, while maintainability reflects readability, modularity, documentation, and ease of modification. Operational risk includes potential failures, vulnerabilities, prompt injection, and unintended manipulation of LLM-integrated development environments [26]. Broader software quality metrics have also been proposed to extend the evaluation of LLM-generated code beyond functional correctness.

The weighting coefficients should reflect the priorities of the intended application and may be elicited from relevant stakeholders, such as developers, technical leads, and security specialists. The

selection rationale and, where applicable, the aggregation of stakeholder assessments should be documented. Sensitivity analysis can then be used to examine how changes in the weights affect the resulting utility. The coefficients are context-dependent rather than universal constants, and the formulation is intended to make evaluation priorities explicit rather than establish a universal ranking of AI4SE systems.

The proposed utility formulation is conceptual and has not been empirically evaluated in this study. No component scores or utility values were measured for the representative AI4SE systems considered in Section 3.2. Accordingly, the formulation is presented as a basis for structuring context-dependent evaluation rather than as a validated metric for comparing system performance. Its empirical operationalization and validation are left for future research.

The following scenarios are illustrative and do not represent empirical evaluations. For example, in repository-scale code generation, correctness and maintainability are relevant because generated changes must satisfy requirements while remaining compatible with the existing codebase. Efficiency may reflect the effort and resources needed to generate, review, and integrate changes, while operational risk concerns the possibility of introducing defects, regressions, or vulnerabilities. In interactive code completion, response latency and developer productivity may receive greater emphasis. These scenarios illustrate how the relative importance of the evaluation dimensions can vary with the intended use.

4.2 Structural Limitations of Current Benchmarks

Several structural limitations appear repeatedly across current benchmark practices.

First, benchmark tasks are often too limited in scope. Short programming problems do not adequately represent architectural dependencies, evolving requirements, repository-scale reasoning, or collaborative engineering workflows.

Second, benchmark coverage varies across programming languages and development environments. Results obtained on a limited set of languages cannot automatically be generalized to other language ecosystems. MultiPL-E illustrates this issue by extending code-generation benchmarks to multiple programming languages, enabling more systematic cross-language evaluation [27].

Third, strong performance on specific benchmarks does not necessarily demonstrate equivalent improvements in broader software engineering capabilities. Evaluation results may be difficult to interpret when metrics capture only selected aspects of system behavior or when evaluation conditions differ across studies [25]. Potential overlap between evaluation data and model training data is an additional concern that requires direct investigation when interpreting benchmark results.

Finally, functional correctness alone provides an incomplete picture of generated code quality. Broader attributes, including maintainability, input validation, error handling, code structure, and documentation, require additional evaluation criteria [28]. Longer-term properties, such as integration stability and system evolution, also call for evaluation approaches that extend beyond isolated programming tasks.

These limitations suggest that future evaluation practices should move beyond isolated benchmark performance toward more realistic system-level assessment involving repositories, iterative workflows, contextual reasoning, and validation under changing development conditions.

4.3 Reproducibility and Reliability

Reproducibility remains a central issue in AI4SE research. Large language models are sensitive to variations in prompts, sampling parameters, context ordering, and execution environments. As a result, two studies evaluating the same model may report noticeably different outcomes under slightly different conditions [25].

Reliability is closely related to reproducibility but represents a distinct evaluation dimension. Current models frequently produce plausible but incorrect results, especially when contextual

information is incomplete or ambiguous. Such failures are not always immediately visible and may remain undetected during superficial evaluation.

This issue becomes more significant in partially autonomous development pipelines where generated artifacts directly influence downstream engineering activities. In these settings, local inaccuracies may propagate through multiple stages of development and become increasingly difficult to trace.

Growing attention has therefore been directed toward combining generative systems with stronger validation mechanisms, including formal verification and automated testing [18-19, 23, 29], as well as agent-based approaches to software engineering tasks [21].

5. Discussion

The analysis presented in this paper suggests that AI4SE is evolving from isolated assistance tools toward integrated software engineering workflows in which generation, validation, and feedback are closely interconnected. Recent advances in large language models have substantially expanded AI capabilities across coding, testing, documentation, and software analysis. At the same time, reliable deployment of these systems continues to depend on factors extending beyond model performance alone.

One of the central observations of this study is that AI-assisted software engineering cannot be adequately characterized using a single analytical perspective. The proposed model highlights that software lifecycle stage, AI model class, autonomy, operational context, and evaluation criteria jointly determine the behavior of AI-assisted systems. Considering these dimensions independently may therefore lead to incomplete conclusions regarding system capabilities or practical applicability.
Context remains one of the principal challenges. Most current AI systems operate with only partial knowledge of repository structure, architectural decisions, dependency relationships, and long-term project evolution. Consequently, outputs that appear locally correct may still conflict with broader engineering constraints or project-specific requirements.

Autonomy introduces a similar trade-off. Higher levels of automation improve development efficiency but also increase the importance of validation, traceability, and operational control. As AI systems become more deeply integrated into software engineering workflows, evaluation must increasingly consider engineering reliability in addition to benchmark performance.
The analysis also confirms several structural limitations of current evaluation practice. Existing benchmarks remain valuable for measuring individual capabilities, but they rarely capture repository-scale reasoning, long-term maintenance, collaborative development, or evolving project contexts. More comprehensive evaluation will therefore require system-level assessment that combines benchmark performance with contextual information, validation procedures, and operational considerations.

These observations suggest that future progress in AI4SE will depend not only on more capable foundation models but also on tighter integration between generation, validation, contextual reasoning, and engineering evaluation. From this perspective, the proposed analytical model provides a basis for comparing AI-assisted software engineering systems operating under different engineering conditions.

6. Conclusion

This paper presented a system-level analytical model for Artificial Intelligence for Software Engineering that relates software lifecycle stages, AI model classes, levels of autonomy, contextual factors, and evaluation criteria within a unified representation.

Building on this model, the study organized AI capabilities into five functional classes and examined current developments in repository-scale workflows, formal verification, and AI-assisted software engineering pipelines. The analysis further identified recurring limitations of current evaluation

practice, including restricted benchmark scope, reproducibility challenges, incomplete contextual reasoning, and insufficient consideration of engineering constraints.

To support more balanced assessment of AI-assisted software engineering systems, the paper introduced a multi-dimensional utility model incorporating correctness, efficiency, maintainability, and operational risk. Rather than providing a universal scoring function, the proposed formulation offers a configurable framework for evaluating AI systems under different software engineering conditions.

The main contribution of this work is not a new taxonomy of AI methods but a unified analytical representation that makes explicit the relationships among lifecycle activities, AI models, autonomy, context, and evaluation. This perspective supports more systematic comparison of AI-assisted software engineering systems and complements existing AI4SE surveys. The proposed framework is intended to support future comparative analyses rather than replace existing AI4SE taxonomies or evaluation methodologies.

References

[1]. Menzies T., Zimmermann T. Software Analytics: So What? IEEE Software, 2013, vol. 30, no. 4, pp. 31-37. DOI: 10.1109/MS.2013.86.

[2]. Yang Y., Xia X., Lo D., Grundy J. A Survey on Deep Learning for Software Engineering. ACM Computing Surveys, 2022, vol. 54, no. 10, Art. 206, pp. 1-73. DOI: 10.1145/3505243.

[3]. Fan A. et al. Large Language Models for Software Engineering: Survey and Open Problems. arXiv preprint, 2023. DOI: 10.48550/arXiv.2310.03533.

[4]. Görmez M.K., Yilmaz M., Clarke P.M. Large Language Models for Software Engineering: A Systematic Mapping Study. In Systems, Software and Services Process Improvement, Springer, 2024. DOI: 10.1007/978-3-031-71139-8_5.

[5]. Belzner L., Gabor T., Wirsing M. Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study. In Bridging the Gap between AI and Reality, Springer, 2024. DOI: 10.1007/978-3-031-46002-9_23.

[6]. Chen M. et al. Evaluating Large Language Models Trained on Code. arXiv preprint, 2021. DOI: 10.48550/arXiv.2107.03374.

[7]. Alon U. et al. Code2vec: Learning Distributed Representations of Code. In Proc. ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), 2019. DOI: 10.1145/3290353.

[8]. Guo D. et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proc. ICLR, 2021. DOI: 10.48550/arXiv.2009.08366.

[9]. Husain F. et al. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv preprint, 2019. DOI: 10.48550/arXiv.1909.09436.

[10]. Amalfitano D. et al. Artificial Intelligence Applied to Software Testing: A Tertiary Study. ACM Computing Surveys, 2023, vol. 56, no. 3, pp. 1-38. DOI: 10.1145/3616372.

[11]. Wohlin C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In Proc. EASE, 2014. DOI: 10.1145/2601248.2601268.

[12]. Liu J., Wang K., Chen Y., Peng X., Chen Z., Zhang L., Lou Y. Large Language Model-Based Agents for Software Engineering: A Survey. arXiv preprint, 2024. DOI:10.48550/arXiv.2409.02977.

[13]. Jiang J., Wang F., Shen J., Kim S., Kim S. A Survey on Large Language Models for Code Generation. ACM Computing Surveys, 2026. DOI:10.1145/3747588.

[14]. Jin H., Huang L., Cai H., Yan J., Li B., Chen H. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv preprint, 2024. DOI: 10.48550/arXiv.2408.02479.

[15]. Wang K., Li T., Zhang X., Wang C., Sun W., Liu Y., Shi B. Software Development Life Cycle Perspective: A Survey of Benchmarks for Code Large Language Models and Agents. arXiv preprint, 2025. DOI: 10.48550/arXiv.2505.05283.

[16]. Austin J. et al. Program Synthesis with Large Language Models. arXiv preprint, 2021. DOI: 10.48550/arXiv.2108.07732.

[17]. Le Goues C., Pradel M., Roychoudhury A. Automated Program Repair. Communications of the ACM, 2019, vol. 62, no. 12, pp. 56-65. DOI: 10.1145/3318162.

[18]. Fraser G., Arcuri A. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. ACM Transactions on Software Engineering and Methodology, vol. 24, no. 2, Art. 8, pp. 1-42, 2014. DOI: 10.1145/2685612.

[19]. Molinelli D., Di Grazia L., Martin-Lopez A., Ernst M.D., Pezzè M. Do LLMs Generate Useful Test Oracles? An Empirical Study with an Unbiased Dataset. In Proc. 40th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), 2025. DOI: 10.1109/ASE63991.2025.00031.

[20]. Al Haque E., Brown C., LaToza T.D., Johnson B. The Evolution of Information Seeking in Software Development: Understanding the Role and Impact of AI Assistants. In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), 2025, pp. 1494–1502. DOI: 10.1145/3696630.3731665.

[21]. Yang J., Jimenez C.E., Wettig A., Lieret K., Yao S., Narasimhan K., Press O. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. Advances in Neural Information Processing Systems, 2024, vol. 37, DOI: 10.52202/079017-1601.

[22]. Bubeck S. et al. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. arXiv preprint, 2023. DOI: 10.48550/arXiv.2303.12712.

[23]. Sevenhuijsen M., Etemadi K., Nyberg M. VeCoGen: Automating Generation of Formally Verified C Code with Large Language Models. In Proc. IEEE/ACM Int. Workshop on Formal Methods in Software Engineering (FormaliSE) , 2025, pp. 101-112. DOI: 10.1109/FormaliSE66629.2025.00017.

[24]. Shanahan M. Talking About Large Language Models. arXiv preprint, 2022. DOI: 10.48550/arXiv.2212.03551.

[25]. Torun U.B. et al. Evaluation of LLM-Based Software Engineering Tools: Practices, Challenges, and Future Directions. arXiv preprint, 2026. DOI: 10.48550/arXiv.2604.24621.

[26]. Greshake K. et al. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. arXiv preprint, 2023. DOI: 10.48550/arXiv.2302.12173.

[27]. Cassano F., Gouwar J., Nguyen D. et al. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. IEEE Transactions on Software Engineering, 2023, vol. 49, no. 7, pp. 3675–3691. DOI: 10.1109/TSE.2023.3267446.

[28]. Umre J., Parihar A.S., Gupta A. From Correctness to Code Quality: Formalizing Software Engineering Metrics for Evaluating General LLMs. in Proceedings of the 19th Innovations in Software Engineering Conference (ISEC '26), Jaipur, India, Feb. 19-21, 2026. New York, NY, USA: ACM, 2026. DOI: 10.1145/3796563.3796609.

[29]. Bursuc S. et al. A Benchmark for Vericoding: Formally Verified Program Synthesis. arXiv preprint, 2025. DOI: 10.48550/arXiv.2509.22908.

Информация об авторах / Information about authors

Сергей Михайлович АВДОШИН – кандидат технических наук, профессор департамента компьютерной инженерии МИЭМ НИУ ВШЭ. Сфера научных интересов: программная инженерия, разработка программного обеспечения с поддержкой искусственного интеллекта (AI4SE), большие языковые модели, формальная верификация и методы оценки в программной инженерии, проектирование и разработка корпоративных обучаемых экспертных систем, криптография и защита информации.

Sergey Michailovich AVDOSHIN – Cand. Sci. (Tech.), Prof., Professor at the Department of Computer Engineering, MIEM, HSE University. Research interests: software engineering, AI-assisted software development (AI4SE), large language models, formal verification and evaluation methods in software engineering, design and development of corporate trainable expert systems, cryptography and information security.

Елена Юрьевна ПЕСОЦКАЯ – кандидат экономических наук, доцент департамента программной инженерии, НИУ ВШЭ. Сфера научных интересов: программная инженерия, машинное обучение, управление рисками программных проектов.

Elena Yurievna PESOTSKAYA – Cand. Sci. (Econ.), Assistant Professor at the Department of Software Engineering, HSE University. Research interests: software engineering, machine learning, software risk management.