AI-Resilient Coding Assessment: Valid Evidence in the Generative AI Era

André Delwyn Gopal
Digital Arts Department, Wits School of Arts, University of the Witwatersrand, Johannesburg, South Africa

andre.gopal@wits.ac.za

https://doi.org/10.34190/ejel.24.5.4965

An open access article under CC Attribution 4.0

Abstract: Generative AI tools can generate, explain, debug, refactor, and document code, and they are now routinely available to students in code-based courses. This does not make programming knowledge irrelevant, but it changes what a submitted coding artefact can credibly evidence. A polished website, application, script, or repository may demonstrate that a product was produced; it does not necessarily demonstrate that the student who submitted it can independently read, trace, debug, modify, explain, decompose, implement, or integrate code. Much current debate frames this shift primarily as a problem of academic misconduct. This conceptual article argues that the deeper and more urgent problem is validity: many widely used coding assessments may no longer support the inferences that educators have traditionally drawn from them, regardless of whether any rule has been broken. If this gap is left unaddressed, institutions risk awarding qualifications on the basis of evidence that no longer demonstrates the competences those qualifications are meant to certify. To address this gap, the article synthesises four bodies of literature: assessment validity theory, computing education research on programming competences such as tracing, debugging, and comprehension, generative AI assessment guidance, and research on AI-assisted programming tools. From this synthesis, it develops a framework of AI-resilient assessment design for code-based courses. The framework links four design dimensions: the programming construct being assessed, the AI-access condition under which the task is completed, the assessment task type, and the credibility of the evidence that the task generates. Building on these dimensions, the article distinguishes five assessment families: controlled fundamentals assessments, constrained transformation assessments, process-evidenced project assessments, AI-documented assessments, and AI-integrated professional assessments. The article's central contribution is not an empirical finding but a practical framework. It gives educators a structured way to ask what a given task actually provides evidence of, under what AI-access condition, and whether that evidence supports the learning claim being made. Used together, the five assessment families allow courses to evidence both independent coding fluency and accountable AI-supported practice, rather than relying on a single artefact to evidence both at once. For code-based courses across higher education, adopting this kind of framework may be less a refinement than a precondition for assessment remaining a credible measure of programming competence at all.

Keywords: AI-resilient assessment, Programming education, Assessment validity, Generative AI in higher education, Code comprehension, Academic integrity

1. Introduction

Code-based courses have long relied on submitted artefacts: scripts, websites, applications, notebooks, repositories, prototypes, and project documentation. These artefacts matter because programming is a productive discipline. Students must eventually make things that run, respond, calculate, display, store, or communicate. Yet a submitted artefact has never been a simple window into individual competence. Students can receive help from peers, copy fragments from examples, rely on libraries, reuse templates, follow tutorials, or solve a task through trial and error without fully understanding why the final program works. Generative AI intensifies this long-standing evidentiary problem because it can now contribute directly to the production of code and to the explanation of code-like artefacts.

Large language model tools can generate plausible source code from natural-language prompts, suggest fixes, explain unfamiliar code, rewrite functions, produce tests, and draft documentation. Research in programming education reports both opportunities and risks: such tools may support debugging, feedback, practice, and productivity, but they may also obscure whether students have developed durable conceptual understanding (Kasneci et al., 2023; Xue et al., 2024). Studies that evaluate ChatGPT and related systems on programming tasks show that these systems can produce useful solutions for many introductory exercises while remaining uneven on correctness, testing, explanation, and more complex problem solving (Ali and Wibowo, 2023; Khan et al., 2024). The assessment question is therefore not whether AI can write code in every circumstance. It is what a particular assessment still provides evidence of when AI can plausibly contribute to the artefact.

Cite this article: Gopal, A.D. 2026. "AI-Resilient Coding Assessment: Valid Evidence in the Generative AI Era ", *Electronic Journal of e-Learning*, 24(5), pp 87-99, https://doi.org/10.34190/ejel.24.5.4965

The dominant public framing of this problem has often been misconduct. That concern is legitimate. If an assessment brief prohibits AI and a student submits AI-generated work as their own, the issue is academic integrity. However, misconduct is not the only issue and may not be the deepest one. Assessment validity concerns the interpretation of evidence: what claim about learning is supported by the task, the conditions under which it was completed, and the evidence collected (Kane, 2013; Messick, 1989). A coding assignment may support the claim that a student delivered a functioning product under open-resource conditions. It may not support the stronger claim that the student can independently trace unfamiliar code, locate a logic error, justify a design decision, or modify an existing codebase under time constraints. Generative AI widens the gap between artefact quality and individual ability unless assessment design makes the intended inference explicit.

This article uses the term code-based courses for courses in which coding is a substantial object of assessment, including introductory programming, web development, app development, creative coding, software engineering, data scripting, and related digital production courses. In such courses the framework applies where programming is itself a substantial assessed competence; visual design or creative execution may be co-assessed but is not the framework's target construct. Such courses are multi-construct assessment environments. They do not assess one generic skill called "coding". They may assess syntax, semantics, code reading, tracing, debugging, decomposition, implementation, integration, testing, documentation, interface behaviour, version-control workflow, and the ability to explain or defend technical choices. These constructs are related but not identical. A student can assemble an attractive web interface while struggling to explain event handling; another can trace code accurately but produce poor project documentation; another can use AI effectively to improve a prototype but lack independent debugging fluency.

The article argues that assessment validity after generative AI depends less on whether AI is banned or permitted in general than on whether the assessment is designed to generate credible evidence of specific programming abilities under clearly defined AI-access conditions. AI-prohibited invigilated tests may remain appropriate for independent fluency, code tracing, short debugging, and basic implementation. AI-permitted project work may be appropriate for authentic production, iteration, documentation, and professional workflow. AI-documented or AI-integrated tasks may be appropriate when the outcome includes responsible tool use, critique of generated code, or code review. The problem arises when a single artefact is asked to provide evidence of all of these abilities at once.

To address this problem, the article develops a framework of AI-resilient assessment design for code-based courses. AI-resilient assessment is defined here as assessment that remains capable of producing credible evidence of learning under explicitly stated AI-access conditions. The framework distinguishes five assessment families: controlled fundamentals assessments, constrained transformation assessments, process-evidenced project assessments, AI-documented assessments, and AI-integrated professional assessments. The contribution is conceptual and design-oriented. It does not offer a universal policy or claim that any one assessment format is sufficient. Instead, it provides a structured basis for aligning programming constructs, task types, AI conditions, and evidence claims.

2. Theoretical Framing: Assessment Validity, Constructive Alignment, and Code-Based Learning

2.1 Assessment Validity: Evidence, Inference, and Learning Claims

Validity is not a property of an assessment task in isolation. It concerns the interpretation and use of assessment evidence. Messick (1989) defines validity as an integrated evaluative judgement of the degree to which evidence and theory support the interpretations of scores and other assessment outcomes. Kane's (2013) argument-based approach similarly asks whether the inferences from performance to score, from score to interpretation, and from interpretation to use are warranted. Three related terms are used throughout with distinct meanings. A construct is the attribute an assessment is designed to target — for example, debugging fluency or code comprehension — in the assessment-validity sense of Messick (1989) and Kane (2013), not in the programming sense of a language feature such as a loop or conditional. A competence is an ability a student actually possesses. An intended learning outcome is the claim a course makes about what a student should be able to do. An assessment targets a construct in order to license an inference about a student's competence against an intended learning outcome. Applied to code-based courses, this means that a submitted program is not automatically evidence of every learning outcome that might be associated with programming.

An assessment artefact is the product submitted for marking: a code file, website, app, repository, notebook, prototype, or documentation pack. Assessment evidence is the basis on which the educator infers what the

 ©The Authors

student knows or can do. The distinction matters because a strong artefact can be weak evidence for a particular claim. A functioning app may support the claim that the student can assemble a project under open-resource conditions. It may be weaker evidence that the student can independently predict program behaviour, identify a fault, explain asynchronous execution, or adapt a component to a new requirement. Generative AI does not create this distinction, but it makes it harder to ignore.

In validity terms, the central risk is construct-irrelevant support and construct under-representation. If the intended construct is independent debugging fluency, unrestricted AI help may supply assistance that is irrelevant to the construct and inflate the apparent performance. If the intended construct is professional AI-supported development, a strictly AI-prohibited handwritten test may under-represent the real task environment. The validity question is not "Is AI good or bad?" but "What inference is the educator making, and does the evidence justify it?"

2.2 Constructive Alignment and Assessment Design

Constructive alignment provides the second foundation for the framework. In Biggs's (1996) formulation, intended learning outcomes, teaching activities, and assessment tasks should be aligned so that students are required to enact the forms of learning that the course values. Biggs and Tang (2011) later developed this as a practical approach to outcomes-based teaching in higher education. In code-based courses, alignment requires more precision than "submit a program" or "build a website". If an outcome states that students should debug code, the assessment must generate evidence of debugging. If an outcome states that students should use development tools responsibly, the assessment may need to include version control, testing, documentation, or AI-use decisions.

Authentic assessment literature helps explain why project work remains valuable. Wiggins (1990) argues that authentic assessment asks students to perform meaningful tasks rather than merely recall information. Gulikers, Bastiaens and Kirschner (2004) propose that authenticity includes the task, physical or digital context, social context, result, and criteria. A project-based coding assessment can therefore be highly valuable when the course aims to assess design integration, iteration, documentation, user-facing behaviour, or professional workflow. However, authenticity does not automatically solve validity. A realistic task can still produce ambiguous evidence if the permitted tools, collaboration rules, and verification mechanisms are unclear.

Constructive alignment in the context of generative AI therefore requires explicit AI-access conditions. An AI-access condition is the stated level of AI use permitted in an assessment: prohibited, limited, documented, permitted, or required. Each condition supports different learning claims. Prohibiting AI may be justified where the outcome is independent fluency. Permitting AI with documentation may be justified where the outcome is accountable tool use. Requiring AI may be justified where the outcome is critical evaluation of generated code. Problems arise when students and assessors infer different conditions, or when the condition does not match the construct being assessed.

2.3 Code-Based Courses as Multi-Construct Assessment Environments

Computing education research has repeatedly shown that novices struggle with distinct aspects of programming, including syntax, semantics, planning, tracing, debugging, and comprehension (Robins, Rountree and Rountree, 2003; Winslow, 1996). Code tracing is not merely a mechanical activity; it requires students to mentally simulate program execution and reason about state changes (Lister et al., 2004; Sorva, 2013). Program comprehension matters because students must learn to read and reason about code created by others, not only write code from a blank page (Izu et al., 2019). Debugging is likewise a distinctive competence involving hypothesis formation, fault localisation, testing, and explanation, rather than a simple extension of code writing.

Assessment formats can target these constructs differently. Parsons problems, in which students rearrange mixed-up code blocks, reduce the burden of syntax production while still assessing ordering, logic, and comprehension (Parsons and Haden, 2006). Faded Parsons problems add blanks that require students to complete parts of the solution and can scaffold the transition from reading to writing code (Ericson et al., 2022; Weinman, Fox and Hearst, 2021). Short tracing questions can expose misconceptions about control flow, variables, scope, and program state. Debugging tasks can reveal whether students can locate and explain faults. Modification tasks can show whether students understand existing code well enough to adapt it.

These distinctions are central to AI-resilient design. A take-home project — one completed outside controlled or invigilated conditions — may be the right task for assessing integration, user experience, documentation, and sustained development. It is a weaker task for assessing unaided code tracing unless paired with a controlled

task. Conversely, an invigilated tracing test may validly assess independent reasoning, but it is weak evidence of project planning, interface polish, or professional workflow. A resilient assessment mix therefore combines assessment families rather than searching for a single AI-proof format.

3. Generative AI and the Changed Evidentiary Value of Coding Tasks

Generative AI changes coding assessment because it can assist with many behaviours that educators previously treated as stronger evidence of student knowledge. A student can ask a tool to write a function, interpret an error message, explain an unfamiliar loop, refactor repetitive code, generate documentation, produce tests, or suggest a likely fix. These capabilities vary by tool, prompt quality, language, task complexity, and model version, but the direction of travel is clear enough for assessment design: open-resource coding artefacts now have a more ambiguous evidentiary status than they did before the public release of widely accessible generative AI systems.

Recent computing education studies present a mixed picture. Xue et al. (2024) found that ChatGPT use in an introductory programming context did not automatically translate into improved learning performance, suggesting that access to generated help is not the same as durable competence. Khan et al. (2024) show that ChatGPT can be useful on introductory programming (CS1) assignments but also raises concerns about how prompts, explanations, and solution correctness are evaluated. Ali and Wibowo (2023) similarly assess ChatGPT-generated code against introductory programming exercises and show that generated solutions can be plausible enough to matter for assessment. The implication is not that students no longer need to learn programming. It is that educators can no longer treat an independently submitted coding artefact as self-evident evidence of all underlying competences.

AI-assisted programming also changes the professional context of code-based education. Industry-facing programming increasingly includes completion tools, code assistants, automated documentation, test generation, and AI-supported debugging. Higher education guidance on generative AI increasingly recommends moving beyond simple prohibition toward assessment designs that clarify acceptable use and align it with outcomes (Lodge et al., 2023; Perkins et al., 2024; UNESCO, 2023). The AI Assessment Scale, for example, offers levels of AI use that range from no AI to AI exploration, helping educators communicate whether AI is forbidden, limited, used for editing, used for task completion, or central to the task (Perkins et al., 2024). Such frameworks are useful because they make tool access part of the assessment design rather than an implicit background condition.

However, allowing AI everywhere would also be a mistake. Students need independent fluency because they cannot responsibly evaluate, repair, or extend AI-generated code without enough underlying understanding. A novice who cannot trace a simple function may accept a confident but faulty explanation. A student who cannot debug may paste successive error messages into a tool without forming a testable hypothesis. A student who cannot read code may struggle to recognise security, accessibility, maintainability, or logic problems. AI-supported competence therefore depends partly on non-AI competence. This is why this article distinguishes independent fluency from AI-supported competence rather than ranking one above the other.

AI detection is a weak primary solution. Detection systems are adversarial, tool-specific, and vulnerable to false positives and false negatives. Research on AI-generated text detection has shown that detector reliability and fairness are serious concerns, including bias risks for non-native English writers (Liang et al., 2023). In code assessment, the problem is even more complex because similar short solutions may legitimately converge, students often use common patterns, and code can be reformatted or rewritten. Detection may have a limited evidentiary role in academic-integrity investigations, but it cannot carry the full burden of assessment validity.

The more defensible response is to specify AI-access conditions and diversify assessment evidence. Take-home projects can remain valuable if they assess project planning, iteration, design integration, documentation, version-control workflow, and the ability to explain decisions. Controlled tests can remain valuable if they assess independent tracing, debugging, short implementation, and misconceptions. Transformation tasks can assess whether students can work with code they did not generate. AI-documented and AI-integrated tasks can assess responsible use, critique, verification, and accountability. The changed evidentiary value of coding tasks therefore calls not for abandoning projects or returning to exams alone, but for a clearer assessment mix.

4. Methodology: Conceptual Synthesis and Framework Development

This article is a conceptual design-framework article. Its literature base was assembled through a purposive conceptual synthesis rather than a systematic review: sources were selected for their theoretical relevance to

 ©The Authors

the four strands the argument requires, prioritising seminal contributions and recent work on generative AI in programming education, rather than through an exhaustive or protocol-driven search. The selection is therefore deliberately selective rather than comprehensive, and is intended to support conceptual development rather than to establish the prevalence of findings across a field. It does not report an empirical study, analyse student submissions, collect survey or interview data, or measure learning outcomes. Its purpose is to synthesise relevant bodies of literature and derive a practical framework that can guide assessment design in code-based courses after generative AI. Conceptual articles are valuable when they clarify constructs, integrate previously separate literatures, and propose relationships or frameworks that can be examined in later empirical work (Jaakkola, 2020). The contribution here is therefore not a claim that the framework improves learning or reduces misconduct, but a reasoned framework for designing assessments that make more defensible evidence claims.

The synthesis draws on four bodies of literature, developed more fully in Sections 2 and 3. The first is assessment theory — validity, constructive alignment, and authentic assessment — which supplies the central distinction between assessment artefact, assessment evidence, and learning inference (Biggs, 1996; Biggs and Tang, 2011; Gulikers, Bastiaens and Kirschner, 2004; Kane, 2013; Messick, 1989; Wiggins, 1990). The second is computing education research on programming competences, code comprehension, tracing, debugging, and Parsons problems, which distinguishes the constructs that code-based courses assess under the broad label of programming (Ericson et al., 2022; Izu et al., 2019; Lister et al., 2004; Robins, Rountree and Rountree, 2003; Sorva, 2013; Weinman, Fox and Hearst, 2021). The third concerns generative AI in education and assessment, supporting the claim that assessment design should specify AI-use conditions rather than treat AI only as an external disruption (Kasneci et al., 2023; Lodge et al., 2023; Perkins et al., 2024; UNESCO, 2023). The fourth concerns AI-assisted programming and professional coding practice, situating code-based education in a changing tool environment (Ali and Wibowo, 2023; Khan et al., 2024; Xue et al., 2024).

The framework was developed by asking four design questions across these literatures. First, what programming construct is being assessed? Second, what AI-access condition applies? Third, what task type is appropriate for that construct and condition? Fourth, what evidence does the task produce, and what inference does that evidence justify? The resulting typology organises assessment designs into five families: controlled fundamentals, constrained transformation, process-evidenced projects, AI-documented assessment, and AI-integrated professional assessment. Each family is described in terms of suitable constructs, task types, evidence produced, validity risks, and safeguards.

Because the article uses only existing literature and illustrative design reasoning, and does not involve students, participants, identifiable student work, grades, interviews, surveys, or observations, it would typically fall outside human-subject research ethics review. However, ethics requirements vary by institution and journal. The safer claim is not that ethics approval is never relevant for conceptual articles, but that this article itself does not use human-subject data. Future empirical studies applying the framework to real courses would require appropriate ethics review where student data or staff/student perspectives are collected.

5. The AI-Resilient Assessment Framework

Table 1 presents the typology. The categories are not intended as a rigid sequence or exclusive hierarchy; a single course may draw on several families in parallel, while a short module may use only two or three. The key is that the assessment family should match the learning claim being made.

Table 1: Typology of AI-resilient assessment design for code-based courses

Assessment family	Primary constructs assessed	AI-access condition	Example task types	Evidence produced	Main validity risk	Safeguard
Controlled fundamentals	Syntax, semantics, tracing, debugging, misconceptions, short reasoning	AI prohibited or inaccessible	Trace code; explain output; repair a small bug; complete missing lines	Evidence of independent reasoning about code	May overemphasise exam performance or memorisation	Use meaningful short tasks and pair with authentic work

Assessment family	Primary constructs assessed	AI-access condition	Example task types	Evidence produced	Main validity risk	Safeguard
Constrained transformation	Comprehension, modification, debugging, refactoring, integration	Prohibited, limited, or tightly specified	Modify supplied code; fix seeded bugs; add a feature; write tests	Evidence that student can work with existing code	Generic tasks may be solved by AI; obscure tasks may be unfair	Use course-specific code, short windows, and required explanation
Process-evidenced project	Planning, integration, iteration, documentation, design decisions, workflow	Prohibited, limited, documented, or permitted	Project build; Git milestones; demos; code review; design rationale	Evidence of development process and project ownership	Process evidence can be fabricated or misunderstood	Use checkpoints, walkthroughs, and controlled modifications
AI-documented assessment	AI literacy, judgement, critique, accountability	AI permitted but documented	AI-use log; compare AI output with revision; annotate decisions	Evidence of transparent and critical AI use	Logs and reflections may be superficial or staged	Assess critique quality and pair with independent checks
AI-integrated professional assessment	Code review, testing, maintainability, debugging, responsible AI use	AI permitted or required	Critique generated code; refactor AI output; test flawed AI solution	Evidence of professional judgement with AI tools	Can mask weak fundamentals or create inequitable access	Use common tools, flawed code, explanations, and tests

5.1 Controlled Fundamentals Assessments

Controlled fundamentals assessments are designed to generate evidence of independent fluency. Their defining feature is not that they are traditional or punitive, but that AI assistance is prohibited or inaccessible because the construct being assessed requires unaided reasoning. Suitable tasks include code tracing, explain-the-output questions, short debugging tasks, syntax correction, prediction of program behaviour, completion of missing lines, and concise explanations of code logic. These tasks are especially useful when the educator needs evidence that students can reason about variables, control flow, functions, state changes, events, data structures, or errors without external generation support.

The strength of controlled fundamentals assessment is evidentiary clarity. If the task is well designed and conditions are controlled, the educator can make a more credible claim about independent code comprehension and short-form problem solving. This is important because independent fluency is a prerequisite for responsible AI-supported work. The limitation is that controlled tasks can overemphasise artificial exam performance, test anxiety, or recall if they are poorly designed. They also under-assess authentic development practices such as sustained design, version control, documentation, and integration. For this reason, controlled fundamentals should be used for fundamentals, not for the full range of assessed competences.

5.2 Constrained Transformation Assessments

Constrained transformation assessments ask students to understand, modify, repair, or extend existing code. They occupy a useful middle ground between short tests and open-ended projects. A student might receive a small codebase and be asked to add a feature, repair seeded bugs, refactor a repetitive function, adapt behaviour to a new requirement, compare two implementations, or write tests for supplied code. These tasks assess comprehension and modification rather than blank-page production.

Transformation tasks are well suited to AI-resilient assessment because they can be local, contextual, and tied to course-specific code. They generate evidence that students can work with code they did not author from scratch, a common professional requirement. Their main risk is task design. If the code and requirement are generic, AI may solve the task with little student understanding. If the code is too large or obscure, the task becomes unfair. Safeguards include using short time windows, unseen variants of familiar patterns, locally written codebases, and required explanations of why the modification works. The explanation should be tied to specific lines, tests, or behaviours rather than a general reflective paragraph.

 ©The Authors

5.3 Process-Evidenced Project Assessments

Process-evidenced projects preserve the value of authentic coding work while making the development process more visible. They may include a website, app, creative coding artefact, data script, or software prototype, accompanied by staged milestones, design documentation, Git history, issue tracking, code review, progress demonstrations, a development log, and a final code walkthrough. Such projects are appropriate when the learning outcomes include planning, design integration, iteration, documentation, professional workflow, and the ability to explain design and coding choices.

The validity risk is that process evidence is useful but not conclusive. Git history can be generated in bursts, logs can be reconstructed, and demonstrations can be rehearsed. Process evidence should therefore be treated as strengthening the evidentiary chain, not replacing verification. Strong project assessment pairs product and process with short code walkthroughs, live demonstrations of current work, and controlled modification tasks in which students adapt their own submitted code. In large classes, full oral defences may be impractical, but lightweight verification can still be sampled or used for borderline cases. The aim is not surveillance; it is to align the evidence with the claim of student ownership and understanding.

5.4 AI-Documented Assessment

AI-documented assessment permits AI use but requires students to make that use visible and accountable. Task types include AI-use declarations, prompt and output logs, annotations of accepted and rejected suggestions, comparison between AI-generated and student-revised code, bug reports on AI output, and reflections on how AI changed the development process. The construct is not independent fluency; it is transparent and critical tool use.

Documentation alone is weak evidence if it becomes a compliance exercise. Prompt logs can be incomplete, staged, or too voluminous to assess. Reflections can become formulaic. The safeguard is to assess the quality of judgement rather than the mere presence of documentation. Students should be asked to identify specific AI suggestions, explain why they accepted or rejected them, test claims made by the tool, and take responsibility for the final code. AI-documented tasks are strongest when paired with controlled fundamentals or constrained transformation tasks, because transparency about tool use does not by itself evidence underlying competence.

5.5 AI-Integrated Professional Assessments

AI-integrated professional assessments make AI part of the task. Students may be asked to generate an initial solution using AI and then critique it, evaluate AI-generated code for accessibility or security, compare alternative generated implementations, write tests for generated code, refactor flawed AI output, or identify hallucinated APIs and faulty logic. The assessed construct is professional judgement under AI-supported conditions.

These tasks are important because they prepare students for contemporary tool environments. However, they should not be introduced as a substitute for fundamentals. If students cannot read and reason about code, they cannot reliably critique generated code. Access also matters: students using different tools may receive different levels of support. Safeguards include using institutionally available tools where possible, providing common AI-generated artefacts for critique, requiring explanation and testing, and combining AI-integrated tasks with independent assessments. The goal is to assess whether students can work productively with AI while remaining accountable for correctness, maintainability, ethics, and user impact.

6. Design Principles for AI-Resilient Coding Assessment

Table 2 summarises these design principles, linking each principle to its rationale, an example implementation, and the principal validity risk it addresses. The first principle is to assess the construct, not merely the artefact. A website, app, or script should not be treated as direct evidence of all relevant coding abilities. Rubrics should identify whether marks are being awarded for implementation, design integration, debugging, testing, documentation, explanation, or professional workflow. This reduces the risk that an attractive artefact receives credit for competences that were never directly evidenced.

Table 2: Design principles for AI-resilient coding assessment

Principle	Rationale	Example implementation	Assessment risk addressed
Assess the construct, not merely the artefact	A finished product is not direct evidence of every coding ability	Map each mark category to a construct such as tracing, debugging, integration, or explanation	Over-inference from polished submissions
Specify the AI-access condition	Students and assessors need shared rules for tool use	State whether AI is prohibited, limited, documented, permitted, or required	Ambiguous academic-integrity expectations
Separate independent fluency from AI-supported production	Both may matter, but they produce different evidence	Use invigilated tracing plus AI-documented project work	Confusing tool-supported output with unaided competence
Use controlled tasks for fundamentals, not everything	Controlled conditions support independent evidence but under-assess authenticity	Short debugging and tracing tests alongside projects	Over-reliance on exams
Require explanation, modification, and debugging	These tasks reveal understanding more directly than production alone	Ask students to modify supplied code and justify the change	AI-generated blank-page solutions
Treat process evidence as useful but insufficient	Logs and Git history strengthen but do not by themselves evidence understanding	Milestones plus final code walkthrough	Fabricated or superficial process evidence
Pair projects with verification tasks	Large artefacts need supporting evidence of ownership	Live demo, code walkthrough, or controlled change to own code	Outsourcing or uncritical AI use
Assess AI critique, not just AI use	Responsible AI use requires evaluation and accountability	Mark tests, corrections, and rejected AI suggestions	Prompt-log compliance without judgement

The second principle is to specify the AI-access condition. Each assessment brief should state whether AI is prohibited, limited, documented, permitted, or required. This condition should be connected to the learning outcome, not added as a generic policy statement. For example, AI may be prohibited in a tracing test because the target construct is independent reasoning, while AI may be required in a code-review task because the target construct is critique of generated code.

The third principle is to separate independent fluency from AI-supported production. Students may need both. Independent fluency includes the ability to read, trace, debug, and explain code without AI. AI-supported competence includes the ability to prompt, evaluate, test, revise, and take responsibility for generated assistance. A balanced course should avoid pretending that professional tools do not exist, while also avoiding the assumption that tool fluency replaces foundational understanding.

The fourth principle is to use controlled tasks for fundamentals, not for everything. Invigilated or controlled short tasks are valuable when the evidence claim concerns independent comprehension or debugging. They are less useful for assessing sustained design, iteration, project management, or authentic integration. Overuse can narrow the curriculum and produce brittle learning. Underuse can leave educators with little credible evidence of unaided reasoning.

The fifth principle is to require explanation, modification, and debugging. Tasks that ask students to explain, repair, modify, or extend code often generate stronger evidence than tasks that ask only for code production. This does not make them AI-proof, but it improves the evidentiary chain because students must connect code behaviour, intention, and justification. The explanation should be specific enough to be assessable: what changed, why it works, how it was tested, and what trade-offs remain.

The sixth and seventh principles concern process and verification. Git history, design logs, milestones, and demos provide useful evidence of development over time, especially in project-based courses. But they should be paired with verification tasks such as code walkthroughs, short viva-style checks, live demonstrations, or controlled modifications. These safeguards can be lightweight and targeted. They are most defensible when designed into the assessment from the start rather than used only as suspicion-driven policing.

 ©The Authors

The final principle is to assess AI critique, not merely AI use. If AI is allowed, students should be evaluated on whether they can identify errors, test claims, improve maintainability, check accessibility or security implications, and justify final decisions. A prompt log without critique is administrative evidence; a tested, annotated revision of AI-generated code is stronger evidence of responsible AI-supported competence.

7. Example Assessment Mix for a Code-Based Course

Figure 1 illustrates a balanced assessment mix. It is not a universal model. Weightings should depend on level, class size, discipline, learning outcomes, institutional AI policy, accessibility requirements, and available support for invigilation or verification.

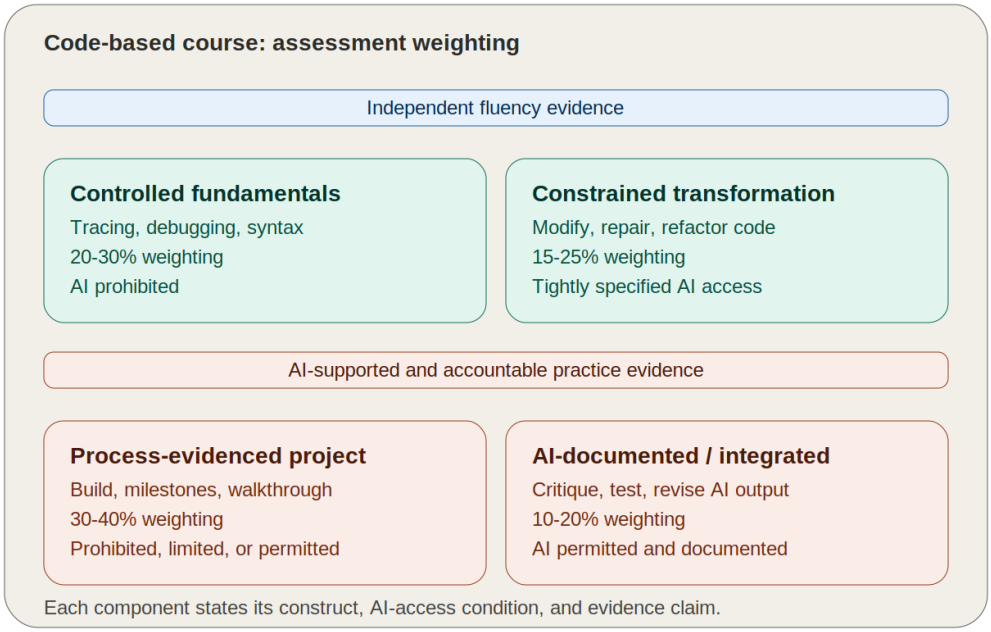

Figure 1: Balanced assessment mix for code-based courses

An introductory or intermediate code-based course might combine four kinds of evidence. First, controlled fundamentals tests could carry 20-30% of the assessment weighting. These would assess code tracing, explain-the-output questions, short debugging, syntax and semantics, and small implementation tasks under AI-prohibited conditions. Their function would be to produce credible evidence of independent fluency, not to replace all authentic work.

Second, constrained transformation tasks could carry 15-25%. Students might modify supplied code, repair seeded bugs, add small features, refactor existing functions, or write tests for existing components. These tasks are useful because they assess code comprehension and modification rather than only blank-page production. They can also be designed around course-specific code, reducing the value of generic generated answers.

Third, a process-evidenced project could carry 30-40%. In a web development course, for example, students might build a cohesive website or small application supported by a product brief, wireframes, style guide, Git milestones, progress demonstrations, and a final code walkthrough. The project would assess planning, design integration, iteration, implementation over time, documentation, and ownership of design and coding choices. If AI use is prohibited, limited, or documented, that condition should be explicit and reflected in the marking criteria.

Fourth, an AI-documented or AI-integrated task could carry 10-20%, depending on the course outcomes. Students might critique AI-generated code, compare AI output with their revised version, document accepted and rejected suggestions, or test a generated implementation for faults. This component recognises that AI-supported development is becoming part of the professional environment, while still requiring students to exercise judgement.

This assessment mix is deliberately balanced. It rejects two weak extremes: relying only on invigilated fundamentals, which may narrow what counts as coding competence, and relying only on open-ended projects, which may make independent understanding difficult to verify. A balanced mix also allows educators to adjust assessment design over time. In a first-year programming course, independent fundamentals may carry more weight. In a senior software engineering or creative technology course, process-evidenced projects and AI-integrated professional tasks may carry more weight. The validity principle remains the same: each component should state what it assesses, under what AI condition, and what inference the evidence supports.

8. Discussion: From AI Resistance to Assessment Resilience

The phrase AI-resistant assessment can be useful when educators need tasks that are difficult for AI to complete on behalf of students. Yet taken too literally, it can lead to a narrow and defensive design logic: make tasks more obscure, more locked down, more handwritten, or more surveilled. Some resistance is necessary when the learning claim concerns independent fluency. But code-based education also prepares students for tool-rich professional and creative environments. A more productive goal is assessment resilience: designing assessment systems that continue to produce credible evidence of learning under realistic and explicitly stated AI conditions.

Assessment resilience avoids two weak extremes. The first is banning AI everywhere and pretending that professional coding practice has not changed. This approach may protect some evidence of independent competence, but it can also prevent students from learning how to evaluate, document, and take responsibility for AI-supported work. The second extreme is allowing AI everywhere and treating output quality as sufficient evidence. This approach may produce impressive artefacts but can weaken assurance that students understand the code they submit. The framework proposed here gives educators a principled basis for avoiding both extremes.

Independent fundamentals still matter. This claim is not nostalgic. It follows from the logic of AI-supported practice itself. To use AI responsibly, students need enough knowledge to recognise faulty logic, insecure patterns, inaccessible interfaces, hallucinated APIs, misleading explanations, or brittle tests. A student who cannot form an independent hypothesis about a bug is poorly positioned to evaluate an AI-generated fix. Controlled fundamentals and constrained transformation tasks therefore remain important, especially in early and intermediate courses.

AI-integrated assessment also matters. If students are likely to use AI tools in professional, research, or creative coding contexts, then assessment should sometimes require them to practise transparent, critical, and accountable use. This means marking the evaluation of AI output, not merely the production of an AI-assisted artefact. Students should be asked to test generated code, compare alternatives, justify revisions, disclose assistance appropriately, and accept responsibility for final submissions. These are assessable forms of judgement.

The framework also prioritises ex ante assessment design over primary reliance on post hoc misconduct detection. Clear AI-access conditions help students understand what is permitted. Mixed assessment evidence reduces reliance on detection. Verification tasks make ownership and understanding visible. Process evidence can support learning rather than only policing. This does not eliminate misconduct, but it reduces the extent to which score validity depends on detecting it after submission.

These design commitments meet real institutional friction, and a framework that ignores that friction risks being adopted in name only. The most immediate constraint is workload: verification tasks, oral walkthroughs, and controlled modifications generate credible evidence precisely because they are individualised, but individualisation scales poorly. In large cohorts, live checks and viva-style confirmation may be feasible only if sampled, automated, or compressed into short structured formats, and marking reliability across multiple assessors becomes a moderation problem in its own right. A second constraint is equity of access. If AI-integrated tasks assume that all students can reach comparable tools, differences in institutional licensing, hardware, or connectivity may introduce construct-irrelevant variance — advantaging students for their access rather than their judgement. A third constraint is staff capability and institutional policy: specifying AI-access conditions per task presupposes both academic freedom to set them and staff development to design them well, neither of which can be assumed. Finally, the safeguards themselves carry a risk. Controlled conditions, documentation requirements, and verification steps can, if applied mechanically, narrow the assessed construct, penalise students who reason differently, or impose accessibility burdens that have nothing to do with coding competence. The framework is therefore best understood not as a fixed prescription but as a set of design trade-offs to be balanced against the capacity, equity, and governance realities of a given teaching context.

 ©The Authors

The article's main contribution is therefore a design framework. It distinguishes the artefact from the evidence, the evidence from the inference, and independent fluency from AI-supported competence. It helps educators ask more precise questions: Does this task assess tracing or production? Does it assess debugging or documentation? Is AI prohibited because the construct requires independence, or permitted because the construct includes tool use? What evidence would justify the mark? These questions are likely to remain relevant even as specific AI tools change.

9. Limitations and Future Research

This article is conceptual. It does not empirically validate the proposed framework, measure student AI use, analyse student performance data, or test whether framework-based assessment improves learning, fairness, or academic integrity outcomes. Its claims are therefore design claims grounded in literature, not findings from an intervention. The framework should be read as a guide for assessment design and later empirical study.

The framework also requires adaptation. Code-based courses vary widely across introductory programming, web development, data scripting, creative coding, software engineering, and advanced project work. Class size, institutional policy, staff workload, available invigilation, student access to tools, and accessibility requirements will affect which safeguards are practical. Oral walkthroughs, live coding checks, and individualised verification tasks may be feasible in small cohorts but difficult in large classes unless sampled, automated, or redesigned as shorter structured checks.

The framework's transferability across programming paradigms also warrants qualification. Its four design dimensions are intended to be paradigm-independent, but the task types that use them are not: tracing and transformation take different forms in block-based or node-and-patch environments than in text-based languages, where syntax, testing, and API use dominate, and the AI affordances and appropriate safeguards differ accordingly. A full paradigm-by-paradigm treatment is beyond the present scope and is left to future work. Relatedly, established instructional-design frameworks could extend the model by specifying performance standards, contexts, and progression; integrating that literature, and testing whether the framework transfers to non-programming performance assessment, is a promising directionthat is not pursue here.

AI capabilities are changing quickly. A task that is difficult for current tools may become easier within a short period. This is one reason the article avoids defining resilience as AI-proofing. The more durable contribution is the alignment logic: specify the construct, define the AI-access condition, choose the task type, and state what evidence supports the learning claim.

Future research should examine the framework in real courses. Useful studies could compare relationships between take-home project performance and controlled fundamentals tests, investigate whether constrained transformation tasks reveal understanding more effectively than blank-page assignments, and evaluate student perceptions of fairness under different AI-access conditions. Further work should examine staff workload, moderation reliability, accessibility, and equity implications. Comparative studies across web development, CS1, software engineering, creative coding, and digital arts courses would help refine the framework and identify discipline-specific adaptations.

10. Conclusion

Generative AI changes what coding assessments can credibly evidence. It does not make programming knowledge obsolete, and it does not mean that all take-home coding assignments should be abandoned. It does mean that educators can no longer assume that a polished coding artefact is direct evidence of independent coding competence. A website, app, script, or repository may be valuable evidence, but only for claims that the assessment conditions and supporting evidence can justify.

This article has argued that the central issue is assessment validity. Code-based courses assess multiple constructs: code reading, tracing, debugging, modification, explanation, decomposition, implementation, integration, testing, documentation, and professional workflow. These constructs require different forms of evidence. The article therefore proposed a framework of AI-resilient assessment design linking programming construct, AI-access condition, task type, evidence produced, validity risk, and safeguard.

AI-resilient assessment should combine assessment families rather than searching for a single solution. Controlled fundamentals assessments can evidence independent fluency. Constrained transformation assessments can evidence comprehension, debugging, and modification. Process-evidenced projects can preserve authentic production while making development visible. AI-documented assessments can support

transparency and accountability. AI-integrated professional assessments can evaluate students' ability to critique, test, and improve AI-generated code.

The goal is not to eliminate AI from learning. Nor is it to allow AI-supported production to replace evidence of understanding. The goal is to design assessment mixes that make credible, transparent claims about what students know and can do. In the generative AI era, the strongest coding assessments will not be those that merely resist tools, but those that align tools, tasks, evidence, and learning claims with care.

Ethics Statement: This article is conceptual and does not involve human participants, student data, or identifiable individuals. Ethical approval was therefore not required.

AI use Statement: No AI tools were used in the ideation, analysis, or writing of this article.

References

Ali, A. and Wibowo, K., 2023. Assessment of ChatGPT-generated programming code based on exercises in an introductory programming course. *Issues in Information Systems*, 24 (2), pp 203-212. Available at <https://doi.org/10.48009/2_iis_2023_117>.

Biggs, J., 1996. Enhancing teaching through constructive alignment. *Higher Education*, 32 (3), pp 347-364. Available at <https://doi.org/10.1007/BF00138871>.

Biggs, J. and Tang, C., 2011. *Teaching for quality learning at university*. 4th ed. Maidenhead: Open University Press.

Ericson, B.J., Denny, P., Prather, J., Duran, R., Hellas, A., Leinonen, J., Miller, C.S., Morrison, B.B., Pearce, J.L. and Rodger, S.H., 2022. Parsons problems and beyond: systematic literature review and empirical study designs. In: *2022 ITiCSE Working Group Reports (ITiCSE-WGR '22)*. New York: Association for Computing Machinery. Available at <https://doi.org/10.1145/3571785.3574127>.

Gulikers, J.T.M., Bastiaens, T.J. and Kirschner, P.A., 2004. A five-dimensional framework for authentic assessment. *Educational Technology Research and Development*, 52 (3), pp 67-86. Available at <https://doi.org/10.1007/BF02504676>.

Izu, C., Schulte, C., Aggarwal, A., Cutts, Q., Duran, R., Gutica, M., Heinemann, B., Kraemer, E., Lonati, V., Mirolo, C. and Weeda, R., 2019. Fostering program comprehension in novice programmers: learning activities and learning trajectories. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pp 27-52. Available at <https://doi.org/10.1145/3344429.3372501>.

Jaakkola, E., 2020. Designing conceptual articles: four approaches. *AMS Review*, 10, pp 18-26. Available at <https://doi.org/10.1007/s13162-020-00161-0>.

Kane, M.T., 2013. Validating the interpretations and uses of test scores. *Journal of Educational Measurement*, 50 (1), pp 1-73. Available at <https://doi.org/10.1111/jedm.12000>.

Kasneci, E., Sessler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., Groh, G., Günnemann, S., Hüllermeier, E., Krusche, S., Kutyniok, G., Michaeli, T., Nerdel, C., Pfeffer, J., Poquet, O., Sailer, M., Schmidt, A., Seidel, T. and Kasneci, G., 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences*, 103, Article 102274. Available at <https://doi.org/10.1016/j.lindif.2023.102274>.

Khan, M.F.A., Ramsdell, M., Falor, E. and Karimi, H., 2024. Assessing the promise and pitfalls of ChatGPT for automated CS1-driven code generation. In: *Proceedings of the 17th International Conference on Educational Data Mining*. Available at <https://doi.org/10.5281/zenodo.12729778>.

Liang, W., Yuksekgonul, M., Mao, Y., Wu, E. and Zou, J., 2023. GPT detectors are biased against non-native English writers. *Patterns*, 4 (7), Article 100779. Available at <https://doi.org/10.1016/j.patter.2023.100779>.

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L., 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36 (4), pp 119-150. Available at <https://doi.org/10.1145/1041624.1041673>.

Lodge, J.M., Howard, S., Bearman, M., Dawson, P. and Associates, 2023. *Assessment reform for the age of artificial intelligence*. Australia: Tertiary Education Quality and Standards Agency. Available at <https://www.teqsa.gov.au/sites/default/files/2023-09/assessment-reform-age-artificial-intelligence-discussion-paper.pdf>.

Messick, S., 1989. Validity. In: R.L. Linn, ed. *Educational measurement*. 3rd ed. New York: American Council on Education/Macmillan, pp 13-103.

Parsons, D. and Haden, P., 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In: *Proceedings of the 8th Australasian Conference on Computing Education*, pp 157-163.

Perkins, M., Furze, L., Roe, J. and MacVaugh, J., 2024. The Artificial Intelligence Assessment Scale (AIAS): a framework for ethical integration of generative AI in educational assessment. *Journal of University Teaching and Learning Practice*, 21 (6). Available at <https://doi.org/10.53761/q3azde36>.

Robins, A., Rountree, J. and Rountree, N., 2003. Learning and teaching programming: a review and discussion. *Computer Science Education*, 13 (2), pp 137-172. Available at <https://doi.org/10.1076/csed.13.2.137.14200>.

Sorva, J., 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13 (2), Article 8. Available at <https://doi.org/10.1145/2483710.2483713>.

 ©The Authors

UNESCO, 2023. *Guidance for generative AI in education and research*. Paris: UNESCO. Available at <https://doi.org/10.54675/EWZM9535>.

Weinman, N., Fox, A. and Hearst, M.A., 2021. Improving instruction of programming patterns with faded Parsons problems. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, Article 53. Available at <https://doi.org/10.1145/3411764.3445228>.

Wiggins, G., 1990. The case for authentic assessment. *Practical Assessment, Research, and Evaluation*, 2 (2). Available at <https://doi.org/10.7275/ffb1-mm19>.

Winslow, L.E., 1996. Programming pedagogy: a psychological overview. *ACM SIGCSE Bulletin*, 28 (3), pp 17-22. Available at <https://doi.org/10.1145/234867.234872>.

Xue, Y., Chen, H., Bai, G.R., Tairas, R. and Huang, Y., 2024. Does ChatGPT help with introductory programming? An experiment of students using ChatGPT in CS1. In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, pp 331-341. Available at <https://doi.org/10.1145/3639474.3640076>.