Teaching Governance Literacy for AI-Assisted Software Engineering: A Practitioner-Informed Framework for Information Systems Curricula

Rishav Bhandari[1]

[1]Independent Researcher, London, United Kingdom

Abstract

Generative AI has become routine in enterprise software engineering. Code assistants now draft functions, scaffold deployment scripts, write documentation, and propose architectural patterns on most delivery teams the author has worked with. The productivity gains are measurable. The educational gap that has opened beside them is less measurable but, for information systems programmes, more urgent. Graduates entering AI-assisted workplaces in 2026 are usually fluent prompters of code assistants but rarely prepared for the specific governance work that AI-assisted development now demands of them in the first six months on the job. Article 4 of the EU AI Act, in force since February 2025, makes this gap a regulatory matter rather than a soft recommendation. Drawing on practitioner reflection across UK and Indian enterprise teams between 2018 and 2025, this paper proposes a three-layer framework: architectural ownership, structured validation, and operational observability, and maps each layer to concrete curriculum proposals for IS and software engineering programmes. The argument is that governance literacy belongs across the IS curriculum rather than in a single ethics module and that workshops focused on co-creating new ways of IS education are an appropriate venue for that argument to be tested.

Keywords

AI literacy, information systems education, generative AI, software engineering pedagogy, AI governance, curriculum design

1. Introduction

Something changed in undergraduate software engineering education sometime around late 2022, even if the syllabi have not yet caught up. Students started arriving in their final-year projects with code that was largely generated by Copilot or ChatGPT, and academics started arguing about whether to ban these tools, tolerate them, or teach with them [20]. Three years on, the industry question is settled: code assistants are part of the toolchain on most teams. The question that remains open, especially in the venues where IS curricula get debated, is what universities should do about it.

Two empirical points frame the picture. Peng and colleagues, in the most-cited Copilot productivity study, found programmers using the tool finished a server task about 56% faster than those without it [6]. Practitioner observation in the author's recent work on a major energy-sector engagement in 2024–25 broadly matched the direction of that finding, though the magnitude was smaller — closer to 30% across multiple squads, on a wider mix of tasks. On the other side, Pearce and colleagues' security evaluation found that roughly 40% of Copilot completions in a battery of security-relevant scenarios contained known vulnerabilities [8]. The defects were rarely obvious negligence; the suggestions usually looked plausible. The productivity headline and the security tail belong to the same story.

So the technical case is reasonably settled: AI assistance helps, AI assistance hurts, and both effects are large. The educational case is not. IS graduates entering enterprise workplaces are usually competent prompters of code assistants. They are not usually prepared to challenge what comes back, surface architectural decisions buried inside generated code, or argue with a tech lead about whether to accept a particular suggestion into production. This competency gap is where the paper starts.

It develops in five steps. Section 2 surveys related work, deliberately placing IS education first rather than treating it as an afterthought. Section 3 sets out three specific competencies that graduates

NWISEd 2026: Workshop on Co-Creating New Ways of Information Systems Education, July 1–3, 2026, Swansea, United Kingdom
rishavbhandari111@gmail.com (R. Bhandari)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

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

published 2026-11-19

CEUR-WS.org/Vol-4333/paper3.pdf

entering AI-assisted teams need and typically do not have. Section 4 explains the reflective practitioner methodology used to develop the argument. Sections 5 and 6 present a three-layer framework that doubles as a curriculum scaffold, with concrete module-level proposals for IS programmes. Sections 7 and 8 close with a short discussion of what educators can try this semester, plus limitations and future work.

The contribution is targeted. This is not a paper about AI governance in general — Khan and colleagues have already published a systematic review of more than sixty frameworks on that topic [4]. It is not a paper about AI literacy in general either — Gu and Ericson have already covered that ground for higher education [12]. The narrower argument here is that IS and software engineering programmes can be specific about what governance literacy means in their own discipline, and can teach it without bolting on a separate ethics module.

2. Related Work

The paper sits at the intersection of three literatures. They are sketched here in the order of their relevance to the workshop venue: information systems and computing education first, AI governance second, productivity and risk in AI-assisted development third.

2.1. Information Systems Education and the AI Literacy Question

The post-2022 wave of AI literacy research in computing education has moved unusually fast. Becker and colleagues' SIGCSE 2023 paper argued early on that introductory programming pedagogy would have to adapt to a world in which students could generate working code without writing it [19]. Denny and colleagues, writing in *Communications of the ACM*, broadened the argument across computing education more generally, surveying both the threats to existing pedagogical models and a range of legitimate teaching responses [20]. Prather and colleagues' ICER 2024 study followed novice programmers using generative AI over time and reported a "widening gap" — strong students benefited, weaker students often did not [21]. Together, these three papers frame the empirical core of what computing educators currently know about teaching in the presence of code assistants.

Beyond computing education proper, Tadimalla and Maher have proposed that AI literacy is best understood as a layered construct: a general capability layer (what tools do and how to use them), a sociotechnical layer (how the tools interact with people and systems), and an ethical or governance layer (what responsible use looks like in context) [13]. Gu and Ericson's review of K-12 and higher education AI literacy research confirmed both the rapid growth of the field since 2023 and the relative scarcity of governance-layer studies compared to general-capability work [12]. Eboh has argued more pointedly that AI governance education needs to be interdisciplinary by design, pulling together technical, policy, ethical, and organisational threads [11]. Kalantzis and Cope, writing on literacy more broadly, treat AI as the kind of technological shift that requires a redefinition of what literacy itself now means [15].

The institutional response has lagged the empirical research. Tong and colleagues' content analysis of university policies on generative AI found that institutions have spent most of their energy on academic integrity questions (can students use AI in coursework?) rather than on professional preparation (can graduates govern AI in workplaces?) [14]. The asymmetry matters. The paper's argument depends on it.

2.2. AI Governance Frameworks

The standards landscape is more mature than the educational one. The US National Institute of Standards and Technology's AI Risk Management Framework, published in 2023, organises risk management into four functions — Govern, Map, Measure, Manage — and is deliberately voluntary and sector-neutral [1]. The 2024 Generative AI Profile of the same framework extends it to generative systems specifically [2]. Both are useful as conceptual scaffolds but are explicit that they do not prescribe engineering practices. The translation problem — how should teams actually change what they do? — is left open.

The EU Artificial Intelligence Act came into force in August 2024 [3]. Most of its provisions apply from August 2026, but Article 4 — which requires staff involved in operating AI systems to have sufficient AI literacy — has applied since February 2025. That date matters here. Article 4 is, to the author's knowledge, the first major regulatory instrument to treat workforce AI literacy as a compliance matter rather than a recommendation. For universities producing graduates who will deploy or operate AI systems, that timing is consequential.

Outside formal regulation, the principles-level consensus on responsible AI is broad and well-mapped. Jobin and colleagues reviewed 84 ethics guidelines and identified five recurring themes: transparency, justice, non-maleficence, responsibility, and privacy [5]. Khan and colleagues' systematic review of more than sixty frameworks reached substantively similar conclusions [4]. Both reviews flag the same difficulty: agreement on principles is far easier than translation into operational engineering or educational practice. That translation problem is where this paper starts.

2.3. Productivity and Risk in AI-Assisted Software Development

Research on Copilot and similar tools has expanded quickly since 2022. Peng and colleagues' randomised study reported the ~56% productivity gain mentioned earlier [6]. Ziegler and colleagues, writing from inside GitHub, later reported gains across a broader mix of tasks and developer profiles [7]. Between them, these studies make a defensible case that AI assistance accelerates routine coding.

The risk side is more uncomfortable. Pearce and colleagues evaluated security in Copilot-generated code and found about 40% of completions in security-relevant scenarios contained known vulnerabilities [8]. Stray and colleagues' longitudinal mixed-methods study followed developers using Copilot over two years. While their commit-based activity metrics did not show statistically significant productivity gains for Copilot users, their interview data documents substantial verification overhead and what the authors describe as trust-calibration issues — developers spending time disambiguating whether to trust a particular suggestion [10]. For the present argument, the qualitative finding is the relevant one: verification effort is real even where it does not show up in headline productivity measures. Bacchelli and Bird's older but still-influential study of modern code review noted that even before AI assistance, code review's effectiveness was bounded by the quality of human attention developers could bring to it [22]. Generative AI does not relax that constraint; it tightens it.

A more recent conceptual contribution comes from Kohl and Carro, who argue that software engineering as a discipline needs to reorient itself around orchestration and verification as code becomes cheap to generate [9]. The framework proposed in Section 5 is sympathetic to that diagnosis, though it differs in emphasis and in its educational focus.

Across the three literatures, the picture is consistent. Educational responses to AI in computing have moved quickly but have focused more on general capability and academic integrity than on governance literacy. Governance frameworks have reached principles-level maturity but leave operational translation open. And practitioner reality features substantial productivity gains alongside under-acknowledged verification costs. The competency gap the rest of the paper addresses lies in the intersection of those three.

3. The Competency Gap: Three Things Graduates Need to Be Able to Do

What follows is not a comprehensive taxonomy of AI-related risks in software development; Section 2 already pointed to several. It is, instead, a narrower account of three specific competencies that graduates entering enterprise AI-assisted teams are typically expected to demonstrate within their first six months, and that they are typically not prepared for by their undergraduate programmes. The selection criterion is that conventional engineering controls handle these competencies poorly, so they fall on the human in the loop — usually the most junior human in the loop.

3.1. Reading Generated Code as If You Wrote It

The first competency is unfamiliar in its specifics even though the underlying skill — explaining your own code — has always been part of engineering practice. When a function comes out of a code assistant, the developer who accepted it often cannot say why a particular library was chosen, why one implementation pattern was used over another, or what trade-offs the model implicitly made. The code reads fluently; the rationale is missing.

This becomes expensive during incident response. When a production system fails, engineering teams typically have to reconstruct the decisions that contributed to the failure. If large sections of code were generated and the rationale was never written down, that reconstruction takes longer and produces less defensible conclusions. Missing design rationale stays invisible until something breaks, and then it is suddenly central.

Graduates rarely receive structured practice in this skill. The standard undergraduate exercise is to write your own code and defend your own design choices. The post-Copilot equivalent — accept a generated function, then defend the design choices made by a probabilistic model on your behalf — is a different cognitive task. It is at least as important. In the author's mentorship experience with early-career engineers, it is also the area where they most often need supervision in their first year.

3.2. Recognising Architectural Decisions When They Arrive Disguised as Autocomplete

The second competency is architectural. Code assistants tend to suggest libraries, frameworks, and patterns that occur frequently in their training data. Many of these suggestions are sensible defaults. But they can also quietly shape architectural decisions that were never reviewed at the system level. A team building a new service may receive AI-generated suggestions for authentication libraries, logging frameworks, and caching layers within a single afternoon. Each suggestion looks reasonable in isolation. Across a sprint or two, the architecture of the service can drift toward the model's preferences rather than the team's deliberate choices.

The problem is not that any individual suggestion is wrong. The problem is that architectural decision-making slides away from intentional human review without anyone noticing. In small systems this often does not matter. In enterprise environments — where dependency management, licence compliance, and architectural consistency carry weight — it matters more.

The competency a graduate needs is recognition: the ability to notice that an architectural decision is being made even when it arrives in the unceremonious form of an IDE suggestion. This is a different skill from choosing a framework on a clean greenfield project. It is closer to the skill a clinician needs to notice a symptom in passing rather than diagnose it in a textbook. It is taught best by repeated exposure, with feedback.

3.3. Owning the Outcome When You Did Not Write the Code

The third competency is the most uncomfortable. When AI-generated code contributes to a production incident, accountability becomes contested in ways that conventional engineering practice handles awkwardly. Responsibility nominally rests with the developer who accepted and shipped the code. That logic still applies. But it now shares the room with several other parties: the tool that proposed the suggestion, the vendor who provided the tool, the architectural team that approved its use, and the procurement function that licensed it. Most vendors disclaim responsibility for the substance of generated outputs. Architectural approvers are usually not close to the specific incident.

For a junior engineer, the question is whether to accept this accountability cleanly or to deflect it onto the tool. Both options are available. The difference between them is professional. Mentorship work with early-career engineers suggests this is a question many of them have not encountered before in any structured form, and that the way they answer it in their first year tends to harden into a pattern. Universities have an opportunity here that they are not yet using.

The three competencies reinforce each other. Limited transparency makes incident investigation slower. Hidden architectural dependencies increase the chance of incidents. Unclear accountability stretches response timelines and creates disagreement about who owns the fix. Any curriculum proposal worth taking seriously has to address all three. The next section sets out the methodology used to develop one.

4. Methodology

The approach taken here is reflective practitioner inquiry, in the tradition of Schön's work on how professionals build knowledge through engagement with their own practice [16]. It is a qualitative, interpretive approach that draws systematically on direct involvement in the phenomenon being studied, and it is well established in information systems research and in studies of complex sociotechnical environments where controlled experimentation is impractical.

The author has worked in enterprise software engineering across the UK and India since 2018, in roles ranging from full-stack development to technical lead, across sectors including energy, manufacturing, media, pharmaceuticals, and identity and access management. The observations behind this paper come from direct involvement in AI-assisted delivery during 2024 and 2025, when generative tools moved from experimental adoption to routine use on the teams concerned. They are supplemented by mentorship engagement with several hundred early-career engineers across multiple cohorts. That mentorship informs the curriculum argument in Section 6; it does not stand in for empirical educational data, which this paper does not present.

A few methodological commitments shape what follows. Observations are presented anonymised: no client is named, no individual identified, and no specific commercial incident described in identifiable detail. Where a sector is mentioned, it is at the level of abstraction sufficient for the argument and no further. The framework in Section 5 is offered as an analytical synthesis rather than an empirically validated artefact — a hypothesis for further evaluation, not a finished product. The analysis triangulates practitioner observation with the published literature surveyed in Section 2, in the situated-learning tradition that treats workplace knowledge as continuous with formal scholarship rather than separate from it [17].

The limitations are real. Observations come from a particular professional trajectory and a particular set of sectors. The framework has not been formally evaluated in a controlled organisational or educational setting. Both points are acknowledged here and shape the future work outlined in Section 8. Position papers in the NWISEd tradition are explicitly described in the workshop call as papers intended to provoke discussion; the framing here is consistent with that purpose.

5. A Three-Layer Framework

The framework presented here organises the response to the three competencies of Section 3 into three layers of practice. Each layer addresses one competency; each carries direct curriculum implications developed in Section 6. The structure is summarised in Table 1.

5.1. Layer One: Architectural Ownership

The first layer concerns who owns architectural decisions when generative tools are in the workflow. The principle is simple: architectural ownership cannot be delegated to a model. A named human, normally a senior engineer or architect, remains explicitly accountable for the architectural surface of a system — choice of libraries, frameworks, integration patterns, and dependency boundaries.

In practice, AI-assisted teams should be required to surface architectural decisions made during code generation and route them through a defined review path. When a code assistant recommends a particular library for a new feature, that recommendation belongs in the architectural pipeline rather than being silently accepted by whichever developer triggered it. The mechanism does not need to be

heavy. A pull request label, a short architectural decision record, or a justification field in the commit message can each carry the weight of accountability if applied consistently. The point is that the decision becomes visible, not that the process becomes bureaucratic.

For educators, the implication is that final-year IS or software engineering students should leave their programmes able to recognise an architectural decision in the wild — including when it arrives as autocomplete. This is a different competency from being able to specify an architecture on a clean greenfield project. It is the competency of noticing that a choice has been made and that it deserves to be made deliberately.

5.2. Layer Two: Structured Validation

The second layer addresses verification of AI-generated outputs before they reach production. The empirical work surveyed in Section 2 establishes that AI-generated code contains a non-trivial share of security and correctness defects [8, 10]. Stray and colleagues' longitudinal study also documents the verification overhead this implies [10]. A governance framework that takes that overhead seriously has to make verification structured rather than ad hoc.

Three elements are useful. The first is treating AI-generated code as a distinct category in code review, marked as such, and reviewed with attention to the failure modes generative tools are known to produce: authentication weaknesses, input validation gaps, opinionated dependency choices that do not fit the system context. The second is integrating automated static analysis tuned to those failure modes. The third is recording, for audit and learning, the proportion of accepted AI suggestions that pass review unchanged versus those that require modification. Kept over time, that proportion becomes a calibration signal: how much should this organisation actually trust its tooling?

The curricular implication is that code review and testing — usually taught as generic engineering competencies — should be extended to include the specific failure modes of generated code. Graduates should be able to explain why a piece of code that looks plausible may still be unsafe, and they should be familiar with the categories of error generative tools most commonly produce. Bacchelli and Bird's foundational study of modern code review provides one anchor for that teaching [22]; the Pearce–Stray empirical thread provides another. A capable graduate should know both.

5.3. Layer Three: Operational Observability

The third layer covers what happens after AI-assisted code reaches production. Conventional observability practice already covers logs, metrics, and traces. The argument here is that AI-assisted environments benefit from an additional observability dimension concerned with the provenance and behaviour of generated components. This means recording which components were produced with substantial AI assistance, which tool versions were in use at the time, and which classes of incident correlate with AI-assisted versus conventionally-written code.

The principle is that, after an incident, the organisation should be able to answer the question whether AI assistance was a contributing factor. The purpose of that information is not blame allocation but calibration. Without it, decisions about how aggressively to adopt or constrain AI assistance are made on instinct rather than evidence.

For IS education, the implication is that observability should be taught as a governance practice and not only as a reliability practice. Graduates should be familiar with the principles of explainability in AI systems [18], with the regulatory drivers of audit trails [3], and with the wider organisational reasons that responsible AI governance is increasingly being treated alongside other disclosure regimes.

6. Embedding Governance Literacy in Information Systems Curricula

The framework in Section 5 is only useful for this venue if it produces concrete educational proposals. This section offers four, each pitched at something an IS programme could reasonably do without restructuring its degree.

Table 1
The three-layer framework and corresponding curriculum implications.

Layer	Governance practice	Curriculum target
Architectural ownership	Keeping architectural decisions with accountable humans when proposed by generative tools.	System design accountability; architectural decision records; human-in-the-loop principles.
Structured validation	Reviewing AI-generated outputs through processes designed for their failure modes.	Testing methods adapted for generated code; security pattern recognition; calibrated trust in automated tools.
Operational observability	Tracing the production behaviour of AI-assisted components and using it to calibrate adoption.	AI-specific monitoring; audit trails; explainability principles; regulatory drivers of observability.

6.1. Move Governance Out of the Ethics Module

AI ethics and governance content typically sits in a single dedicated module in IS curricula — often optional, often delivered late in the programme, often staffed by a colleague asked to "cover the ethics side." Tong and colleagues' content analysis of university generative-AI policies points in a related direction: institutional attention has concentrated on academic-integrity questions (whether students may use AI in coursework) rather than on professional preparation (whether graduates can govern AI in workplaces) [14]. The two patterns are consistent — both treat AI literacy as an addendum rather than a structural concern. That arrangement made sense when ethics and governance were genuinely separable from day-to-day engineering practice. It makes less sense now that a routine pull request involves judgements about provenance, accountability, and observability of generated code.

The proposal is to distribute governance content across the curriculum at the points where it actually arises in engineering practice. When students learn version control, they encounter provenance metadata for generated artefacts. When they learn testing, they encounter the specific failure modes of AI-generated code. When they learn system design, they engage with how architectural decisions are protected when generative tools are part of the development environment. The situated learning tradition supports this distribution: competencies are best learnt in the context of the practice they support [17].

6.2. Build a Final-Year "AI-Assisted Engineering" Module

A more specific proposal is a final-year capstone-style module organised around AI-assisted development as the central subject matter, not the means of delivery. The module uses Copilot or a comparable assistant throughout, with two assessments designed to force the evaluative judgement graduates will be expected to bring to a first job.

The first assessment is a code review of an AI-generated artefact. Students receive a non-trivial piece of generated code — the instructor controls what, and some defects can be seeded for pedagogical reasons — and produce a structured review: which suggestions are sound, which need modification, which should be rejected, and why. The mark scheme weights the reasoning at least as heavily as the conclusions. Becker and colleagues' SIGCSE paper provides one source of exemplar tasks [19]; the Pearce vulnerability catalogue provides another [8].

The second assessment is an architectural decision record justifying an AI-suggested dependency in a small system design. Students propose the dependency, accept or override the assistant's suggestion, and produce an ADR that defends the choice on architectural grounds. The mark scheme weights the awareness that a decision was being made — that is, the recognition competency from Section 3 — alongside the quality of the justification itself.

Neither assessment is expensive to design. Both are difficult to mark by rubric alone, which is a feature, not a bug: they require the same evaluative attention from the instructor that the students are being asked to develop.

6.3. Use Practitioner-Informed Case Material

Curriculum material in this area benefits substantially from being grounded in real practitioner experience rather than idealised laboratory examples. Anonymised case material drawn from enterprise environments — the kind this paper itself relies on — works better than abstract treatments of the same topics. Where direct practitioner contact is available through guest lectures, industry advisory boards, or shared training material, it should be used. The author's own circulation of internal training assets within a large engineering consultancy suggests that practitioner-developed material can travel widely if its provenance is clear. There is no structural reason similar material cannot also reach universities, given appropriate licensing.

6.4. Treat AI Literacy as a Layered Construct

Following Tadimalla and Maher [13], AI literacy in IS education is best treated as a layered construct rather than a single competency. The base layer covers general capability literacy — what generative tools do, how they are prompted, what their outputs look like. A second layer covers technical understanding of model behaviour, including known failure modes and the basics of explainability. A third layer covers governance literacy specifically: the regulatory environment, the organisational consequences of deployment choices, and the practical mechanics of frameworks like the one in Section 5.

Different roles will need different depths of engagement with each layer. A graduate moving into product management will need less of the second layer and more of the third than a graduate moving into platform engineering. But no graduate of a contemporary IS programme should leave without some grounding in all three. The first layer is now widely taught. The second is patchily taught. The third is largely not taught at all. Closing the third-layer gap is the single most concrete educational implication of the argument here.

7. Discussion: What Educators Can Do This Semester

The proposals in Section 6 are pitched at the level of curriculum direction. To make them tractable for the audience this paper is intended for, this short section turns those proposals into things an educator could reasonably try inside an existing module without a full curriculum review.

The first thing is to introduce code review of an AI-generated artefact as a single laboratory exercise inside an existing software engineering module. The exercise itself can sit inside an afternoon; the discussion that follows it tends to occupy substantially longer. Students who have only ever defended their own code find this exercise unfamiliar, which is the point.

The second is to require a short architectural rationale for any AI-suggested dependency in a project assessment. The rationale need not be long. A paragraph forces the recognition that a decision has been made. Recognition is most of the competency.

The third is to add one lecture on the Pearce vulnerability findings and one on the Stray verification-overhead findings to the security or testing strand of the programme. Both can be supplemented with the EU AI Act Article 4 obligation as a regulatory motivation [3]. Students respond to regulatory specifics with more attention than they respond to ethical principles in the abstract. This is unfortunate but useful.

The fourth is to find a practitioner. Industry advisory boards, alumni networks, and consultancy mentorship channels all exist; not all of them are being used. Practitioner contributions need not be polished to be useful — they need to be specific.

None of these is hard to do. None requires institutional permission. All of them move some distance toward closing the third-layer gap in AI literacy that Section 6 described.

8. Limitations and Future Work

The framework was developed through practitioner reflection and synthesis with the published literature; it has not been evaluated in a controlled organisational or educational setting. The curriculum proposals are pitched at the level of direction and module sketch rather than full module-level design with detailed learning outcomes and validated assessment rubrics. Observations come from a particular set of sectors (energy, manufacturing, media, pharmaceuticals, identity and access management) and a particular professional trajectory; they will not generalise straightforwardly to all enterprise contexts and certainly not to all educational ones.

Three directions for future work follow. The first is empirical evaluation of the three-layer framework in an enterprise setting, ideally in collaboration with an organisation already adopting AI-assisted development at scale. The second is the development of detailed module specifications — learning outcomes, marking rubrics, and case material — suitable for piloting in IS programmes. The third, the most ambitious, is a longitudinal study of how AI literacy among early-career engineers correlates with incident outcomes in production environments. Such a study would provide the empirical bridge between the educational argument and the governance argument that this paper has had to make on conceptual grounds.

The wider point is that responsible AI in software engineering cannot be a conversation conducted only by industry or only by academia. The framework here is most useful when it is being adopted in organisations and taught in universities at the same time. NWISEd, and workshops like it, are among the places where that simultaneous adoption can be coordinated.

9. Conclusion

This paper has argued that the adoption of generative AI in enterprise software engineering has opened a competency gap that information systems education is, at present, mostly leaving unaddressed. Article 4 of the EU AI Act has made that gap a regulatory matter since February 2025 [3]. The paper has set out three specific competencies that graduates entering AI-assisted workplaces need — reading generated code as if you wrote it, recognising architectural decisions when they arrive disguised as autocomplete, and owning the outcome when you did not write the code — and has proposed a three-layer framework, architectural ownership, structured validation, and operational observability, that maps onto those competencies. Each layer has then been translated into curriculum-level proposals that an IS programme could implement without restructuring its degree.

The proposals are not radical. They do not require new departments, new degree programmes, or wholesale curriculum revision. What they do require is a recognition that the discipline of software engineering, after three years of routine AI-assisted practice in industry, has changed enough that the curricula training graduates for it should change in proportion. The argument here is one contribution toward that change. It is offered to the workshop in the spirit, explicitly invited in the NWISEd call, of papers that exist to provoke discussion.

Acknowledgments

The author thanks the practitioners and early-career engineers whose workplace experiences informed the observations on which this paper draws. All identifying details have been anonymised.

Declaration on Generative AI

During the preparation of this work, the author used generative AI tools to assist with literature search, draft editing, and improving clarity and readability of the manuscript. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

References

[1] National Institute of Standards and Technology, Artificial Intelligence Risk Management Framework (AI RMF 1.0), U.S. Department of Commerce, 2023.

[2] National Institute of Standards and Technology, Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile (NIST AI 600-1), U.S. Department of Commerce, 2024.

[3] European Parliament and Council of the European Union, Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act), *Official Journal of the European Union*, 2024.

[4] A. A. Khan, S. Badshah, P. Liang, M. Waseem, B. Khan, A. Ahmad, M. Fahmideh, M. Niazi, and M. A. Akbar, Ethics of AI: A systematic literature review of principles and challenges, in *Proc. 26th Int. Conf. Evaluation and Assessment in Software Engineering (EASE 2022)*, ACM, 2022.

[5] A. Jobin, M. Ienca, and E. Vayena, The global landscape of AI ethics guidelines, *Nature Machine Intelligence*, vol. 1, pp. 389–399, 2019.

[6] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, The impact of AI on developer productivity: Evidence from GitHub Copilot, *arXiv preprint arXiv:2302.06590*, 2023.

[7] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, Measuring GitHub Copilot's impact on productivity, *Communications of the ACM*, vol. 67, no. 3, 2024.

[8] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions, *Communications of the ACM*, vol. 68, no. 2, 2025.

[9] K. Kohl and L. Carro, When code becomes abundant: Redefining software engineering around orchestration and verification, in *Proc. IEEE/ACM 48th Int. Conf. Software Engineering: Future of Software Engineering Track (ICSE-FoSE 2026)*, Rio de Janeiro, Brazil, 2026.

[10] V. Stray, E. G. Brandtzæg, V. T. Wivestad, A. Barbala, and N. B. Moe, Developer productivity with and without GitHub Copilot: A longitudinal mixed-methods case study, *arXiv preprint arXiv:2509.20353*, 2025.

[11] G. Eboh, Educating for responsible AI governance: Building interdisciplinary frameworks for the management of artificial intelligence, *Southwestern Business Administration Journal*, vol. 22, no. 2, 2025.

[12] X. Gu and B. J. Ericson, AI literacy in K-12 and higher education in the wake of generative AI: An integrative review, in *Proc. 2025 ACM Conf. International Computing Education Research (ICER 2025)*, 2025.

[13] S. Y. Tadimalla and M. L. Maher, AI literacy as a core component of AI education, *AI Magazine*, vol. 46, no. 2, e70007, 2025.

[14] A. Tong, Z. Zainol, T. S. Chong, and K. Renganathan, AI governance on young consumers in higher education: A content analysis of policies for generative AI, *Young Consumers*, vol. 26, no. 5, pp. 865–885, 2025.

[15] M. Kalantzis and B. Cope, Literacy in the time of artificial intelligence, *Reading Research Quarterly*, vol. 60, no. 1, e591, 2024.

[16] D. A. Schön, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York, 1983.

[17] J. Lave and E. Wenger, *Situated Learning: Legitimate Peripheral Participation*, Cambridge University Press, Cambridge, 1991.

[18] A. B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, et al., Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI, *Information Fusion*, vol. 58, pp. 82–115, 2020.

[19] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, Programming is hard — or at least it used to be: Educational opportunities and challenges of AI code generation, in *Proc. 54th ACM Technical Symposium on Computer Science Education (SIGCSE 2023)*, pp. 500–506, 2023.

[20] P. Denny, J. Prather, B. A. Becker, J. Finnie-Ansley, A. Hellas, J. Leinonen, A. Luxton-Reilly, B. N. Reeves, E. A. Santos, and S. Sarsa, Computing education in the era of generative AI, *Communications of the ACM*, vol. 67, no. 2, pp. 56–67, 2024.

[21] J. Prather, B. N. Reeves, P. Denny, B. A. Becker, J. Leinonen, A. Luxton-Reilly, G. Powell, J. Finnie-Ansley, and E. A. Santos, The widening gap: The benefits and harms of generative AI for novice programmers, in *Proc. 2024 ACM Conf. International Computing Education Research (ICER 2024)*, 2024.

[22] A. Bacchelli and C. Bird, Expectations, outcomes, and challenges of modern code review, in *Proc. 35th Int. Conf. Software Engineering (ICSE 2013)*, IEEE Press, pp. 712–721, 2013.