Towards Neuro-Symbolic Assurance for Artificial Intelligence via Graph Transformation

Lukas Sebastian Hofmann[1,2,*], Jens Kosiol[1], José Ignacio Requeno Jarabo[2], Reiko Heckel[3] and Gabriele Taentzer[1]

[1]*Philipps-Universität Marburg, Department of Mathematics and Computer Science, Hans-Meerwein-Straße, 35032 Marburg, Germany*

[2]*Universidad Complutense de Madrid, Facultad de Informática, Calle del Profesor José García Santesmases, 28040 Madrid, Spain*

[3]*University of Leicester, Department of Computing and Mathematical Sciences, University Road, Leicester, LE1 7RH, England*

Abstract

Artificial intelligence (AI) is becoming an increasingly common feature in software engineering. As AI systems grow in autonomy, they can address progressively complex, multi-step tasks without human interaction. This evolution, however, introduces significant challenges with respect to ensuring quality attributes, such as correctness, completeness, reliability, traceability, and trustworthiness. Given that many AI techniques are commonly considered as black-box systems, assurance for AI tends to focus on the inputs and outputs, as well as the execution paths within AI-based workflows and agentic systems. We review the key assurance modes for AI usage and demonstrate them using an example from software modelling. Then we situate our vision within the existing body of related work. Finally, we outline how graph transformation techniques can be used to support and enhance AI assurance.

Keywords

Generative AI, AI Recommendation Systems, AI Workflows, Agentic AI, Software System Architecture, Quality Assurance, Verification and Validation, Graph Transformation, Neuro-Symbolic Approach

1. Introduction

Artificial Intelligence (AI) systems are increasingly being used in high-stakes software engineering endeavours, including the design and development of healthcare applications, aerospace software, cyber-physical systems, and critical infrastructure systems [1, 2]. These systems span a broad spectrum of applications, ranging from conventional recommendation systems operating in controlled settings, to semi-automated AI workflows and, ultimately, to fully autonomous agentic architectures [3, 4, 5]. Such systems rely on complex AI models that often behave as black boxes from the user's perspective, revealing only the observable inputs and outputs and their relationships. Moreover, as these systems become more autonomous, they can solve increasingly complex, multi-step tasks [6, 7]. At the same time, their potential impact and the consequences of failure grow substantially, depending on the operational context; cf. Figure 1. Next, we will briefly review the main AI usage modes (system architectures) and highlight the associated challenges and risks in terms of assurance.

AI recommendation systems are commonly encountered through chat-based interfaces provided by proprietary platforms such as AI services from OpenAI[1] and Anthropic[2]. Comparable capabilities are

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29–July 3, 2026

*Corresponding author.

Lukas.Hofmann@uni-marburg.de (L. S. Hofmann); kosiolje@mathematik.uni-marburg.de (J. Kosiol); jrequeno@ucm.es (J. I. Requeno Jarabo); rh122@leicester.ac.uk (R. Heckel); taentzer@mathematik.uni-marburg.de (G. Taentzer)

https://www.ucm.es/jrequeno/ (J. I. Requeno Jarabo); https://le.ac.uk/people/reiko-heckel (R. Heckel); https://www.uni-marburg.de/de/fb12/arbeitsgruppen/swt (G. Taentzer)

0009-0004-3823-3894 (L. S. Hofmann); 0000-0003-4733-2777 (J. Kosiol); 0000-0001-5111-8357 (J. I. Requeno Jarabo); 0000-0003-4719-0772 (R. Heckel); 0000-0002-3975-5238 (G. Taentzer)

© 2026 Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

[1]https://openai.com/

[2]https://www.anthropic.com/

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/GCM-paper4.pdf

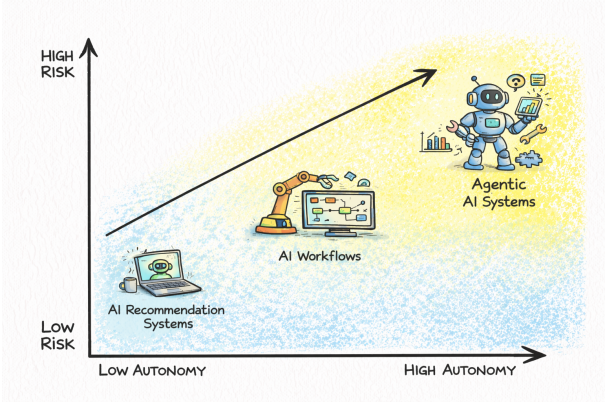


Figure 1: Conceptual illustration of the progression of AI systems along autonomy and risk dimensions

also emerging with Gemma 4[3] and DeepSeek[4], which can be run with open-source ecosystems like Ollama[5] combined with interfaces like open-webui[6]. This architectural setting typically exhibits a low degree of autonomy and risk, as a human is involved in the interaction with the outputs. They are therefore primarily used for knowledge work, such as information retrieval, content generation, and other types of text-encoded information [8, 9]. Nevertheless, these systems have known limitations, including hallucinations and susceptibility to manipulation. Consequently, they cannot guarantee correctness, and their trustworthiness depends on users critically evaluating the outputs and calibrating their expectations accordingly.

AI workflows offer a greater degree of process abstraction through the orchestration of actions, branches, and iterations. Such workflows can be implemented using tools such as n8n[7]. This AI usage is vulnerable to the same challenges as AI recommendation systems, and others like incorrect parameter passing, i.e., API miscalls or misinterpretation of retrieved information. However, every action in an AI workflow can be monitored and logged, which improves traceability. Trust in them can be established because, through the decomposition of a sequence of actions, various subtasks can be computed and documented using the appropriate building blocks.

AI agents, also known as *agentic AI*, represent the highest level of autonomy among the described architectures [10, 11]. These systems operate as independent agents with access to data, tools, and external resources, and they can delegate subtasks. The set of dominant frameworks for developing such agents includes LangChain[8], and AgentKit[1]. Additionally, there are ready-to-use agents, such as Claude Code[2], and OpenClaw[9], which provide predefined capabilities through modular specification documents, often referred to as *SKILLS*. Due to their autonomy and ability to act on external systems, AI agents introduce various challenges and risks.

Current research on AI assurance typically focuses on correctness, completeness, reliability, traceability, and trustworthiness as key quality attributes of AI systems. To assure the input and output of AI, assurance techniques range from prompting strategies (e.g. [12, 13]) to testing (e.g. [14, 15]) and verification techniques (e.g. [16, 17]). Current verification-based output assurance assumes that the AI results are programs or elements of other pre-defined languages. We did not find any verification approaches for structured, language-agnostic AI results. In the context of software engineering, language-agnostic denotes approaches that are not tied to a particular programming language, framework, or analysis tool, but are instead broadly applicable across different software development methods and techniques. *In our view, graph transformation–based, neuro-symbolic approaches provide deeper insights into the properties of structured, language-agnostic AI results than existing approaches, and thus offer assurance across a*

[3] https://deepmind.google/models/
[4] https://www.deepseek.com/en/
[5] https://ollama.com/
[6] https://github.com/open-webui/open-webui
[7] https://n8n.io/
[8] https://github.com/langchain-ai/langchain
[9] https://github.com/openclaw/openclaw

broader range of applications.

Our contributions are twofold: First, we provide an overview of how AI usage modes (system architectures) can be assured with neuro-symbolic approaches (see Section 2). Second, we present a practical demonstration of the AI usage modes using a software modelling example, thereby identifying graph transformation techniques applicable within assurance frameworks (Section 3). Finally, we discuss related work (Section 4) and summarise our considerations (Section 5).

2. Assurance of AI Usage Modes

In this section, we introduce how the three AI usage modes (system architectures), namely, (i) *AI recommendation systems*, (ii) *AI workflows*, and (iii) *AI agents* can be assured. These system architectures are shown in an abstracted form, alongside their respective assurance layers, in Figure 2. We differentiate between three assurance types:

- ***Input Assurance:*** Serves as a pre-processing boundary that retrieves or validates inputs to a black-box system, thereby mitigating the risk of, e.g., invalid, incorrect or Out-of-Distribution (OOD) inputs.
- ***Output Assurance:*** Constitutes a post-processing verification step that evaluates the outputs of a black-box system against specified constraints, ensuring that generated content remains within defined operational bounds and the constraints are satisfied.
- ***Assurance by Design:*** Focuses on the structural integrity of the system. It relies on transparent algorithms or symbolic logic, where the *execution path* is deterministic, interpretable, and amenable to verification.

The aim of AI assurance is to improve the quality of the participating artefacts and processes, as described in [16, 18, 19], the ISO/IEC 25059:2023[10], and ISO/IEC TR 24028:2020[11] standards:

1. ***Correctness***: the degree to which the system produces outputs that satisfy specified requirements, constraints, or intended goals.
2. ***Completeness***: the degree to which a system's output addresses all relevant aspects of an input query within a defined reference context.
3. ***Reliability***: the ability of the system to maintain consistent and dependable performance over time and across varying operating conditions.
4. ***Traceability***: the extent to which the system's decisions, outputs, and internal processes can be reconstructed, audited, and linked to their underlying data, models, or reasoning.
5. ***Trustworthiness***: the level of justified confidence, emerging from demonstrated correctness, completeness, reliability, and traceability, together with alignment to "user expectations".

Assured AI Recommendation Systems AI recommendation systems commonly rely on a single input–output interaction with an AI model [3, 4]. Thereby, the internal decision-making process of the model is typically opaque to the user and may exhibit non-deterministic behaviour. Still, the surrounding assurance framework can be designed in a deterministic and observable manner.

In particular, prompt creation for Large Language Models (LLMs) can be structured and controlled to transform raw user requests into well-defined inputs. This process may further include input validation, constraint checking, and repair mechanisms, resulting in a refined query that is subsequently submitted to, e.g., a LLM. While the internal inference process itself cannot usually be inspected or directly controlled, the quality of the interaction can be substantially improved through high-quality input. After inference, the generated output can be subjected to additional assurance steps, including validation, and repair mechanisms. In this way, assurance is established not through direct control of the model's internals, but through deterministic control of the framework surrounding the model. Representative application scenarios include *smart contract generation*, where outputs must satisfy a set of functional

[10] https://www.iso.org/standard/80655.html
[11] https://www.iso.org/standard/77608.html

Figure 2: A conceptual framework for assured AI usage modes

Legend: Gray boxes represent optional components

and non-functional constraints; *construction of knowledge graphs*, where logical consistency can be assured by the comparison to a type graph, or *requirements traceability graph*, where links between requirements, design artefacts, and implementation elements are generated and have to be validated.

Assured AI Workflow An AI workflow consists of a sequence of predefined, containerised, processing actions, where each step consumes an input and produces an output and may have third-party dependencies [20]. These steps correspond either to *deterministic tool executions*, which can be *assured by design*, or to *non-deterministic executions* by an AI or a human actor.

A key advantage of this architecture is that multi-step processes which include branching, iterations, and conditional executions can be decomposed into smaller, self-contained actions. Each action can then be individually analysed and equipped with an appropriate assurance layer according to its characteristics. Deterministic components may rely on correctness guarantees, whereas non-deterministic components require surrounding verification and correction mechanisms. As a result, assurance is achieved locally at the level of the containerised processing action while propagating globally throughout the overall workflow. Representative application scenarios include *decision support workflows*, where data from multiple sources is analysed and transformed into feasible automatised actions that must be consistent, explainable, and aligned with predefined objectives; *automated business processes*, where documents, requests, or transactions are processed and linked into traceability graph structures across several workflow steps, and each intermediate result has to be validated for correctness and completeness; or *graph-based network orchestration workflows*, where network topologies and their dependencies are modelled as graph structures and processed through coordinated workflow steps for monitoring, traffic analysis, fault diagnosis, decision-making, and automated reconfiguration.

Assured AI Agent Agents are not a new concept for automation [10, 11]. Traditional agents have long been used to coordinate tasks, invoke services, and interact with software systems. Thereby, such systems were typically based on predefined rules, static workflows, or human executors who supervised decisions, controlled processes, and delegated tasks. Yet, advances in modern AI models, i.e., LLMs, have substantially expanded these capabilities through training and fine-tuning for agentic behaviour such as reasoning, planning, execution, and self-evaluation [10, 11]. Thus, modern AI agents are able to pursue complex tasks with a high degree of autonomy. They can decompose tasks into subtasks and either delegate them to subsystems through APIs, or iteratively process and refine them. Hence, from an assurance perspective, the initial input and the final output returned to the user can be validated and, if needed, repaired. Still, the internal behaviour of the agent remains largely non-deterministic as it independently decides which actions to perform, in which order to call them, and when to terminate.

Principally, an agent may interact with two categories of callable components. First, it may invoke systems (functional tools), or subordinate agents whose outputs become part of the ongoing reasoning process. Whenever these components are deterministic and verified according to predefined constraints, they can be regarded as *assured by design*. This guarantee does not hold for newly composed runtime behaviours, i.e., dynamically generated code. Second, an agent may call dedicated assurance mechanisms such as validation or repair functions, in order to assess and improve its own intermediate outputs during execution. Nevertheless, the decision whether such assurance mechanisms are actually used, and whether they are invoked with semantically appropriate parameters, is itself subject to the agent's non-deterministic control policy. While, e.g., the syntactic correctness of API-calls can be checked, their semantic adequacy, completeness, and suitability cannot be guaranteed. Furthermore, highly autonomous agents operating with large or unbounded planning horizons raise fundamental verification challenges. It may be impossible to determine beforehand whether an agent will terminate, whether it will achieve its objective, or whether all intermediate behaviours remain within acceptable bounds. For this reason, the design of correct and trustworthy agent systems requires careful restriction of permissions, bounded execution environments, monitoring, and the selective exposure of tools and resources. Graph transition systems or probabilistic Petri nets could be used to model these kinds of systems to assure that all paths an agent takes are verified traces.

Representative application scenarios include *IT support agents*, where user requests are analysed and resolved through coordinated access to documentation and operational tools, *software engineering*

agents, where code, tests, and deployment artefacts are generated and validated across development pipelines, or *security agents*, where threats are detected, investigated, and mitigated through autonomous interaction with monitoring and response systems.

3. Example Case

Using LLMs to generate software models of various kinds (e.g. [21, 22, 23]) or to transform such software models for various purposes (e.g. [24, 25]) is a rapidly growing field of research. We use a small example from this domain of software modelling to illustrate the AI usage modes presented above and discuss how these could be assured using graph transformation techniques.

3.1. Requirements and Architectural Model for Sustainable Farming Planner

Our illustrating example concerns *The Sustainable Farming Planner*, which aligns with the United Nations Sustainable Development Goals, specifically SDG 2 (Zero Hunger) and SDG 13 (Climate Action) [26]. We used the following prompt to generate a high-level architectural model:

> From the requirements below, create UML diagrams 1-5 to model the application as a service-oriented system:
>
> 1. a use case diagram for the app, including external services
>
> 2. a sequence diagrams modelling the component interactions
>
> 3. a component diagram identifying components, interfaces, implementation and usage relations
>
> After each diagram, add a traceability table listing the main elements of the diagram and their mapping to elements of the requirements and earlier diagrams.
>
> Output must be in plantUML notation. Diagrams must be consistent with each other and reflect the given requirements.
>
> Sustainable Farming Planner
>
> SDG 2: Zero Hunger, SDG 13: Climate Action
>
> • Small farmers in Andhra Pradesh
>
> • Get crop recommendations
>
> • View soil conditions and rainfall predictions
>
> • Agricultural officers
>
> • View district-level crop patterns and risk alerts
>
> Scenario for “Get crop recommendations”
>
> 1. Farmer selects land location on map.
>
> 2. Backend retrieves data from NASA Soil Moisture API and OpenWeather API.
>
> 3. App suggests best crops for the next season based on sustainability metrics.
>
> 4. Farmer receives reminders for sowing, irrigation, and fertilization schedules.
>
> 5. Officers get analytics dashboards on crop health trends.

When calling the LLM[12], it generates the following diagrams[13]. In addition, it generates several traceability tables. Two of them show the traces between the requirements for the “Get crop recommendations” scenario and the sequence diagram, as well as the sequence participants and the component diagram. We integrated these two tables into Table 1.

As shown in Figures 3–5, the UML diagrams model the application as a service-oriented system, covering use cases, interactions, and components.

[12]The free GPT 5.3 on 22 April 2026 https://chatgpt.com/share/69df0e68-d508-832d-b441-086c0f3fd2ee
[13]Rendered by the plantUML editor https://www.plantuml.com

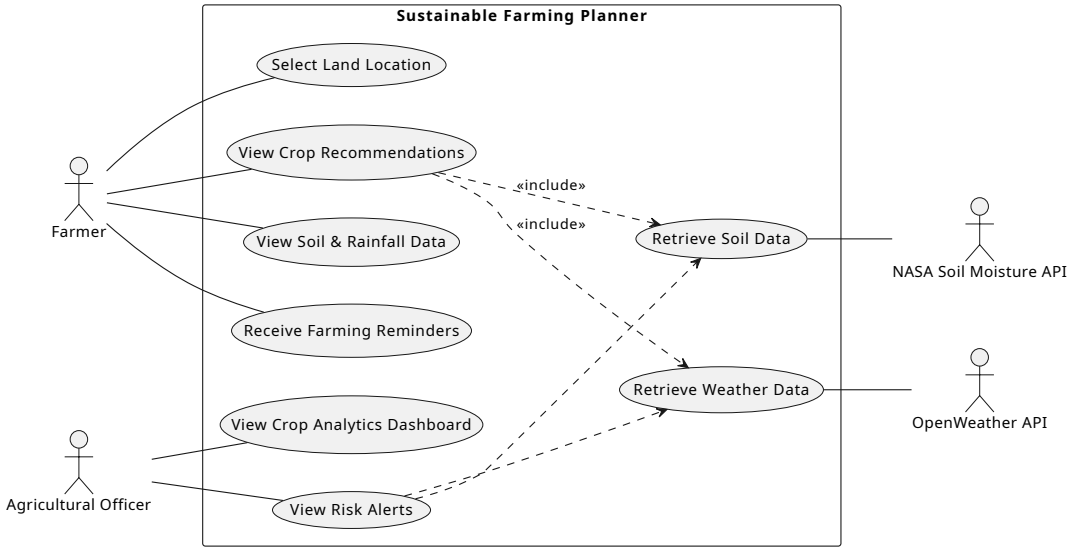


Figure 3: Use case diagram for the app with the top level use cases as given in the requirements

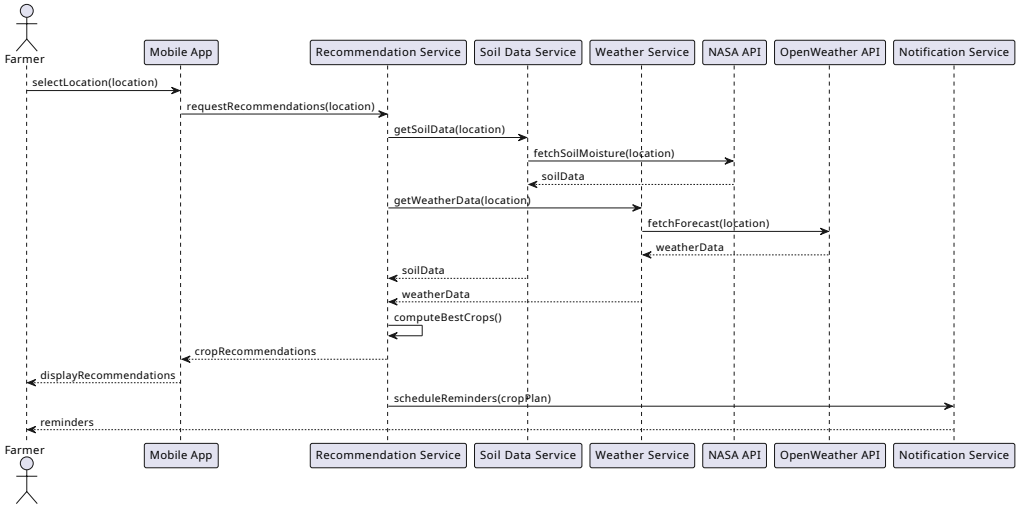


Figure 4: Sequence diagram(s) modelling the main scenario at component level

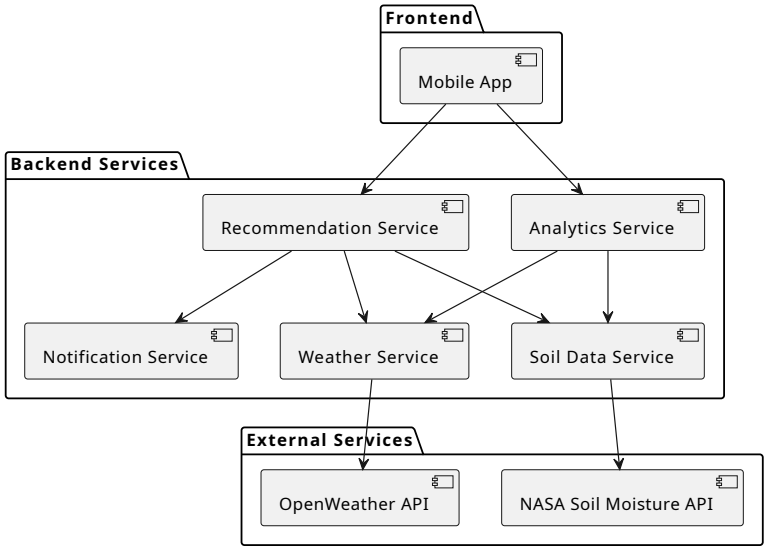


Figure 5: Component diagram identifying components and their use relations

Table 1
Sequence to component to requirement traceability

Sequence Participant / Message	Component	Requirement
Farmer /select Location	–	Step 1
MobileApp /requestRecommendations	MobileApp	–
Recommendation Service / getSoilData	Recommendation Service	Step 3
Soil Data Service / fetchSoilMoisture	Soil Data Service	–
Weather Service / getWeatherData	Weather Service	–
NASA API	NASA Soil Moisture API	Step 2
Open Weather API / fetchForecast	Open Weather API	Step 2
Notification Service / scheduleReminders	Notification Service	Step 4

3.2. Verification Modes and Levels

When AI is used to generate software architecture models, both the input requirements and the generated models have to adhere to certain consistency conditions, design guidelines, and semantic requirements.

In an *AI recommendation system*, both the input to the LLM and its output can be validated, possibly repeating the LLM call if the output check identifies any errors (see Figure 2). In our example, the input is provided by the prompt in Section 3.1. This includes a description of the application's requirements and usage scenarios, as well as the structure of a design model. As this model will consist of several diagrams, the prompt also includes constraints related to correctness, consistency and trustworthiness. Output validation involves a syntax check to verify that all diagrams are correctly written in PlantUML notation, a consistency check of all diagrams in the design model, and a semantic check to assure that requirements are clearly expressed in the model. A language parser can perform the syntax check depending on the output format. The consistency between the sequence and component diagrams can be formally verified by applying graph transformation rules [27, 28]. With the help of a tracing table, this check can be carried out using a triple graph grammar [29, 30], for example, to verify the relationship between the models. Semantic correctness can be validated by processing the natural language requirements, which may involve AI-based assurance.

Rather than generating all diagrams at once with one large, complex call, it may be more effective to generate them one by one. This leads to an assured *AI workflow* involving several LLM calls and feedback loops, all of which are guarded by input, output and intermediate filters (see Figure 2). Our AI-based approach to architectural modelling allows the diagrams to be produced in order, with later diagrams depending on earlier ones, but not vice versa. The consistency checks can be broken down into incremental checks of the most recent diagrams against all previous ones.

To allow for adaptive granularity and iterations, for example, depending on the size and complexity of the model, we have to replace the rigidly predefined AI workflow with an *AI agent* (see Figure 2). An AI agent can autonomously decide how to break down the problem and call LLMs and verification steps, which may be guarded by mandatory input and output verification steps. In our scenario, we could add a requirement for a strictly layered architecture, without cyclic or transitive dependencies. This can easily be expressed using graph constraints [28, 31] to specify forbidden patterns over component diagrams or a graph grammar [27, 32] that defines a language of graphs without cycles, for example. However, since the structure of the component diagram is determined by the sequence diagram, changing the component diagram would require changing the sequence diagram, as well as potentially the use case diagram. An alternative approach would be to apply a rule-based repair action [33, 34, 35] to the component diagram and then regenerate the other diagrams. Similarly, if architectural constraints are important, generating the sequence diagram first may not be the best strategy. In this case, a modelling agent could make LLM calls to generate or regenerate the diagrams in a different order and apply suitable rule sets to filter and check the outputs.

In all three scenarios, AI is used as a modelling tool in application development, albeit in different modes. More broadly, AI can occur at all levels of the modelling hierarchy. For example, at the meta-level, consistency and repair rules can be generated using an AI workflow to generate graph transformation

rules. In [36], create rules form part of the application and the LLM is used as a tool to generate an executable model. At the instance level, AI workflows or agents can be components of the application that are invoked at runtime to explain and justify crop recommendations to farmers, for example.

4. Related Work

The assurance of AI is a rapidly growing field of research. There are position papers for the specific use case of software engineering [1] or on the integration of AI with formal methods [7]; first literature surveys on the topic appeared [18] and "Program Verification" is one of the categories of the literature database on *Agentic Software Engineering* curated and maintained by the PurCL group[14]. We categorise the approaches for AI assurance into four classes:

1. *Input guardrails for AI:* Mechanisms that inspect, constrain, improve or retrieve inputs before processing to reduce the risk of unsafe, invalid, or out-of-distribution data.
2. *AI for AI:* The use of AI to monitor, assess, constrain, or improve other AI (systems). Although some are calling it a verification approach, it is actually a plausibility check.
3. *Testing for AI:* The empirical validation of AI systems through test cases, benchmarks, simulations, or scenario-based evaluation to identify failures under selected conditions.
4. *Formal methods for AI:* Mathematically rigorous techniques that analyse or verify whether an AI system satisfies formally specified properties, such as liveness or correctness.

Input Assurance Techniques to assure input accuracy are presented in papers such as [12, 13, 37, 38]. In [12], the concept of safety prompts is presented. These prompts are prepended to a user query before it is processed by an LLM. This work builds on earlier prompt tuning techniques presented in [39, 40, 41]. [37] shows that the output structure described in a prompt affects the correctness of the result; for example, XML formats tend to produce incorrect outputs, whereas triples enhance correctness. [38] lists Retrieval-Augmented Generation (RAG) approaches for improving the correctness of AI responses. Building on this paradigm, [13] introduces *LLatrieval*, a retrieval framework in which the AI refines its own input.

AI-based Output Assurance In AI-based assurance, the dominant approach is the *LLM-as-a-judge* paradigm, which enables the evaluation of AI outputs that cannot be formalised or are too computationally expensive to assess. Hence, it provides plausibility arguments that support the quality of an output. Key works in this field include [42, 43, 44]. This kind of approach has the advantage of being applicable to structured, semi-structured and unstructured outputs. Furthermore, it enables the evaluation of results with multiple valid answers spanning different levels of abstraction, such as in the generation of certain models (e.g., BPMN[15]).

Test-based Output Assurance Testing is a widely used technique for ensuring the quality of AI outputs [14, 45, 15]. [14] proposes *AITEST*, a test-driven workflow in which black-box AI models are evaluated using test cases derived from a baseline model. A related approach, *CodeT*, is presented in [15]. This approach involves executing code generated by pre-trained AI models against predefined test cases to assess correctness and quality. In addition to functional correctness, the evaluation considers metrics such as execution time and pass@k. Overall, these approaches require structured output (e.g., a formal language) and are tied to concrete cases. They are not universally applicable to similar cases (as test cases which are logically coupled to a method under test).

Formal Verification-based Output Assurance Various formal verification techniques have been used to assure the correctness of AI-generated content. Research results include the use of formal verification techniques for verifying LLM-generated code of specific (programming) languages and the behaviour of AI agents, as well as translating natural language requirements into a formal language

[14] https://github.com/PurCL/ASE; the category contains 44 entries as of April 24, 2026.
[15] https://www.bpmn.org/

for verification purposes. LLM-generated code is verified, for example, in [46, 16, 47]. [46] generates C programs and instructs the LLM to generate the code in such a way that it is amenable to formal verification, which is subsequently performed in Rocq (also supported by an LLM). [16] develops a controlled but natural-language-like DSL to express the desired properties of programs, in Ansible, a language for managing remote systems. Statements from the DSL and from LLM-generated Ansible programs are then translated into a *state calculus* that enables equivalence to be formally proven between the two. [47] uses bounded model checking (in Alive2) of the original code and an LLM-generated vectorised version to prove the equivalence; failed proofs are used to re-query the LLM. To verify the system behaviour of AI agents, [48] monitors the running system, learns a Markov Decision Process from the run and applies model checking. Additionally, LLMs have also been used to generate input for formal verification techniques from natural language text. [17] extracts basic mathematical content, e.g., algebraic computations, from LLM answers and formally verifies them using Isabelle. While the examples in [17] are rather simple, Isabelle has also been used to formalise graphs and even graph transformations [49]. However, this is very demanding.

In conclusion, we have not found any formal AI assurance techniques that are particularly suited to verifying structured, language-agnostic LLM output and its dynamic changes. This is what we envisage for graphs and graph transformation techniques. Henshin and Groove are two graph transformation–based tools that can hide the inherent complexity of graph transformation to some extent, making them ideal for verifying structured, language-agnostic LLM results and their dynamic changes.

5. Conclusion

As many AI techniques are considered black-box systems, efforts to provide assurance tend to focus on the quality of inputs and outputs, and the execution paths within AI-based workflows and agents. We reviewed the key assurance modes for AI usage and demonstrated them using a simple software modelling example. We also discussed how graph transformation techniques can be used to support and enhance AI assurance. In the future, we plan to develop graph transformation-based AI assurance techniques for all three of the AI usage modes outlined in this paper. This will involve providing graph transformation techniques via a user-friendly API, and applying these techniques to several example applications, including the one presented in this paper. We will also conduct evaluations to understand the potential and limitations of these techniques. Overall, we will investigate how graph transformation techniques can be advantageously used to assure the effective use of AI in software engineering.

Declaration on Generative AI During the preparation of this work, the authors used GPT-5.4, Gemini and Deepl.com for grammar and spelling checks and smaller text recommendations and improvements. Figure 1 was generated with gpt-image-2. After using these tools, the authors reviewed and edited the content as needed and take responsibility for the publication's content.

References

[1] N. Alshahwan, M. Harman, et al., Assured LLM-Based Software Engineering, in: InteNSE 2024, 2024, pp. 7–12.

[2] T. Davenport, R. Kalakota, The potential for artificial intelligence in healthcare, Future healthcare journal 6 (2019) 94–98.

[3] Q. Zhang, J. Lu, Y. Jin, Artificial intelligence in recommender systems, Complex & intelligent systems 7 (2021) 439–457.

[4] A. Dave, H. Bang, et al., A framework for effective ai recommendations in cyber-physical-human systems, IEEE Control. Syst. Lett. 8 (2024) 1379–1384.

[5] L. S. Hofmann et al., Toward safeguarding genai with graph transformation: Defining an exchange format in json, in: GCM 2025, Open Publishing Association, 2025, pp. 10–11.

[6] I. Bouzenia, P. Devanbu, M. Pradel, RepairAgent: An Autonomous, LLM-Based Agent for Program Repair, in: ICSE 2025, 2025, pp. 2188–2200.

[7] Y. Zhang, Y. Cai, et al., Position: Trustworthy AI Agents Require the Integration of Large Language Models and Formal Methods, in: ICML 2025, OpenReview.net, 2025.
[8] D. Roy, M. Dutta, A systematic review and research perspective on recommender systems, J. Big Data 9 (2022) 59.
[9] L. Wu, Z. Zheng, et al., A survey on large language models for recommendation, World Wide Web 27 (2024) 60.
[10] N. Shinn, F. Cassano, et al., Reflexion: language agents with verbal reinforcement learning, in: NeurIPS 2023, 2023.
[11] J. Yang, C. E. Jimenez, et al., SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering, in: NeurIPS 2024, 2024.
[12] C. Zheng, F. Yin, et al., On prompt-driven safeguarding for large language models, arXiv preprint arXiv:2401.18018 (2024).
[13] X. Li, C. Zhu, et al., Llatrieval: Llm-verified retrieval for verifiable generation, in: NAACL 2024, 2024, pp. 5453–5471.
[14] A. Aggarwal, S. Shaikh, et al., Testing Framework for Black-box AI Models, in: ICSE-Companion 2021, 2021, pp. 81–84.
[15] B. Chen, F. Zhang, et al., CodeT: Code Generation with Generated Tests, in: ICLR 2023, OpenReview.net, 2023.
[16] A. Councilman, D. Fu, et al., Towards Formal Verification of LLM-Generated Code from Natural Language Prompts, CoRR abs/2507.13290 (2025).
[17] J. P. Zhou, C. Staats, et al., Don't Trust: Verify – Grounding LLM Quantitative Reasoning with Autoformalization, in: ICLR 2024, OpenReview.net, 2024.
[18] G. Dolcetti, E. Iotti, A dual perspective review on large language models and code verification, Frontiers Comput. Sci. 7 (2025).
[19] M. Felderer, R. Ramler, Quality assurance for ai-based systems: Overview and challenges, in: International Conference on Software Quality, Springer, 2021, pp. 33–42.
[20] W. Brewer, A. Gainaru, et al., AI-coupled HPC workflow applications, middleware and performance, arXiv preprint arXiv:2406.14315 (2024).
[21] A. H. et al., Generating Software Architecture Description from Source Code using Reverse Engineering and LLMs, in: MODELS-C 2025, IEEE, 2025, pp. 566–575.
[22] B. Chen, O. Wei, et al., Accurate and Consistent Graph Model Generation from Text with Large Language Models, in: MODELS 2025, IEEE, 2025, pp. 130–141.
[23] J. S. Mercado, Q. Ma, et al., CLERK: A Companion Large Language Model Expert for modeling Regulatory Knowledge, Data & Knowledge Engineering 164 (2026) 102578.
[24] Z. Hachm, T. L. Calvar, et al., Towards llm agents for model-based engineering: A case in transformation selection, in: MODELS-C 2025, 2025, pp. 421–431.
[25] D. Dao, A. Bucaioni, A. Cicchetti, Learning to Transform: Evaluating LLMs on Model Transformation by Example, in: MODELS-C 2025, 2025, pp. 576–585.
[26] United Nations, The 17 sustainable development goals, 2015. URL: https://sdgs.un.org/goals.
[27] H. Ehrig, K. Ehrig, et al., Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2006.
[28] R. Heckel, G. Taentzer, Graph Transformation for Software Engineers – With Applications to Model-Based Development and Domain-Specific Language Engineering, Springer, 2020.
[29] A. Schürr, Specification of Graph Translators with Triple Graph Grammars, in: WG '94, Springer, 1994, pp. 151–163.
[30] A. Anjorin, E. Leblebici, A. Schürr, 20 Years of Triple Graph Grammars: A Roadmap for Future Research, Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 73 (2015).
[31] A. Habel, K. Pennemann, Correctness of high-level transformation systems relative to nested conditions, Math. Struct. Comput. Sci. 19 (2009) 245–296.
[32] G. Rozenberg, Handbook of graph grammars and computing by graph transformation, volume 1, World scientific, 1997.
[33] G. Taentzer, M. Ohrndorf, et al., Change-preserving model repair, in: FASE 2017, 2017, pp. 283–299.

[34] L. Iovino, A. Barriga, et al., Model repair with quality-based reinforcement learning, J. Object Technol. 19 (2020).
[35] A. Lauer, J. Kosiol, G. Taentzer, Empowering model repair: a rule-based approach to graph repair without side effects – extended version, Innov. Syst. Softw. Eng. 20 (2024) 597–618.
[36] R. Heckel, M. Saad, Can AI Make the Rules? A Pipeline for Generating Graph Transformation Systems, in: MODELSWARD 2026, SCITEPRESS, 2026, pp. 5–14.
[37] Y. Liu, D. Li, et al., Are LLMs good at structured outputs? A benchmark for evaluating structured output capabilities in LLMs, Inf. Process. Manag. 61 (2024).
[38] M. Arslan, H. Ghanem, et al., A Survey on RAG with LLMs, Procedia computer science 246 (2024) 3781–3790.
[39] B. Lester, R. Al-Rfou, N. Constant, The power of scale for parameter-efficient prompt tuning, in: EMNLP2021, 2021, pp. 3045–3059.
[40] Y. Zeng, H. Lin, et al., How johnny can persuade llms to jailbreak them: Rethinking persuasion to challenge ai safety by humanizing llms, in: NAACL, 2024, pp. 14322–14350.
[41] X. L. Li, P. Liang, Prefix-tuning: Optimizing continuous prompts for generation, in: ACL/IJCNLP 2021, 2021, pp. 4582–4597.
[42] S. Tan, S. Zhuang, et al., Judgebench: A benchmark for evaluating llm-based judges, arXiv preprint arXiv:2410.12784 (2024).
[43] A. Bavaresco, R. Bernardi, et al., LLMs instead of Human Judges? A Large Scale Empirical Study across 20 NLP Evaluation Tasks, in: ACL (2) 2025, 2025, pp. 238–255.
[44] Z. Xiang, L. Zheng, et al., Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning, arXiv preprint arXiv:2406.09187 (2024).
[45] H. Inan, K. Upasani, et al., Llama Guard: LLM-based Input-Output Safeguard for Human-AI Conversations, CoRR abs/2312.06674 (2023).
[46] P. Mukherjee, M. Lu, B. Delaware, LLM-Assisted Synthesis of High-Assurance C Programs, in: ASE 2025, IEEE, 2025, p. 3108.
[47] J. Taneja, A. Laird, et al., LLM-Vectorizer: LLM-Based Verified Loop Vectorizer, in: CGO 2025, ACM, 2025, pp. 137–149.
[48] R. Koohestani, AgentGuard: Runtime Verification of AI Agents, in: ASEW 2025, IEEE, 2025, pp. 74–77.
[49] R. Söldner, D. Plump, Formalising the double-pushout approach to graph transformation, Logical Methods in Computer Science 20 (2024).