International Journal of Mathematics and Computer Research
ISSN: 2320-7167
Volume 13 Issue 09 September 2025, Page no. – 5663-5671
Index Copernicus ICV: 57.55, Impact Factor: 8.615
DOI: 10.47191/ijmcr/v13i9.10

AI-Assisted Coding: Evolution or Erosion of Software Development Skills?

Augustin NYEMBO MPAMPI[1], Richard ILUNGA KALANDA[2], Pélagie MPEMBA MUKONKOLE[3], Annie EBAMBI LUKOMBE[4], Germain MULUMBA MULONDO[5], Clarisse NGOYI TSHITE[6]
[1]Graduate in Computer Science and Independent Researcher
[2,3,4,5,6]Assistant at Notre Dame de Lomami University, Faculty of Computer Science

ARTICLE INFO
Published Online:
30 September 2025

Corresponding Author:
Augustin NYEMBO MPAMPI

ABSTRACT
In the era of generative artificial intelligence, tools such as GitHub Copilot and ChatGPT are profoundly transforming software development practices. This technological advancement, while facilitating the automation of complex tasks and improving productivity, raises a crucial question: *does AI-assisted coding represent an evolution or an erosion of programming skills?* This article explores the cognitive, pedagogical, and professional effects of AI assistance in software development. Through a structured analysis, it highlights the proven benefits of AI (speed, error reduction, work comfort), while highlighting the risks associated with non-critical use, including the reduction of algorithmic reasoning, cognitive dependence, and loss of creativity.
The study shows that the role of the developer is evolving: they are becoming supervisors, critics, and strategists, rather than mere executors. To support this shift, the article suggests concrete avenues: rethinking training curricula, training in prompt engineering, establishing a culture of human-machine collaboration, and promoting an ethic of increased responsibility.

KEYWORDS: Artificial intelligence, assisted coding, algorithmic reasoning, software development, skill erosion, cognitive autonomy, prompt engineering, computer training, human-machine collaboration, increased responsibility.

INTRODUCTION

The advent of generative artificial intelligence marks a decisive step in the evolution of software development. Tools like GitHub Copilot, ChatGPT, Amazon CodeWhisperer, and Tabnine are now able to assist developers in writing, completing, documenting, and correcting source code, thanks to powerful language models trained on huge data sets. In just a few years, these assistants have become established in developers' work environments, both in the workplace and in academia, disrupting traditional programming methods.

While these tools promise a significant increase in productivity, a reduction in common errors, and the automation of repetitive tasks, they also raise fundamental questions about their impact on developers' core skills. Algorithmic reasoning, solution modeling, cognitive autonomy, and technical creativity, which are at the heart of the programming profession, risk being gradually relegated to the background if the use of these assistants becomes systematic, unsupervised, or a substitute.

In this context, leading figures in the sector, such as Thomas Dohmke, CEO of GitHub, insist on the importance of preserving the basics: *"knowing how to code manually remains an irreplaceable skill* ." This statement highlights a growing tension between technological evolution and the potential erosion of know-how. Indeed, while AI can alleviate certain cognitive loads, it can also lead to forms of unlearning, functional dependency, and dehumanization of the creative process, particularly among novice or inexperienced developers.

It is therefore legitimate to ask: does AI-assisted coding represent an evolution of practices or a threat to fundamental skills? What know-how is at risk of being eroded, and how can it be preserved in a now highly automated environment? How can we train tomorrow's developers to interact intelligently with powerful tools without giving in to cognitive passivity? And above all, how can we establish a balanced and responsible relationship between human and artificial intelligence in software design processes?

This article offers an in-depth analysis of these issues through three axes: first, the study of the transformations induced by AI in contemporary software development ; then, the examination of the cognitive and pedagogical impacts on the

developer 's skills ; finally, the formulation of training, supervision and collaboration strategies aimed at establishing augmented cohabitation, based on complementarity and not substitution. The objective is to contribute to a lucid and critical understanding of the place that AI should occupy in coding, so that it remains a tool for amplifying human capacities, and not a factor of technical regression.

I. ARTIFICIAL INTELLIGENCE IN SOFTWARE DEVELOPMENT: A MAJOR TECHNOLOGICAL TURNING POINT

1.1. Overview of AI-assisted coding tools

In recent years, artificial intelligence has emerged as a major transformative force in software development environments. Its integration into modern code editors has led to the emergence of a new category of tools called intelligent programming assistants, capable of automatically generating code, suggesting corrections, or supporting the implementation of complex structures. These systems rely primarily on large language models (LLMs), trained on huge corpora of source code, technical documentation, and natural language texts.

Among the most iconic tools is GitHub Copilot, developed jointly by GitHub and OpenAI. Powered by the Codex model, itself derived from GPT-3, Copilot is designed to work in real time within integrated development environments (IDEs), including Visual Studio Code. Its operation is based on a contextual analysis of the code being written, from which it automatically offers suggestions, complete functions, or even entire application structures [1].

ChatGPT, on the other hand, although originally designed as a general-purpose conversational agent, is now widely used for code generation and explanation. Thanks to its natural language processing capabilities, it can respond to human-language queries such as: "write me a function in Python to sort a list with the quicksort algorithm", while explaining the underlying logic, advantages, or possible variations. This dialogic approach makes ChatGPT a popular teaching tool in both educational and industrial settings [2].

Alongside these two dominant solutions, other specialized tools have emerged, also integrating artificial intelligence into the developer workflow. Amazon CodeWhisperer, for example, offers a solution similar to Copilot, focused on the AWS ecosystem and DevOps integration. Tabnine, based on models trained specifically for multi-language code completion, emphasizes data privacy and personalized learning. Codeium offers an open-source alternative, integrating local models and optimized for speed [3].

All these tools have in common the use of advanced deep learning techniques, notably transformers, to analyze partial contexts (a few lines of code or comments) and generate coherent complements, often very close to developers' expectations. These technologies are made possible by the massive availability of open source code on platforms like GitHub, GitLab or Bitbucket, which provide models with a nearly exhaustive learning base on contemporary programming standards.

It's worth noting that the use of these assistants isn't limited to basic tasks. They can intervene at more abstract levels of development, suggesting class structures, generating unit tests, and even helping with automatic function documentation. Some tools also include intelligent refactoring or vulnerability detection features, expanding their scope beyond simple code writing.

Thus, the landscape of AI-assisted coding tools reveals a profound upheaval in the way code is produced. These increasingly powerful and ubiquitous systems are redefining the traditional roles of the developer, while imposing new demands in terms of mastery, control, and critical thinking.

1.2. Recognized advantages: speed, automation, reduction of errors

One of the most frequently cited arguments for integrating artificial intelligence into development environments is the significant improvement in productivity. Indeed, AI coding assistants allow for the rapid generation of code segments, the automation of repetitive tasks, and the reduction of human errors, particularly in routine tasks. These benefits are not simply perceived; they are documented by several empirical studies and industrial experiments.

a) Speed and increased productivity

According to a 2022 study conducted by GitHub, developers using GitHub Copilot for standard coding tasks observed a reduction in task completion time ranging from 30% to 55%, compared to a control group not using the tool [1]. The same study highlights that developers experience less mental fatigue, allowing them to focus on more complex and strategic tasks. This speed gain is particularly visible in the generation of repetitive structures (such as getters/setters, loops, input checks), but also in the initial writing of prototypes or unit tests.

Furthermore, a Microsoft survey of over 2,000 developers found that 88% felt Copilot saved them time, and 74% said they were able to focus more on more interesting aspects of their work [2]. These figures reflect a shift in the way people relate to development time, particularly in agile and DevOps environments, where iteration speed is a major issue.

b) Automation of repetitive and routine tasks

AI assistants are particularly effective at automatically performing well-known coding sequences, often considered tedious or of low intellectual added value. These include:

- Generation of standard functions (CRUD);
- Writing models or controllers in MVC architectures;
- Creation of unit test structures;
- Automatic documentation of functions, based on existing code.

This level of automation frees developers from the cognitive burden of mechanical tasks, allowing them to focus on architecture, optimization, or solving novel problems. As J.

Lin et al. note in a 2023 study, AI acts as an augmented working memory, making well-designed code patterns immediately accessible without the effort of searching or rewriting [3].

c) Reduced errors and improved code quality

Another notable benefit is the reduction of syntactic, logical, and typo errors. AI models are trained on huge corpora of validated and compilable source code, allowing them to suggest more robust alternatives, warn of dangerous practices, or automatically fix typical bugs.

Additionally, suggestions are often accompanied by explanations of best practices, encouraging cleaner, more modular, and maintainable code. Some platforms (like Tabnine or CodeWhisperer) even integrate security vulnerability detection systems, flagging risky patterns like SQL injections or memory management errors.

However, it should be noted that the quality of these suggestions depends heavily on the context provided and the clarity of the comments or variable names used. Therefore, a skilled developer is still required to interpret, adjust, or reject certain suggestions.

In summary, the benefits of AI-assisted coding are well established in terms of execution speed, cognitive offloading, and functional reliability, contributing to a measurable improvement in productivity and software quality. However, these technical benefits should not overshadow the deeper issues related to skills development, which we will address in the next chapter.

1.3. Changing role of the developer

The massive introduction of artificial intelligence tools into development environments is not only changing technical practices; it is also leading to a profound transformation in the professional identity of the developer. The latter is no longer simply a code producer, but is gradually becoming a supervisor of algorithmic generation, a critical validator, and an architect of software consistency. This evolution is changing expectations, responsibilities, and, above all, the fundamental skills associated with the profession.

a) From the executing coder to the critical supervisor

Historically, software development relied on the linear and sequential mastery of the coding process: design, line-by-line writing, testing, and debugging. With AI assistants, much of this writing is now automated. The developer's role is shifting toward one of review, selection, and interpretation.

In other words, he becomes a curator of generated code, responsible for checking its consistency, robustness, security and suitability to specifications. This transition requires new critical skills, particularly in rapid code reading, detection of logical anomalies, and implicit understanding of business objectives [1].

b) Evolution of expected skills

In this new paradigm, pure syntactic skills become secondary, while high-level skills become important. These include:

- The ability to *clearly formulate an algorithmic intention* (effective prompting);
- Mastery *of software architectures* and design models;
- The ability to *debug code not written by oneself* , but generated by a probabilistic system;
- *Ethical and security vigilance* , in the face of sometimes opaque or biased suggestions.

According to G. Salvaneschi et al., this transformation imposes a new computational literacy, in which the developer must not only know how to code, but also know how to judge the code produced by a machine that is not aware of the functional context [2].

c) New emerging roles in teams

Within development teams, this shift is also reflected in a restructuring of responsibilities. Some developers are becoming generative AI experts (prompt engineers), capable of making the most of the tools to increase software production. Others are specializing in verifying, reviewing, and documenting assisted code.

In software-intensive projects, we are seeing the emergence of hybrid profiles, capable of translating business needs into instructions that can be understood by AI assistants. This intermediation between humans, functional requirements, and automatic generators constitutes a new frontier in programming [3].

d) A new relationship with knowledge and training

This shift is also redefining how developers learn and update their knowledge. AI platforms, such as ChatGPT, are becoming sources of interactive learning and sometimes even substitutes for traditional documentation. Developers learn "by doing," by experimenting with the tool, asking questions, and testing.

But this phenomenon, while increasing access to technical knowledge, also carries the risk of superficial training, focused on the immediate solution, and not on in-depth understanding. It is therefore a question of strengthening a reflective stance, so that AI serves human competence, rather than replacing it.

Ultimately, AI doesn't devalue the developer; it shifts the core of their value: less in the raw production of code, and more in design, decision-making, and anticipating the effects of the generated code. This shift requires a redefinition of the profession, but also of training standards and collaboration models within teams.

II. ALGORITHMIC REASONING, KNOW-HOW AND AUTONOMY: WHICH SKILLS ARE THREATENED?

One of the historical foundations of learning and practicing software development lies in the mobilization of fundamental cognitive skills: abstraction, modeling, logical structuring, and algorithmic reasoning. These abilities are not limited to simple mastery of a programming language, but relate to a

form of logical and procedural intelligence mobilized in problem solving. However, with the growing assistance of artificial intelligence in code generation, these skills risk being displaced, diluted, or neglected in favor of automated task execution.

2.1. The foundations of programming know-how

Programming know-how is not just about the ability to produce functional code: it is based on a structured set of cognitive, technical, and methodological skills that allow the developer to design, model, implement, and optimize software solutions adapted to various problems. These skills are the result of progressive and structured learning, mobilizing working memory, logical thinking, and the capacity for abstraction.

a) Algorithmic thinking as a central pillar

At the heart of programming know-how is what researchers call algorithmic thinking , defined as the ability to break down a complex problem into logical substeps and formulate precise instructions for solving them. This ability relies on:

- structuring (conditions, loops, iterations);
- Data flow modeling **;**
- The conceptualization of efficient and robust algorithms **.**

Wing [1] emphasizes that algorithmic thinking is not only essential for writing code, but is a transferable skill across all scientific disciplines. It allows the programmer to reason rigorously, predict the consequences of a computer action, and optimize their implementation choices.

b) Progressive learning: from syntax to abstraction

Programming instruction is based on a hierarchical progression. First, learners assimilate syntactic and lexical elements (declarations, operators, types). Next, they are introduced to procedural logic, control structures, and then functions and modularity. Finally, they learn data structures, advanced algorithms, object-oriented programming, and even functional or event-driven paradigms.

This path promotes an increase in abstraction, essential for understanding complex architectures and producing maintainable and scalable code. As Robins et al. [2] indicate, this increase in abstraction is often difficult for beginners to overcome, but it constitutes the foundation of lasting competence in software development.

c) The role of problem solving

Another fundamental aspect of know-how is the ability to formulate, analyze and solve problems using programming. This approach involves:

- *Understand a functional statement* or a real situation;
- *Translate this situation into algorithmic logic* ;
- *Test, correct and improve* a solution until it is fully validated.

This problem-solving approach requires cross-functional skills: perseverance, methodical debugging, error anticipation, and sometimes even creativity. An experienced programmer is able to propose several possible solutions to the same problem and choose the most optimal one based on time, memory, or code readability constraints.

d) Proofreading and maintenance capacity

Know-how is not limited to the initial creation of code: it also includes the ability to review, understand, and modify existing code, sometimes written by someone else. This skill, called code readability , is essential in collaborative and long-term projects. It is based on:

- Rapid identification of logical structure;
- Recognition of the initial coder's intentions;
- The ability to correct, refactor or document.

In an environment where AI-generated code is increasingly common, this skill becomes even more critical: it is no longer about writing, but about understanding code that one has not written oneself, often produced by a statistical system without explicit intention [3].

In short, programming know-how relies on a combination of high-level skills: thinking in terms of algorithms, abstracting problems, modeling solutions, structuring clean code, and reviewing and maintaining software over time. The massive introduction of artificial intelligence in software development calls into question the place and practice of these skills. It is therefore imperative to ensure that they are not marginalized or bypassed in training or production processes.

2.2. Risks of assistance and unlearning

One of the major risks associated with the increasing use of artificial intelligence in software development lies in the gradual decline of the cognitive involvement of the human developer. While AI assistants can automate certain technical tasks, they can also lead to a form of technological assistance, in which the developer relies excessively on the machine's suggestions to the detriment of their own reasoning. This phenomenon paves the way for gradual unlearning, which is particularly worrying for beginners.

a) Cognitive dependence on code suggestions

Assistants such as GitHub Copilot or ChatGPT provide real-time suggestions based on the context of the code or a natural language instruction. This feature, while convenient, tends to reduce the mental effort required to produce code. Several studies have shown that when a solution is immediately proposed, the user tends to accept it without questioning its relevance or internal logic.

According to Fuhry et al. [1], in a Copilot-assisted learning environment, more than 60% of students accept generated suggestions without modifications, even when they contain logical errors or inefficient implementations. This behavior reveals a form of cognitive passivity, where the developer ceases to play an active role in the algorithmic design process. Additionally, automating suggestions can lead to decreased vigilance for potential bugs or flaws, with the developer wrongly assuming that the generated code is reliable by default.

b) Erosion of algorithmic reasoning among beginners

The problem is even more critical in the educational context. Computer science learners need to develop fundamental skills such as problem modeling, algorithm design, and code structuring. However, when AI anticipates or replaces these steps, the student may be tempted to skip essential thinking phases, simply copying and pasting the proposed solution.

Sarsa et al. [2] observed that students who regularly used AI assistants performed worse on exams that focused on conceptual understanding and critical analysis of code, compared to those who developed their solutions without assistance. This suggests that AI, when used as a substitute for active learning, can impair cognitive anchoring of knowledge.

Furthermore, the systematic use of completion tools reduces the exercise of working memory, which is essential in the manipulation of abstract elements such as variables, pointers, or recursive structures.

c) Long-term unlearning and the risk of "passive developer"

Beyond initial learning, the phenomenon of assistantship can continue in professional environments. Experienced developers can gradually lose certain technical skills if they come to systematically delegate design or implementation tasks to AI tools. This silent unlearning can lead to:

- A decline in the capacity for algorithmic innovation;
- Difficulty understanding or correcting complex errors;
- Increased dependence on proprietary tools, reducing technical autonomy.

According to Zhang et al. [3], the phenomenon of delegated competence becomes problematic when the tool becomes a "black box", which the developer no longer understands and on which he can no longer intervene in the event of a malfunction.

Ultimately, while artificial intelligence can play a valuable role in assisting software development, it carries a real risk of cognitive disengagement, especially if its use is not accompanied by educational guidance or critical use. The developer must remain an active, reflective, and responsible player in the construction of the code, and not simply a validator of automated suggestions.

2.3. Dehumanization of the creative process?

Programming is not just a mechanical succession of executable instructions. It is, to a large extent, an act of intellectual and technical creation, mobilizing not only logical skills but also an expressive, aesthetic, and strategic dimension. However, the massive introduction of artificial intelligence into the coding process raises a fundamental question: *does AI standardize code to the point of neutralizing human creativity?*

a) Creativity in programming: an often underestimated reality

In the field of software development, creativity manifests itself at several levels:

- *The choice of algorithmic structures* among many possible solutions;
- *The logical and functional organization of the code* taking into account performance, readability or reusability constraints;
- *The ability to produce elegant and original code* , responding to specific constraints in an innovative way.

As McBreen [1] points out, a good developer is often compared to a craftsman or an architect, whose style, technical choices and quality of execution demonstrate unique expertise. Code thus becomes a language of expression, reflecting conceptual preferences and technical convictions.

b) Statistical standardization of AI suggestions

AI assistants like GPT or Codex generate code from probabilistic models, trained on billions of lines from open source repositories. By definition, their suggestions are the product of a statistical compromise between thousands of similar versions of solving the same problem. This leads to:

- Often correct solutions, but not very original;
- A stylistic smoothing that tends to standardize coding practices;
- A marginalization of unconventional or experimental approaches.

According to Allamanis [2], this phenomenon risks creating an "averageization" of programming practices, where the generated solutions become predictable, to the detriment of algorithmic exploration. The developer, by passively accepting the proposals of AI, exposes himself to a loss of control over the construction of his own logic.

c) Risk of loss of meaning and appropriation of the code

Another important consequence is emotional and intellectual disengagement from the code. When a developer manually writes a program, they develop a detailed understanding of its structure, its potential flaws, and its strengths. This appropriation of the code is crucial, both for correction and for long-term optimization or maintenance.

On the other hand, AI-generated code can appear foreign, even opaque, especially if it is accepted without modification or reformulation. Zhang et al. [3] speak of a "black box effect" in which the code becomes a utilitarian artifact, disconnected from the reasoning that justifies it. This phenomenon weakens technical responsibility and awareness of the potential impact of the software on its environment.

d) The gradual extinction of experimentation and intuition

Creating new software solutions also relies on active experimentation: trying, failing, modifying, observing the effects. However, if AI systematically provides immediate solutions, it short-circuits this exploratory dynamic. Learners, like professionals, risk losing the reflex to explore other

avenues, compare multiple strategies, or refine an algorithm according to personalized criteria.

Finally, elements like code intuition—the ability to sense an error, anticipate an optimization, or guess a relevant architecture—rely on the accumulated experience of writing by hand. When this experience is reduced, intuition freezes, and the developer becomes less responsive to the unexpected, less creative when faced with the new.

Ultimately, the widespread use of AI in software development, if poorly managed, can lead to a gradual dehumanization of the creative process, where code ceases to be a thoughtful human work and becomes an automated response to formalized problems. This trend must be carefully questioned in order to preserve the element of freedom, invention, and responsibility that makes the developer a true designer.

III. RETHINKING PRACTICES AND TRAINING FOR ARTIFICIAL INTELLIGENCE IN THE SERVICE OF SKILLS

Faced with the risks identified in the previous points—erosion of know-how, cognitive dependence, loss of creativity—it is essential to devise a structured and proactive response. The objective is not to reject the use of AI assistants in software development, but to redefine the modalities of their integration so that they serve to strengthen human skills rather than short-circuit them. This implies a rethinking of teaching practices, a revaluation of the intellectual autonomy of the developer, and a new culture of assisted coding, anchored in responsibility and reflexivity.

3.1. Towards a critical and controlled use of AI assistance

The widespread adoption of AI-based coding assistants requires a shift in usage: from passive consumption to critical and reflective use. It's no longer just a matter of interacting with powerful tools, but of developing a methodological and ethical stance in the face of automatic suggestions that, while functional, are neither guaranteed, optimized, nor neutral. To take advantage of AI without falling into a form of technological assistance, it is essential to define the conditions for controlled use, framed by human competence.

a) Promoting reflexivity in everyday use

The first principle of critical use is based on the development of reflexivity: the developer must never accept a suggestion without asking fundamental questions:

- *Does this code match my original intention?*
- *Is it optimal in terms of performance, readability, security?*
- *Could I have designed a better solution?*

As Salvaneschi et al. [1] point out, a competent developer in the face of AI is one who "takes control of the algorithmic process," considering each suggestion as a hypothesis to be evaluated, not as a truth to be applied. This involves establishing a critical distance between the human's intention and the machine's response.

b) Develop evaluation grids for the generated code

One concrete way to regulate the use of AI in development is to implement tools for systematically analyzing suggestions, whether manual or automated. These grids can include criteria such as:

- *Logical consistency* : Does the code follow the problem specifications?
- *Readability* : Is it understandable to another developer?
- *Security* : Does it contain vulnerable patterns or unverified calls?
- *Performance* : Is algorithmic complexity under control?

This approach is inspired by software engineering best practices and can be applied to AI-generated code as well as any other human-produced artifact. It reminds developers that they remain responsible for the quality of the delivered software, regardless of the tool used.

c) Master the art of effective "prompting"

Critical use of AI also requires mastery of the language used to interact with it. In the case of tools like ChatGPT or Copilot, the precision, clarity, and level of detail of the instructions directly influence the quality of the generated code. Hence the importance of prompting training, that is, the art of formulating effective, ethical, and contextualized requests.

Dastin [2] points out that prompting is becoming a new form of technical skill, close to functional specification, which requires rigor, conciseness and anticipation. It not only optimizes the results of AI, but also strengthens the developer's own understanding of the problem.

d) Maintain a balance between autonomy and assistance

Finally, controlled use of AI assistance requires maintaining a dynamic balance between delegation to the machine and the intellectual autonomy of the developer. This involves:

- Code without assistance when learning or analysis requires it **;**
- Limit the use of AI suggestions in exploratory or critical phases **;**
- Use AI as a "second set of eyes ," a secondary assistant, not a substitute for thinking.

The developer must remain the decision-making center of gravity, able to deactivate the wizard, correct it, or ignore it, depending on the needs of the project.

In short, a critical and controlled use of AI in software development requires cultivating a posture of vigilance, competence, and responsibility. AI can enrich the programming process, but only if the developer remains the conscious and active author of the logic implemented.

3.2. Redefining training curricula in development

The rise of artificial intelligence programming assistants is disrupting traditional computer science training frameworks. Faced with this paradigm shift, it is imperative to rethink teaching curricula, both at the university level and in

continuing education programs, to preserve fundamental skills, foster the intellectual autonomy of future developers, and integrate AI strategically and critically.

a) Maintain a rigorous foundation of manual coding learning

Software development, as a discipline, has historically relied on the gradual learning of algorithmic reasoning, data structures, and programming paradigms. In this context, hand coding must remain central, particularly in the early stages of the curriculum. This is to ensure that the student masters:

- Conditional and iterative logic;
- Modularity and factorization of the code;
- Manual resolution of algorithmic-mathematical problems;
- Error management, debugging and traceability.

Several studies show that early exposure to AI assistance, without strict didactic supervision, can hinder the development of these basic skills. As Leinonen et al. [1] demonstrated, students who had access to tools like Copilot from the first months tended not to develop a deep understanding of control mechanisms and data structures.

b) Integrate AI into a comparative and reflective logic

Rather than banning AI assistants from training spaces, it is better to integrate them in a gradual and controlled manner, as tools for analysis, confrontation, and comparison. This can result in:

- Exercises where the student first produces a manual solution, then compares it to that of the AI;
- Directed work aimed at *criticizing the generated code* : identifying limits, correcting flaws, proposing alternatives;
- *"reverse engineering"* workshops : having the student explain how a generated code works, line by line.

This approach encourages the development of a critical and active posture, and makes it possible to transform AI into an educational tool for learning, rather than a cognitive shortcut.

c) Train in prompt engineering and interpretation of AI code

With generative AI, a new skill is emerging: prompt engineering, the ability to effectively interact with a language model to obtain relevant answers. This skill is now essential, particularly in professional environments where AI is becoming an extension of development tools.

Curricula must therefore include:

- Effective query formulation techniques (clarity, precision, contextualization) ;
- Training in interpreting the generated code: understanding, validating, reformulating if necessary;
- Teaching good ethical and security practices related to the use of AI assistants.

As Dastin [2] points out, prompt engineering should be considered as an extension of software design, requiring rigor and mastery, at the border between natural language and programming logic.

d) Adapt the assessment to new tools

Finally, assessment methods must evolve to accommodate the presence of AI without encouraging cheating or passive plagiarism. This may include:

- Oral assessments focused on the student's ability to explain their approach;
- Homework to be completed in a restricted environment (without access to AI), to test basic skills;
- Projects where the use of AI is authorized but explicitly justified: why such code was generated, modified, validated or rejected.

This type of assessment values active understanding, decision making, and the ability to contextualize code instead of copying it.

In short, redefining training curricula doesn't mean going backward or rejecting AI, but rather integrating this new reality into a pedagogy that strengthens human competence. AI must be an accelerator of understanding, not a substitute for thought. This revision of content, tools, and teaching methods is essential to prepare future developers for a digital world where automation coexists with creativity and rigor.

3.3. Towards increased cohabitation: complementarity rather than substitution

AI-assisted software development requires a redefinition of the relationship between humans and machines. It's not a matter of replacement, but rather of augmented coexistence, in which human skills and computational capabilities complement each other to produce higher-quality code faster, while preserving the developer's intellectual autonomy. This approach is based on several essential pillars.

a) Define distinct but complementary roles

AI cannot—and should not—occupy the same place as human intelligence in the development process. The roles must remain clearly differentiated. Humans are the primary decision-makers and designers, responsible for technical choices, functional analysis, and software architecture. Artificial intelligence, on the other hand, acts as a support tool: it proposes solutions, suggests alternatives, or automates repetitive portions of code. This functional separation makes it possible to leverage the respective strengths of each: AI offers speed and completeness, while humans provide judgment, contextualization, and accountability. Brundage et al. emphasize the need for explicit mechanisms to ensure that humans retain decision-making control over AI-assisted systems [1].

b) Refocus the developer on high value-added skills

Automation enabled by AI tools can free the developer from many tedious technical tasks, such as generating standard structures, initial documentation, or correcting simple errors. This time saved must be reinvested in activities with high intellectual value, such as designing innovative solutions,

optimizing performance, code security, or system interoperability. The modern developer thus becomes a solution architect, equipped with transversal skills, at the crossroads of technology, logical reasoning, and ethics. This evolution towards augmented human intelligence has been analyzed by Allamanis as an opportunity to rethink the role of the developer as a creator and strategist, and not just an executor [2].

c) Establish a culture of human-machine collaboration

The success of this cohabitation requires a cultural shift in development teams and organizations. It is not enough to introduce AI into the technical environment; individuals must also be trained to collaborate with the machine. This involves developing appropriate work processes, such as the systematic documentation of AI suggestions used, the implementation of code reviews specific to automatically generated portions, or even comparative analysis between human and machine code. These practices strengthen both software quality and the traceability of decisions. Guo et al. thus propose collaborative pipelines where the generation, interpretation, and validation stages are clearly sequenced and assigned to humans or AI according to their nature [3].

d) Prepare an ethics of increased responsibility

The presence of AI in the coding process also raises ethical issues. It is becoming essential that the developer does not consider the machine-generated code as neutral or infallible. On the contrary, they must exercise increased responsibility, as the final validator of the product. This means ensuring that the code does not contain security flaws, that it respects the principles of non-discrimination, and that it is understandable by other humans. AI can help produce code faster, but only human intelligence can assume the social, economic, and legal consequences. This vision of increased responsibility is part of a digital trust approach where transparency, traceability, and auditability are becoming standards [1].

Conclusion

The introduction of artificial intelligence into software development, through assistants such as GitHub Copilot or ChatGPT, marks a decisive step in the history of programming. While these tools promise unprecedented productivity gains, valuable assistance in writing code, and a reduction in routine errors, they also raise profound questions about the future of **fundamental computer science skills** . AI-assisted coding, far from being a simple technical evolution, challenges the **cognitive, pedagogical, and ethical foundations** of the developer profession.

Throughout our analysis, we have highlighted a double movement: on the one hand, a **change in the role of the programmer** , called upon to become an intelligent supervisor, a strategic designer, a critical evaluator; on the other, a **real risk of cognitive disengagement** , technological assistance and unlearning of essential skills such as algorithmic reasoning, logical structuring or functional creativity.

Faced with this tension between **evolution** and **erosion** , it is becoming imperative to **rethink training practices and systems** . AI should not be a substitute for learning, but a **lever for strengthening human skills** . This involves maintaining the importance of manual coding in the initiation phases, encouraging critical and supervised use of generative tools, and training developers to interact intelligently with language models, in a reflective stance.

More broadly, the challenge is not to choose between human code and machine code, but to build a **hybrid intelligence** , where humans retain control, meaning and responsibility for the software. In this context, the developer of tomorrow will not be replaced by AI, but by **another developer who will know how to use it intelligently, without losing their know-how** . The challenge is therefore less technical than epistemological: it is about ensuring that mastery, creativity and critical thinking remain at the heart of software development in the era of automation.

REFERENCES

1. GitHub, "Quantifying GitHub Copilot's impact on developer productivity and happiness," GitHub Blog, 2022. [Online]. Available: https://github.blog/2022-06-29-github-copilot-research
2. Microsoft, "The developer productivity paradox," Microsoft Research, 2022. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-developer-productivity-paradox
3. J. Lin, H. Zhang, Y. Wang, and T. Gu, "Intelligent Code Completion: Productivity, Learning, and Risk," *IEEE Transactions on Software Engineering* , vol. 49, no. 3, pp. 1632–1645, Mar. 2023.
4. S. Amershi et al., "Guidelines for Human-AI Interaction," *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* , pp. 1–13, May 2019.
5. G. Salvaneschi, B. Liskov, and M. Mezini, "From Coding to Prompting: Redefining the Developer's Role in the Age of AI," *IEEE Software* , vol. 41, no. 1, pp. 10–17, Jan. 2024.
6. D. Cito, G. Pinto, and MD Penta, "Prompt Engineering for Software Development: Challenges and Research Opportunities," *Proceedings of the 2023 International Conference on Software Engineering (ICSE)* , pp. 1023–1034, 2023.
7. M. Vainio, A. Knutas, and P. Ihantola, "The Role of Cognitive Skills in Learning Programming: A Systematic Review," *IEEE Transactions on Education* , vol. 65, no. 2, pp. 195–204, May 2022.
8. G. Fuhry, S. Radhakrishna, and T. Nguyen, "Code Without Understanding? A Study on the Effects of GitHub Copilot in Introductory Programming,"

Proceedings of the ACM Global Computing Education Conference (CompEd) , pp. 112–120, 2023.

9. J. Sarsa, J. Leinonen, and V. Isomöttönen, "Learning with AI Code Generators: Effects on Problem Solving and Knowledge Retention," *IEEE Transactions on Learning Technologies* , vol. 17, no. 1, pp. 56–68, 2024.
10. Y. Zhang, T. Guo, and H. Chen, "Black-Box Dependency and Skill Degradation in AI-Assisted Programming," *IEEE Software* , vol. 40, no. 6, pp. 42–50, Nov.–Dec. 2023.
11. P. McBreen, *Software Craftsmanship: The New Imperative* , Boston, MA: Addison-Wesley, 2001.
12. M. Allamanis, "The future of code is generated: Challenges and opportunities," *Communications of the ACM* , vol. 66, no. 2, pp. 34–36, Feb. 2023.
13. A. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education* , vol. 13, no. 2, pp. 137–172, 2003.
14. A. Kalliamvakou et al., "Code Comprehension Challenges in the Era of AI-Generated Software," *IEEE Software* , vol. 40, no. 5, pp. 22–30, Sept.–Oct. 2023.
15. JM Wing, "Computational Thinking," *Communications of the ACM* , vol. 49, no. 3, pp. 33–35, Mar. 2006.
16. J. Leinonen, A. Knutas, and P. Ihantola, "Blending Code Generation with Human Learning: Pedagogical Frameworks for AI-Assisted Programming," *Proceedings of the 2023 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)* , pp. 450–456, 2023.
17. AR Dastin, "Human-AI Collaboration in Creative Programming: Principles and Pitfalls," *Communications of the ACM* , vol. 66, no. 4, pp. 42–49, Apr. 2023.
18. M. Brundage et al., "Toward Trustworthy AI Development: Mechanisms for Supporting Verifiable Claims," *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society (AIES)* , pp. 1–10, 2022.
19. T. Guo, Y. Zhang, and H. Chen, "Designing Human-AI Collaboration Pipelines for Reliable Software Engineering," *IEEE Transactions on Software Engineering* , vol. 50, no. 1, pp. 144–159, Jan. 2024.