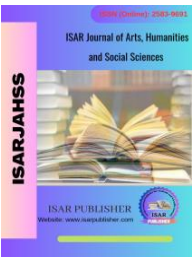
9-2024

# Bridging Disciplines with AI-Powered Coding: Empowering Non-STEM Students to Build Advanced APIs in the Humanities

Daniel Plate
*Lindenwood University*, dplate@lindenwood.edu

James Hutson
*Lindenwood University*, jhutson@lindenwood.edu




**ISAR Journal of Arts, Humanities and Social Sciences**
**Vol-2, Iss-9 (Sep- 2024)**
*Abbriviate Title- ISAR J Arts Humanit Soc Sci*
ISSN *(Online)*- 2583-9691
https://isarpublisher.com/journal/isarjahss

# Bridging Disciplines with AI-Powered Coding: Empowering Non-STEM Students to Build Advanced APIs in the Humanities

**Daniel Plate[1], James Hutson[2*]**
Lindenwood University, USA.

***Corresponding Author**
**James Hutson**

Lindenwood University, USA.

**Article History**
Received: 30.07.2024
Accepted: 21.08.2024
Published: 12.09.2024

**Abstract:** The integration of AI-powered coding assistants, such as Cursor AI, GitHub Copilot, and Replit's Ghostwriter AI, represents a transformative shift in programming education, particularly for non-STEM students. These tools democratize coding by enabling natural language code generation, intelligent error correction, and context-aware assistance within familiar coding environments. This article explores how these technologies empower educators across disciplines to introduce basic and advanced coding concepts to humanities students, a demographic traditionally underserved in programming education. By leveraging AI, instructors can teach non-STEM students the foundational principles of coding and guide them through the development of sophisticated projects, such as building APIs for literary analysis or creative world-building. These endeavors, once reserved for advanced digital humanities research, now become accessible within the framework of undergraduate humanities courses. The article examines the practical applications of AI-assisted coding in humanities education, demonstrating how these tools facilitate a deeper engagement with digital methodologies, thus expanding the horizons of what is possible in these fields. Additionally, it discusses the potential for AI-powered assistants to address the unique needs of non-STEM learners, offering a tailored educational experience that aligns with their academic and creative pursuits. This approach not only enriches the humanities curriculum but also fosters interdisciplinary collaboration, preparing students for a future where coding literacy is an essential skill across all domains.

**Keywords:** *AI-powered coding, non-STEM education, humanities, API development, digital methodologies.*

**Cite this article:**

Plate, D., Hutson, J., (2024). Satirical Deepfakes, Surreal Dreamscapes & Nostalgic Pixels: The Rapid Evolution and Cultural Commentary of AI-Aesthetics. *ISAR Journal of Arts, Humanities and Social Sciences*, *2*(9), 44-55.

## 1. Introduction

Generative AI (GAI) has profoundly reshaped educational practices, influencing everything from instructional design to student engagement (Bahroun et al., 2023). Advanced AI models like OpenAI's GPT series are not only enhancing efficiency but also altering the creative and pedagogical processes in fields such as education, curriculum development, and instructional design (Ruiz-Rojas et al., 2023; Zohuri, 2023). The ability of these generative tools to autonomously create content and offer personalized learning experiences is driving innovation in teaching methodologies, introducing new approaches that emphasize adaptability, personalization, and scalability in educational contexts (Weng & Chiu, 2023). In particular, GAI is transforming how educators approach curriculum development by facilitating the creation of tailored educational materials that can cater to diverse learning styles and levels of student proficiency. This technology allows for the rapid development of interactive learning modules, assessments, and even real-time feedback mechanisms, thereby enriching the learning experience (Yuan et al., 2023). The impact of GAI is especially significant in knowledge-based education sectors, where the automation of routine instructional tasks enables educators to focus more on strategic and creative aspects of teaching (Brynjolfsson et al., 2023).

Despite the rapid advancements in GAI, its integration into educational settings has largely remained confined to curriculum development and instructional design (Ng et al., 2023). While approximately 60% of educators have begun adopting GAI tools, their application has yet to fully penetrate the forward-facing classroom experience, particularly in teaching students how to effectively use these technologies (Hamilton & Swanston, 2024). This limited adoption is especially pronounced within Computer Science departments, which, paradoxically, have lagged behind other disciplines in embracing GAI for pedagogical purposes (Hutson & Jeevanjee, 2024). Many Computer Science programs continue to adhere to traditional methods of teaching coding and programming, missing the opportunity to incorporate GAI as a



***Corresponding Author: James Hutson**

transformative educational tool that could make coding more accessible and engaging for students (Hazzan & Erez, 2024).

Furthermore, in fields where GAI has been integrated into classroom instruction, its usage has primarily focused on generating written content rather than on teaching foundational coding skills (Tseng & Warschauer, 2023). This trend overlooks the potential of GAI to serve as a democratizing force in education, where the focus should now shift towards leveraging these tools to teach basic coding. In doing so, educators across disciplines—not just in STEM fields—can empower students with essential coding skills, making programming accessible to a broader audience and fostering interdisciplinary collaboration. The time has come for educational institutions to move beyond using GAI merely as an aid in writing and curriculum design and to explore its full potential in democratizing coding education, particularly by integrating it into everyday classroom experiences (Melro et al., 2023).

The history of how coding is taught reinforces the integration of new tools as technology evolves. For instance, since the 1980s, the evolution of software development has fundamentally transformed educational practices in programming, particularly through the transition from mainframe systems to object-oriented programming (OOP) (Holo et al., 2023; Nagineni, 2021). In the mainframe era, programming education was often confined to a specialized few, with a focus on centralized, monolithic applications managed by experts (Grütter, 2024; Megargel, Shankararaman, & Walker, 2020). The advent of OOP, however, introduced key concepts such as modularity, encapsulation, and reusability, which not only simplified the development process but also revolutionized the way programming is taught and learned (Krismadinata et al., 2023; Saide, 2024). This shift towards OOP opened the doors for a more diverse range of learners to engage with software development, as the modular nature of OOP reduced the steep learning curve traditionally associated with programming (Gutiérrez, Guerrero, & López-Ospina, 2022; Li et al., 2008). The educational landscape adapted to these changes, with curricula increasingly emphasizing the understanding of class hierarchies, inheritance, and polymorphism—core principles that underpin flexible and maintainable software design (Jablonický & Lang, 2023; Liverman, Berri, & Ben-David Kolikant, 2011). However, these advancements also introduced new complexities, particularly in teaching how to manage evolving software requirements and dependencies between objects, which are critical for maintaining large-scale systems (Kasauli et al., 2021; Mens, 2014).

In response, educators have increasingly incorporated methodologies and tools that support the iterative development and refinement of object-oriented systems, such as refactoring and the application of design patterns (Gamma et al., 1993; Rajlich, 1997). This evolution in pedagogy reflects the broader trend of adapting teaching strategies to address growing system complexities while maintaining efficiency and scalability in the learning process. Moreover, as AI has been increasingly integrated into the software industry, it is anticipated that these tools will further augment the teaching of OOP by automating routine tasks, thereby allowing educators and students to focus more on creative, strategic, and complex problem-solving activities (Santhosh et al., 2023). This mirrors historical shifts in technology, where new tools and methodologies have consistently enabled professionals to transcend routine tasks, engaging instead in higher-value educational activities (Hordern, 2018; Magana, 2017).

As the teaching of coding evolves, it is important to note that, historically, the teaching of it has remained largely within the confines of Computer Science departments, creating a disciplinary boundary that has limited the accessibility of programming skills to students outside of STEM fields (Castro, 2015). This separation has particularly affected the humanities, where the integration of coding into the curriculum has been minimal, often relegated to specialized digital humanities courses or advanced research contexts (Drucker, 2021). However, the advent of GAI has the potential to dismantle these barriers, offering an unprecedented opportunity for all disciplines to incorporate coding into their educational frameworks. These tools, such as Cursor AI, GitHub Copilot, and Replit's Ghostwriter AI, not only lower the entry barriers to basic coding but also enable non-STEM educators to introduce more advanced programming concepts, such as API development, into their courses.

This article will explore actionable strategies for integrating coding into non-STEM curricula, with a particular focus on the humanities. Through demonstration of how educators can utilize GAI to teach both foundational coding skills and complex programming tasks, such as building APIs for literary analysis or creative world-building, this work addresses a significant gap in current educational practices. The ability to teach students in the humanities how to develop APIs—a task previously reserved for advanced digital humanities research—represents a transformative shift in pedagogy. This shift not only equips students with valuable technical skills but also enhances their ability to engage with digital tools in meaningful and innovative ways. Through the examples and strategies presented in this article, educators across disciplines will find practical guidance on how to enrich their curricula, making coding an accessible and integral part of the educational experience for all students, regardless of their disciplinary background.

## 2. History of Software Development Education

The history of software development is a narrative of rapid technological progress, characterized by distinct eras that have shaped the evolution of the industry **(Table 1)**. Beginning in the 1940s and 1950s with the development of early computers, the field has undergone transformative shifts, evolving from rudimentary machine language coding to the sophisticated programming paradigms in use today. These transitions—from the mainframe systems of the mid-20th century to the more modular and flexible development methodologies that followed—have consistently reduced the complexity of coding and expanded access to software development. This ongoing progression has culminated in the highly interconnected and automated systems that define contemporary software engineering (Jadhav, Kaur, & Akter, 2022).

**Table 1. History of Coding Education**

| Era | Period | Key Developments in Teaching Coding | Key Technologies/Concepts |
|---|---|---|---|
| **Early Computing** | 1940s - 1950s | - Focus on machine and assembly language coding in academic settings.<br>- Programming done on mainframes with punch cards and manual input. | ENIAC, Von Neumann architecture, machine language, assembly language |
| **Mainframe Era** | 1960s - 1970s | - Introduction of structured programming principles into curricula.<br>- Emphasis on managing complexity with control structures. | COBOL, Fortran, Structured Programming, Batch Processing |
| **Personal Computing Era** | 1970s - 1980s | - Shift to personal computing with more accessible programming languages like BASIC.<br>- Introduction of Graphical User Interfaces (GUIs). | BASIC, MS-DOS, C++, Graphical User Interfaces (GUIs) |
| **Internet and Open Source** | 1990s - Early 2000s | - Integration of web technologies into curricula.<br>- Rise of open-source software and community-driven development practices. | HTML, JavaScript, PHP, Linux, Apache HTTP Server |
| **Agile and DevOps** | 2000s | - Shift to Agile methodologies and DevOps in software engineering education.<br>- Emphasis on iterative development and continuous deployment. | Agile, DevOps, CI/CD, Git, Jenkins, Docker |
| **AI Integration** | 2020s - Present | - Introduction of AI-powered coding tools into non-CS disciplines.<br>- Focus on democratizing coding and interdisciplinary application. | GitHub Copilot, GPT-4, ChatGPT, Generative AI, AI in Education |

In the 1940s and 1950s, software development education in colleges was characterized by its nascent stage, focusing primarily on programming using machine languages and assembly languages. Early computers like ENIAC, primarily designed for scientific and military purposes, required programmers to manually input instructions via punch cards, making programming a complex and specialized skill (Haigh & Ceruzzi, 2021). The introduction of the Von Neumann architecture during this period marked a significant advance, allowing programs to be stored in memory, which enabled sequential execution and laid the groundwork for future software development practices (Collen & Kulikowski, 2015). However, the academic focus during this era remained highly technical and limited to a small group of specialists who were often involved in pioneering hardware and software development efforts within research institutions.

As computing technology advanced, the mainframe era emerged, characterized by the dominance of large-scale computers primarily utilized by governments and large corporations. This period witnessed the development of programming languages like COBOL and Fortran, which were designed to manage business and scientific applications, respectively (Bessen, 2022). The 1960s also saw the introduction of structured programming principles, which addressed the growing complexity of software systems by promoting more maintainable and efficient code practices (Farley, 2021). Mainframes operated on a batch processing model, where tasks were queued and executed sequentially. Although this model limited interactivity, it was well-suited to the large-scale data processing needs of the time (Campbell-Kelly & Garcia-Swartz, 2015). The era also marked the beginnings of standardization in software development practices, laying the foundation for the more flexible computing systems that would follow.

In the 1960s, the introduction of structured programming principles marked a significant shift in how software development was taught in colleges. This period saw the emergence of methodologies that emphasized the importance of control structures, such as loops and conditional statements, and the avoidance of the "goto" statement, which was seen as a source of programming errors and complexity. These principles, advocated by prominent computer scientists like Edsger Dijkstra, were incorporated into computer science curricula, laying the groundwork for more disciplined and systematic approaches to software development (Dijkstra, 1996). The focus on structured programming was intended to produce code that was not only more reliable and maintainable but also easier to understand and modify. This educational shift was part of a broader movement to professionalize programming and address the "software crisis" of the time, which was characterized by frequent software failures and escalating development costs (Williams, 2013).

The 1970s and 1980s marked a significant shift in software development with the advent of personal computing, largely driven by the development of microprocessors (Khan et al., 2021). As computing power became more affordable and accessible, personal computers (PCs) began to proliferate in homes and offices. This era witnessed the popularization of operating systems like MS-DOS and the widespread adoption of programming languages such as BASIC, which made computing more approachable for both hobbyists and professionals (Bright et al., 2020). The introduction of graphical user interfaces (GUIs) with products like Apple's

Macintosh and Microsoft Windows further revolutionized software usability, making computers more intuitive for non-technical users and significantly expanding the user base (Ceruzzi, 1998). These developments not only facilitated the personal computing boom but also shifted the software industry's focus from mainframes to user-centric applications, paving the way for the democratization of computing (Barlaskar, 2020).

The evolution of software development continued to gain momentum in the 1980s with the emergence of object-oriented programming (OOP), a paradigm that introduced key concepts such as encapsulation, inheritance, and polymorphism (Koti et al., 2024). These foundational principles enabled developers to create software systems that were not only more modular and maintainable but also more scalable. The adoption of languages like C++ and later Java became widespread, facilitating the construction of complex applications with enhanced flexibility and efficiency (Ogala & Ojie, 2020). OOP marked a significant departure from procedural programming by shifting the focus to an object-based approach, where software components could be reused and managed more effectively (Dony et al., 1992). The concurrent rise of the client-server model during this era further empowered businesses to deploy enterprise-level software across interconnected systems, thereby accelerating the adoption of OOP methodologies (Sallow et al., 2020). This period underscores how the convergence of accessible personal computing and innovative programming paradigms like OOP laid the groundwork for the rapid expansion of software development, culminating in the diverse and interconnected systems that underpin today's digital infrastructure.

During the rise of personal computing, it significantly influenced how software development and programming were taught in colleges. During this era, the advent of personal computers, particularly with the introduction of IBM PCs, revolutionized the accessibility of computing resources to students and educators alike (Hunter, 1988). Courses began to incorporate languages such as BASIC, which became widely popular due to its simplicity and adaptability to the emerging personal computing platforms (Kafai & Burke, 2013). This period where GUIs were integrated into Macintosh and WIndows PCs marked a transition from the traditional use of mainframe computers in education to more decentralized, personal computing environments that allowed for more interactive and practical learning experiences in programming and software development (O'Neil, 1987).

The 1990s and early 2000s represented a transformative era in software development, driven by the explosive growth of the internet and the rise of open-source software. The widespread adoption of web technologies such as HTML, JavaScript, and PHP enabled the creation of dynamic, interactive websites, leading to a proliferation of web-based applications that became integral to daily life (Lendarduzzi et al., 2020). During this period, software like web browsers, email clients, and early content management systems emerged as essential tools, reflecting the internet's increasing influence on personal and professional activities. This era also marked a significant shift in software development models with the rise of the open-source movement (Tabarés, 2021). Landmark projects such as Linux and the Apache HTTP server demonstrated the potential of decentralized, community-driven development to produce reliable and scalable software solutions. These open-source initiatives not only spurred innovation but also challenged traditional software business models by making software freely available and modifiable, fostering a culture of collaboration and shared knowledge (Bretthauer, 2001).

At the same time, the teaching of software development in this period in colleges began to adapt to the rapidly evolving technological landscape, particularly with the rise of the internet and the proliferation of new development tools. During this period, academic programs increasingly focused on bridging the gap between theoretical knowledge and practical skills, as students were introduced to software engineering principles that emphasized real-world applications. This era saw the integration of distributed computing systems and networked environments into the curriculum, reflecting the industry's shift towards interconnected software systems (Stewart, 1994). Additionally, courses began to incorporate more collaborative and project-based learning approaches, with an emphasis on teamwork and the use of modern development methodologies, such as Agile and Extreme Programming, to better prepare students for the collaborative nature of software development in the industry (Dubinsky & Hazzan, 2005). Despite these advancements, a gap persisted between academic training and industry expectations, particularly in terms of equipping students with the hands-on experience needed to navigate the complexities of professional software development environments (Craig, 2019).

The 2000s brought about further evolution in software development methodologies with the introduction of Agile practices (Argen et al., 2022). Departing from the rigid, linear waterfall model, Agile methodologies emphasized iterative development, continuous feedback, and close collaboration with customers. This approach allowed development teams to rapidly adapt to changing requirements and deliver software in small, manageable increments, thereby enhancing productivity and customer satisfaction (Ogundipe et al., 2024). Simultaneously, the emergence of DevOps represented a cultural and operational shift, integrating development and operations to streamline software deployment processes. DevOps practices, which focused on automating the entire software delivery pipeline, enabled continuous integration and continuous delivery (CI/CD) (Mishra & Otaiwi, 2020). By breaking down traditional silos between development and operations teams and promoting automation, organizations achieved more frequent and reliable software deployments (Mockus et al., 2002).

The combination of Agile, DevOps, and open-source development practices has fundamentally reshaped the field of software engineering, fostering an environment that supports rapid iteration, enhanced collaboration, and the creation of more resilient systems. They also forced the integration into software development curricula marked a significant shift in how these methodologies were taught in colleges. Agile development practices, which emphasize iterative progress, continuous feedback, and customer collaboration, became increasingly central to software engineering education. This shift was driven by the need to equip students with the skills to manage rapid software delivery cycles and adapt to evolving project requirements. DevOps, which integrates development and operations to streamline the software deployment process, was introduced alongside Agile to provide students with a comprehensive understanding of the end-to-end software development lifecycle (Jennings & Gannod, 2019). Courses began incorporating hands-on experiences with tools like Git, Jenkins, Docker, and AWS, reflecting the industry's shift towards continuous integration and delivery (Kavya & Smitha, 2022).

These educational practices aimed to bridge the gap between theoretical knowledge and the practical demands of the modern software industry, preparing students for the collaborative and fast-paced nature of contemporary software engineering environments (Betz, Olagunju, & Paulson, 2016).

## 3. Teaching Coding in the Age of AI

The integration of AI into software development is not only revolutionizing how code is written, tested, and deployed but also transforming the teaching of coding beyond traditional Computer Science departments. AI-powered tools like GitHub Copilot have gained popularity, offering automated code suggestions and autocompletion that significantly enhance productivity. These tools leverage large language models (LLMs) trained on extensive code repositories to generate relevant code snippets based on natural language inputs. Research shows that developers primarily use these tools to reduce keystrokes, complete tasks more quickly, and recall syntax, making them valuable for both novice and experienced programmers. However, challenges such as limitations in the functional accuracy of generated code and the cognitive overhead required to validate AI-generated suggestions persist (Liang et al., 2023).

AI-driven automation tools are also advancing continuous deployment practices, emphasizing rapid, small, and incremental changes through CI/CD pipelines and orchestration tools. Through automating testing, deployment, and monitoring, AI reduces the need for manual intervention and enables more frequent, reliable releases. Automated deployment pipelines integrated with AI can manage everything from code commits to production deployment, allowing for seamless updates with minimal downtime. This integration enhances agility and reduces time-to-market without compromising reliability (Sailer & Petrič, 2019). However, despite these advancements, the implementation of AI in software development presents complexities, such as compatibility and integration challenges. While AI tools like GitHub Copilot excel in generating code, significant hurdles remain in terms of usability and integration within existing workflows. Future developments are expected to focus on improving the quality of suggestions and reducing the cognitive load on developers, refining these tools to become more reliable and effective (Zhou et al., 2023).

Moreover, GAI tools are expanding their capabilities, enabling professionals outside traditional programming backgrounds to perform complex software development tasks. By automating code generation, bug detection, and deployment processes, GAI tools significantly lower the barriers to entry for those not formally trained in coding. This democratization of software development allows industries such as design, marketing, and data analysis to integrate custom software solutions tailored to their specific needs, driven by domain experts rather than coders (Bull & Kharrufa, 2023).

The implications of this trajectory extend beyond automating routine coding tasks. GAI systems are increasingly used in creative and strategic roles, allowing non-programmers to prototype applications, automate data analysis, and even develop AI models. For instance, in innovation management and digital prototyping, GAI tools enable rapid iteration of designs and generation of diverse solutions, empowering professionals without coding expertise to engage directly in technical processes. This trend suggests a future where software development becomes a collaborative, cross-disciplinary activity, supported by AI tools that handle technical complexities. Such tools not only enhance productivity but also reduce the need for specialized coding knowledge, enabling more professionals across various fields to focus on high-level problem-solving and innovation (Ebert et al., 2023).

In the context of education, this evolution will profoundly impact how coding is taught across disciplines. As AI tools make coding more accessible, educators in non-STEM fields can incorporate programming into their curricula, equipping students with essential skills that were previously confined to Computer Science departments. Leveraging AI to teach coding allows these disciplines can empower students to engage with digital tools in meaningful ways, ultimately broadening the scope and relevance of coding education across the academic spectrum.

The integration of AI-powered coding tools into various disciplines beyond traditional Computer Science departments represents a significant shift in educational practices, offering both opportunities and challenges for educators. AI tools like GitHub Copilot, which leverage large language models to generate code snippets from natural language inputs, have the potential to democratize coding by making it more accessible to students and professionals in non-STEM fields. For instance, in disciplines such as humanities, social sciences, and the arts, AI can facilitate the development of projects that previously required advanced programming skills, such as creating APIs for digital humanities research or automating data analysis for sociological studies (Zawacki-Richter et al., 2019).

Pedagogical strategies for integrating these tools must focus on building confidence and competence among educators who may not have a background in coding. Research has shown that educators in non-STEM fields often view coding as a critical skill, yet many lack the confidence to incorporate it into their teaching practices (Ray et al., 2020). To address this, professional development programs should include hands-on experiences with AI-powered coding tools, peer discussions, and reflective practices that enable educators to see the practical applications of coding in their disciplines. Such programs can help educators transition from viewing coding as an intimidating technical skill to recognizing it as a valuable tool for enhancing teaching and learning across various fields (McInnes et al., 2024).

Furthermore, the co-design of AI tools with educators can lead to the development of more tailored and effective educational resources. By involving teachers in the design process, these tools can be better aligned with the specific needs and contexts of different disciplines, ensuring that AI integration supports, rather than disrupts, existing pedagogical frameworks (Nazaretsky et al., 2021). This collaborative approach not only enhances the relevance and usability of AI tools but also empowers educators to take ownership of the technology and use it to enrich their instructional practices.

Therefore, the future of teaching software development in the age of AI lies in expanding the use of AI-powered coding tools across disciplines, supported by targeted pedagogical strategies that build educator confidence and competence. The following section will be dedicated to exploring a specific example from the Humanities, demonstrating how AI-powered coding tools can be effectively integrated into non-STEM disciplines. This case study will focus on the practical application of these tools in a literature course, where students will learn to develop an API for literary analysis.

This example will illustrate how AI can not only lower the barrier to entry for coding but also enrich the learning experience by enabling students to engage with digital tools in innovative and meaningful ways, thus expanding the traditional boundaries of humanities education.

## 4. Case Study: Integrating AI-Powered Coding in a Literature Course

### *4.1 Overview*

This case study exemplifies the practical application of AI-powered coding tools within a literature course, focusing specifically on the development of an API for Point of View (POV) analysis. By engaging with this process, educators and students in non-STEM fields can harness AI to tackle complex digital tasks, thus broadening the scope of traditional humanities education. The study illustrates a collaborative journey where an instructor, without extensive coding experience, utilizes AI as a software engineering consultant to conceptualize and develop a tool designed to analyze sophisticated literary concepts. These concepts include free indirect discourse, narrator trustworthiness, and epistemic anomalies, all captured in a structured digital format.

The development process of the POV analysis API commenced with a conceptualization phase, where the instructor worked alongside the AI to define the tool's objectives. This initial stage was followed by designing a database structure capable of capturing the nuances of literary analysis, with AI providing critical guidance on data schema creation. The process then moved to API endpoint planning, where the instructor learned about user interaction with data through accessible explanations provided by the AI. In the implementation phase, the AI generated code snippets and offered clear, jargon-free explanations, making the technical aspects of development more approachable for the humanities-focused instructor. The iterative development process, enriched by continuous AI feedback, provided a realistic simulation of software development practices, offering valuable insights into digital project management.

To integrate this API development process into the literature curriculum, the instructor crafted an assignment that walks students through each step of creating their own literary analysis API. This structured assignment not only introduces students to API concepts but also encourages the application of digital humanities methodologies, thus fostering an interdisciplinary approach to literary studies. The case study ultimately demonstrates how AI-powered coding tools can democratize access to technical skills in the humanities, equipping students with valuable competencies for an increasingly digital academic and professional landscape. By providing a model for integrating AI-assisted coding into non-STEM curricula, this case study highlights the potential for AI to bridge the gap between humanities scholarship and technical implementation, paving the way for innovation in both pedagogy and digital humanities research.

### *4.2 API Development Process*

The development of the POV analysis API **(Table 2)** began with a conceptualization phase, during which the instructor, who possessed limited coding experience, collaborated closely with an AI language model serving as a virtual software engineering consultant. The main goal of this collaboration was to create a tool capable of analyzing advanced literary concepts such as free indirect discourse, narrator trustworthiness, and epistemic anomalies in a digital format. This phase was crucial, as the instructor translated abstract literary analysis ideas into actionable digital functions that could be supported by modern programming methodologies. The AI's assistance proved instrumental in transforming these complex theoretical concepts into a practical framework for a software application, underscoring the role of AI in enabling non-technical users to contribute meaningfully to software development.

**Table 2. API Development Process**

| Phase | Description | Role of AI |
|---|---|---|
| **Conceptualization** | Collaborated with AI to conceptualize a literary analysis tool capable of analyzing complex literary concepts like free indirect discourse. | AI acted as a software engineering consultant, translating abstract literary ideas into actionable digital functions. |
| **Database Structure Design** | Designed a data schema to capture the nuances of literary analysis, including fields for POV type, narrator trustworthiness, and narration nodes. | AI provided guidance on structuring the database, helping to translate complex literary concepts into measurable data points. |
| **API Endpoint Planning** | Planned API endpoints for creating, retrieving, updating, and deleting literary analysis entries, focusing on how users would interact with the data. | AI broke down technical concepts into accessible terms, making API interactions understandable for non-technical users. |
| **Implementation** | Developed the API using modern web technologies, with AI generating code snippets and providing explanations of unfamiliar concepts. | AI generated comprehensible code and offered jargon-free explanations, enabling the instructor to actively participate in the coding process. |
| **Iterative Development** | Engaged in an iterative process of refinement, with the AI suggesting improvements and alternative approaches based on best practices. | AI continuously provided feedback and refinement suggestions, mimicking real-world software development scenarios and enhancing the instructor's understanding. |

In the subsequent database structure design phase, the challenge was to create a structured data format that could capture the subtleties of literary analysis. With AI guidance, the instructor designed a schema that included fields for essential elements such as text content, the type of POV, narrator trustworthiness, and key "narration nodes" – points in the text where shifts in narrative perspective occur. This stage exemplified how AI-powered tools can help bridge the gap between domain expertise in literature and technical implementation. AI's recommendations on structuring the database allowed the instructor to convert literary elements into measurable data points, a task that would have been significantly more difficult without technical expertise. By utilizing AI, the instructor was able to devise a schema that could efficiently store and retrieve complex literary data for analysis.

The API endpoint planning stage was pivotal in determining how users would interact with the data stored in the database. The instructor, unfamiliar with how APIs functioned, relied heavily on AI to break down the complexities of API interactions into simple, digestible explanations. The AI provided insights into how to design endpoints for creating, retrieving, updating, and deleting literary analysis entries, offering suggestions on the appropriate web technologies to use. This stage not only highlighted the AI's ability to demystify technical concepts for non-STEM educators but also showcased how accessible coding could become when guided by intuitive AI tools. The planning stage was integral to establishing a user-friendly interface, allowing literary analyses to be conducted and manipulated in a seamless digital environment.

During the implementation phase, the instructor worked alongside the AI to translate the API plans into functioning code. AI-generated code snippets provided a crucial foundation, allowing the instructor to see firsthand how abstract technical concepts came to life through programming. The AI also offered clear, jargon-free explanations of the coding language, helping to bridge gaps in the instructor's technical knowledge. This collaboration underscored the accessibility of AI-assisted development for non-technical users, allowing a humanities-focused instructor to take an active role in building sophisticated digital tools without being overwhelmed by the technicalities of coding. As the AI handled the more challenging aspects of the coding process, the instructor was able to focus on applying literary expertise to ensure the digital tool met the academic goals of the course.

Throughout the process, the instructor and AI engaged in iterative development, an ongoing cycle of refinement based on best practices in software development. The AI continuously suggested improvements and alternative approaches, incorporating insights from the instructor's literary analysis goals into the development process. This iterative refinement phase mimicked real-world software development workflows, providing a valuable learning experience for students as well. By observing how changes and adjustments were made in response to feedback, the instructor gained a deeper understanding of digital project management and agile development principles.

Incorporating these technical components into the curriculum, the case study offers a model for how AI-powered coding tools can facilitate the integration of software development skills into non-STEM disciplines. By using AI as a guide, educators in fields like literature can bring digital projects to life, enabling students to engage with coding in a way that enhances their academic experience without requiring advanced technical skills. This collaborative approach demonstrates how AI can expand the possibilities of teaching and learning, bridging the gap between the humanities and digital technology.

### *4.3 Integration of API into Literature Curriculum*

To effectively integrate the API development process into the literature curriculum, the instructor crafted a comprehensive assignment designed to guide students through the creation of their own literary analysis API **(Table 3).** This structured assignment was carefully developed to ensure that students could engage with both the technical and literary aspects of the project, thereby fostering an interdisciplinary approach to their studies. The assignment begins with an Introduction to API concepts and their relevance to literary studies. In this initial phase, students are introduced to the basics of APIs, focusing on how these tools can be used to facilitate digital literary analysis. By grounding students in the fundamental concepts of API technology, this step ensures they understand the practical applications of digital tools within the context of their literary studies.

**Table 3. Steps to Integrate AI into Curriculum**

| Assignment Step | Description |
|---|---|
| **1. Introduction to API concepts** | Introduces students to the basics of APIs and explains their relevance to literary studies, establishing a foundation for the subsequent technical work. |
| **2. Guided database design** | Guides students through the process of structuring data for literary analyses, focusing on translating literary concepts like POV into structured formats. |
| **3. Collaborative API endpoint creation using AI coding assistants** | Involves students in developing API endpoints with the help of AI tools, making the technical aspects of coding accessible and manageable. |
| **4. Implementation of POV analysis features** | Students apply their literary knowledge and coding skills to build features that analyze narrative perspectives within texts, integrating theory and practice. |
| **5. Development of a basic user interface** | Teaches students the principles of user interface design, allowing them to create a user-friendly interface for interacting with their API. |
| **6. Testing and documentation of the API** | Focuses on the importance of testing and documenting the API, ensuring functionality and providing a clear record of the development process. |
| **7. Reflection on the use of AI in the development process** | Encourages students to reflect on the role of AI in their work, considering its impact on literary analysis and future applications in the humanities. |

Following the introduction, students engage in a Guided database design session, where they learn how to structure data for storing literary analyses. This phase involves creating a schema that reflects the nuances of literary texts, such as Point of View (POV), narrator reliability, and narrative shifts. The guided design process helps students translate complex literary concepts into structured data formats, providing a solid foundation for the API they will develop. Next, students move on to Collaborative API endpoint creation using AI coding assistants. In this stage, students work with AI tools to develop the necessary API endpoints for their analysis. The AI assists in breaking down the coding process, making it accessible to students with limited technical backgrounds. This collaboration allows students to focus on the literary content while the AI handles the more challenging technical aspects, thus bridging the gap between literature and technology.

The fourth step involves the Implementation of POV analysis features. Here, students apply their knowledge of both literature and coding to build features that analyze narrative perspectives within texts. This practical application reinforces their understanding of literary theory and demonstrates how digital tools can enhance traditional literary analysis. Students then proceed to the Development of a basic user interface, which allows users to interact with the API. This step introduces them to the principles of user interface design, ensuring that the tools they create are not only functional but also user-friendly. By building a simple interface, students learn to consider the end-user experience, an important aspect of digital humanities work.

In the Testing and documentation phase, students test their APIs to ensure they function as intended and document the development process. This step emphasizes the importance of thorough testing and clear documentation in software development, skills that are transferable to many other academic and professional contexts. Finally, students conclude the assignment with a Reflection on the use of AI in the development process. This reflective component encourages them to consider the role of AI in their work, its impact on their approach to literary analysis, and the potential for future applications of digital tools in the humanities.

The assignment **(Table 4)** was found to not only equip students with practical coding skills but also encourage them to engage with digital humanities methodologies. By using AI coding assistants, students can overcome technical barriers and focus on applying their literary knowledge to create functional digital tools. This interdisciplinary approach enriches the literature curriculum, preparing students to navigate the increasingly digital landscape of academic research and professional practice.

**Table 4. Assignment: Developing a Literary Analysis API with AI Assistance**

| **Assignment: Developing a Literary Analysis API with AI Assistance** |
|---|
| **Objective:**<br>This assignment aims to guide you through the process of developing an API (Application Programming Interface) for literary analysis, specifically focusing on Point of View (POV) analysis. You will use AI-powered coding tools to create a digital tool that can analyze literary texts, bridging the gap between traditional humanities studies and modern digital methodologies. |
| **Assignment Structure:**<br>**1. Introduction to API Concepts**<br>• **Task:** Read the provided materials on API basics and watch the introductory video on how APIs function in digital humanities.<br>• **Objective:** Understand the fundamentals of APIs and how they can be applied to literary studies.<br>• **Deliverable:** A brief summary (200-300 words) of your understanding of APIs and their relevance to literary analysis.<br>**2. Guided Database Design**<br>• **Task:** Design a database schema for storing literary analyses. Your schema should include fields for text content, POV type, narrator trustworthiness, and narration nodes.<br>• **Objective:** Translate literary concepts into a structured data format that can be analyzed digitally.<br>• **Deliverable:** A diagram or description of your database schema, including explanations of each field.<br>**3. Collaborative API Endpoint Creation Using AI Coding Assistants**<br>• **Task:** Use AI coding assistants like GitHub Copilot to develop API endpoints for creating, retrieving, updating, and deleting literary analysis entries.<br>• **Objective:** Learn how to use AI tools to assist in coding and understand the basic structure of an API.<br>• **Deliverable:** The code for your API endpoints, along with comments explaining the function of each endpoint.<br>**4. Implementation of POV Analysis Features**<br>• **Task:** Implement features in your API that analyze narrative perspectives within texts, such as identifying shifts in POV or assessing narrator reliability.<br>• **Objective:** Apply your literary analysis skills in a digital context, using coding to create analytical tools.<br>• **Deliverable:** The code for your POV analysis features, accompanied by a brief explanation of how each feature works. |

**5. Development of a Basic User Interface**

- **Task:** Design a simple user interface that allows users to interact with your API, such as inputting text for analysis and viewing results.
- **Objective:** Learn the basics of user interface design and how it can enhance the usability of digital tools.
- **Deliverable:** A prototype or wireframe of your user interface, along with a description of its functionality.

**6. Testing and Documentation of the API**

- **Task:** Test your API to ensure it functions correctly and document the development process, including challenges faced and how they were overcome.
- **Objective:** Understand the importance of testing in software development and create a record of your project for future reference.
- **Deliverable:** A report (500-700 words) detailing your testing process, results, and the documentation of your API.

**7. Reflection on the Use of AI in the Development Process**

- **Task:** Reflect on your experience using AI coding assistants in this project. Consider how AI impacted your approach to literary analysis and the potential for AI in the humanities.
- **Objective:** Critically assess the role of AI in your work and its broader implications for digital humanities.
- **Deliverable:** A reflective essay (400-500 words) discussing the benefits and challenges of using AI in your project.

**Submission:**

Please compile all deliverables into a single document and submit it to the course's online portal by [due date].

**Evaluation Criteria:**

Your assignment will be evaluated based on the following criteria:

- Understanding and application of API concepts
- Quality and accuracy of the database schema
- Effective use of AI tools in coding
- Functionality and originality of the POV analysis features
- User interface design and usability
- Thoroughness of testing and clarity of documentation
- Depth of reflection on AI's role in your work

**Resources:**

- API basics guide and video
- AI coding assistant tutorials
- Sample code snippets
- Database design templates
- User interface design tools

This assignment is designed to not only teach you the basics of coding with AI assistance but also to show you how digital tools can be applied to enhance literary studies. Good luck, and enjoy the process of bridging literature and technology!

## Recommendations

The case study highlights the transformative potential of AI-powered coding tools in democratizing technical skill development within the humanities. Based on the experiences and insights gained from integrating AI into a literature course, several key recommendations emerge for educators and institutions aiming to introduce similar initiatives. One of the critical lessons from this case study is the importance of iteration. The first two versions of the API development process faced significant challenges and ultimately failed to meet the desired outcomes. It was only through continuous refinement and the incorporation of feedback that the third version succeeded. Educators should adopt an iterative approach when integrating AI and coding into their curricula, allowing for trial and error, and encouraging students to see failure as a step towards improvement. This mindset not only mirrors real-world software development practices but also fosters resilience and adaptability among students.

Before diving into the technical aspects of coding, it is crucial to establish a solid conceptual foundation. In the case study, the initial phase of conceptualizing the literary analysis tool provided a clear framework that guided the subsequent technical work. Educators should ensure that students have a strong understanding of the

theoretical underpinnings of their projects, which will help them make informed decisions during the development process. AI-powered tools played a crucial role in bridging the gap between the instructor's literary expertise and the technical requirements of software development. Institutions should explore AI tools that can demystify complex coding concepts and make technical skills accessible to non-STEM students and educators. By leveraging AI as a collaborative partner in the development process, humanities scholars can engage with digital tools without being hindered by a lack of programming experience.

The integration of AI-assisted coding in literature courses not only enhances digital literacy but also opens new avenues for interdisciplinary collaboration. Educators should encourage students to collaborate across disciplines, combining their humanities expertise with technical skills to create innovative digital tools. This approach can lead to the development of unique projects that enrich both the humanities and the computational fields. To successfully implement AI-powered coding tools in non-STEM disciplines, educators must be equipped with the necessary skills and confidence. Institutions should invest in professional development programs that provide hands-on training with AI tools and offer continuous support as educators integrate these technologies into their teaching practices. This investment will ensure that instructors can effectively guide students through the technical aspects of their projects.

Continuous evaluation and iteration of the curriculum are essential to the successful integration of AI-assisted coding. Educators should regularly assess the effectiveness of their teaching methods, gather feedback from students, and make adjustments as needed. This iterative approach to curriculum design ensures that the educational experience remains relevant, engaging, and aligned with the evolving digital landscape. Finally, institutions should strive to make digital humanities resources, including AI-powered tools, more accessible to all students. This may involve providing access to software, offering workshops on digital literacy, and creating online repositories of tutorials and best practices. By expanding access, institutions can ensure that more students can benefit from the opportunities presented by AI and coding in the humanities.

The integration of AI-powered coding tools into humanities education represents a significant step towards bridging the gap between literary scholarship and technical implementation. By embracing iterative development, fostering interdisciplinary collaboration, and providing robust support for educators and students, institutions can create a learning environment that prepares students for the increasingly digital academic and professional landscape. This case study offers a model for how non-STEM disciplines can effectively leverage AI to enhance digital literacy, foster innovation, and open new avenues for computational analysis in literary studies.

## Conclusion

This article has explored the transformative potential of AI-powered coding tools in reshaping the landscape of software development education, particularly beyond the confines of traditional Computer Science departments. As highlighted, the integration of AI into software development practices—through tools like GitHub Copilot and other generative AI technologies—has opened up new possibilities for educators and students across various disciplines. By lowering the technical barriers to entry, these tools make coding more accessible to individuals without formal programming backgrounds, thereby democratizing software development and fostering interdisciplinary innovation.

The need to incorporate these AI-driven tools into non-STEM disciplines, such as the humanities, is particularly pressing. As the case study on integrating AI-powered coding into a literature course demonstrated, these tools can empower students to engage with complex digital tasks, such as developing APIs for literary analysis, that would otherwise be beyond their reach. This not only enhances the educational experience by providing students with practical, technical skills but also broadens the scope of what is possible within humanities education, encouraging a more interdisciplinary approach to learning.

Moving forward, the case study serves as a model for how AI-powered coding tools can be used effectively in non-STEM disciplines, offering a blueprint for educators to adapt and apply in their contexts. However, further research is necessary to explore the long-term impacts of these tools on student learning outcomes, the development of interdisciplinary curricula, and the potential challenges that may arise in their implementation. Future studies should focus on evaluating the effectiveness of AI tools in various educational settings, understanding how they can be optimized for different learning environments, and exploring the ethical considerations surrounding their use in education. By continuing to investigate and refine the use of AI in teaching, educators can ensure that these tools contribute meaningfully to the evolution of educational practices and the preparation of students for a rapidly changing technological landscape.

**Data Availability**

Data available upon request.

**Conflicts of Interest**

The authors declare that there is no conflict of interest regarding the publication of this paper.

**Funding Statement**

NA

**Authors' Contributions**

Conceptualization, D. Plate; Plate, D; Validation, J. Hutson; Investigation, Plate, D. – Original Draft Preparation, J. Hutson; Writing – Review & Editing, J. Hutson.; Visualization, J. Hutson.

## References

1. Bahroun, Z., Anane, C., Ahmed, V., & Zacca, A. (2023). Transforming education: A comprehensive review of generative artificial intelligence in educational settings through bibliometric and content analysis. *Sustainability*, *15*(17), 12983.
2. Betz, C., Olagunju, A. O., & Paulson, P. (2016, September). The impacts of digital transformation, agile, and DevOps on future IT curricula. In *Proceedings of the 17th Annual Conference on Information Technology Education* (pp. 106-106).
3. Brynjolfsson, E., Li, D., & Raymond, L. R. (2023). Generative AI at work (No. w31161). *National Bureau of Economic Research.*

4. Bull, C., & Kharrufa, A. (2023). Generative AI Assistants in Software Development Education. *ArXiv, abs/2303.13936*. https://doi.org/10.1109/MS.2023.3300574
5. Castro, S. (2015). *Meeting the 'T'in STEM through Computer Science and Coding*. PhD diss., California State University.
6. Collen, M. F., & Kulikowski, C. A. (2015). The development of digital computers. *The History of Medical Informatics in the United States*, 3-73.
7. Craig, R. (2019). America's Skills Gap: Why It's Real, and Why It Matters. *Progressive Policy Institute*.
8. Dijkstra, E. W. (1996). Edsger W. Dijkstra. *Great Papers in Computer Science*, 378.
9. Drucker, J. (2021). *The digital humanities coursebook: an introduction to digital methods for research and scholarship*. Routledge.
10. Dubinsky, Y., & Hazzan, O. (2005). A framework for teaching software development methods. *Computer Science Education*, *15*(4), 275-296.
11. Ebert, C., Louridas, P., & Ebert, C. (2023). Generative AI for Software Practitioners. *IEEE Software*, *40*, 30-38. https://doi.org/10.1109/MS.2023.3265877
12. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In ECOOP'93—Object-Oriented Programming: *7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7* (pp. 406-431). Springer Berlin Heidelberg.
13. Gutiérrez, L. E., Guerrero, C. A., & López-Ospina, H. A. (2022). Ranking of problems and solutions in the teaching and learning of object-oriented programming. *Education and Information Technologies*, *27*(5), 7205-7239.
14. Grütter, F. (2024). After Mainframes: Computer Education and Microcomputers in Western Switzerland during the 1980s and 1990s. *History of Education*, 1-21.
15. Haigh, T., & Ceruzzi, P. E. (2021). *A new history of modern computing*. MIT Press.
16. Hamilton, I. & Swanston, B. (2024). Artificial intelligence in education: teachers' opinions on AI in the classroom. *Forbes Advisor*. Retrieved: https://www.forbes.com/advisor/education/it-and-tech/artificial-intelligence-in-school/
17. Hazzan, O., & Erez, Y. (2024, March). Generative AI in Computer Science Education. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2* (pp. 1899-1899).
18. Holo, O. E., Kveim, E. N., Lysne, M. S., Taraldsen, L. H., & Haara, F. O. (2023). A review of research on teaching of computer programming in primary school mathematics: moving towards sustainable classroom action. *Education Inquiry*, *14*(4), 513-528.
19. Hordern, J. (2018). Recontextualisation and the education-work relation. In *Knowledge, Curriculum, and Preparation for Work* (pp. 68-88). Brill.
20. Hunter, K. (1988, October). The campus microcenter. In *Proceedings of the 16th annual ACM SIGUCCS Conference on User Services* (pp. 225-228).
21. Hutson, J., & Jeevanjee, T. (2024). Perceptions and Aspirations of Undergraduate Computer Science Students Towards Generative AI: A Qualitative Inquiry. *Journal of Biosensors and Bioelectronics Research*, *2*(3).
22. Jablonický, K., & Lang, J. (2023). Code Based Selected Object-Oriented Mechanisms Identification. *Proceedings http://ceur-ws. org ISSN*, *1613*, *0073*
23. Jennings, R. A. K., & Gannod, G. (2019, October). Devops-preparing students for professional practice. In *2019 IEEE Frontiers in Education Conference (FIE)* (pp. 1-5). IEEE.
24. Kasauli, R., Knauss, E., Horkoff, J., Liebel, G., & de Oliveira Neto, F. G. (2021). Requirements engineering challenges and practices in large-scale agile system development. *Journal of Systems and Software*, *172*, 110851.
25. Kavya, N., & Smitha, P. (2022). Deploying and Setting up Ci/Cd Pipeline for Web Development Project on AWS Using Jenkins. *International Journal of Advanced Engineering Management*, *4*(6), 2325-2332.
26. Krismadinata, E., Boudia, C., Jama, J., & Saputra, A. Y. (2023). Effect of Collaborative Programming on Students Achievement Learning Object-Oriented Programming Course. *International Journal of Information and Education Technology*, *13*(5).
27. Li, H., Huang, B., & Lu, J. (2008, June). Dynamical evolution analysis of the object-oriented software systems. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)* (pp. 3030-3035). IEEE.
28. Liang, J., Yang, C., & Myers, B. (2023). Understanding the Usability of AI Programming Assistants. ArXiv, abs/2303.17125. https://doi.org/10.48550/arXiv.2303.17125
29. Liberman, N., Beeri, C., & Ben-David Kolikant, Y. (2011). Difficulties in learning inheritance and polymorphism. *ACM Transactions on Computing Education (TOCE)*, *11*(1), 1-23.
30. Magana, S. (2017). *Disruptive classroom technologies: A framework for innovation in education*. Corwin Press.
31. McInnes, L. C., Heroux, M., Bernholdt, D. E., Dubey, A., Gonsiorowski, E., Gupta, R., ... & Watson, G. R. (2024). A cast of thousands: How the IDEAS Productivity project has advanced software productivity and sustainability. *Computing in Science & Engineering*.
32. Megargel, A., Shankararaman, V., & Walker, D. K. (2020). Migrating from monoliths to cloud-based microservices: A banking industry example. *Software engineering in the era of cloud computing*, 85-108.

33. Melro, A., Tarling, G., Fujita, T., & Kleine Staarman, J. (2023). What else can be learned when coding? A configurative literature review of learning opportunities through computational thinking. *Journal of Educational Computing Research*, *61*(4), 901-924.
34. Mens, T. (2014). *Evolving Software Systems* (Vol. 190). A. Serebrenik, & A. Cleve (Eds.). Heidelberg: Springer.
35. Nagineni, R. B. (2021). A Research on Object Oriented Programming and Its Concepts. *International Journal*, *10*(2).
36. Nazaretsky, T., Cukurova, M., Ariely, M., & Alexandron, G. (2021, September). Confirmation bias and trust: human factors that influence teachers' attitudes towards AI-based educational technology. In *CEUR Workshop Proceedings* (Vol. 3042).
37. Ng, D. T. K., Lee, M., Tan, R. J. Y., Hu, X., Downie, J. S., & Chu, S. K. W. (2023). A review of AI teaching and learning from 2000 to 2020. *Education and Information Technologies*, *28*(7), 8445-8501.
38. Rajlich, V. (1997). MSE: A methodology for software evolution. *Journal of Software Maintenance: Research and Practice*, *9*(2), 103-124.
39. Ray, B. B., Rogers, R. R., & Hocutt, M. M. (2020). Perceptions of non-STEM discipline teachers on coding as a teaching and learning tool: What are the possibilities?. *Journal of Digital Learning in Teacher Education*, *36*(1), 19-31.
40. Ruiz-Rojas, L. I., Acosta-Vargas, P., De-Moreta-Llovet, J., & Gonzalez-Rodriguez, M. (2023). Empowering education with generative artificial intelligence tools: Approach with an instructional design matrix. *Sustainability*, *15*(15), 11524.
41. Saide, M. (2024). Understanding Object-Oriented Development: Concepts, Benefits, and Inheritance in Modern Software Engineering. *Benefits, and Inheritance in Modern Software Engineering* (July 01, 2024).
42. Sailer, A., & Petrić, M. (2019). Automation and Testing for Simplified Software Deployment. *EPJ Web of Conferences*. https://doi.org/10.1051/EPJCONF/201921405019
43. Santhosh, A., Unnikrishnan, r., Shibu, S., Meenakshi, K., & Joseph, G. (2023). AI impact on job autonation. *International Journal of Engineering Technology and Management Sciences*. https://doi.org/10.46647/ijetms.2023.v07i04.05
44. Stewart, C. (1994). Distributed systems in the undergraduate curriculum. *ACM SIGCSE Bulletin*, *26*(4), 17-20.
45. Tseng, W., & Warschauer, M. (2023). AI-writing tools in education: If you can't beat them, join them. *Journal of China Computer-Assisted Language Learning*, *3*(2), 258-262.
46. Weng, X., & Chiu, T. K. (2023). Instructional design and learning outcomes of intelligent computer assisted language learning: Systematic review in the field. *Computers and Education: Artificial Intelligence*, *4*, 100117.
47. Williams, R. (2013). *Programming a New Society: Modularity as an Instrument of Cooperation and Programmer Autonomy from the 1960s to the Free Software Movement* (Doctoral dissertation, Vanderbilt University. Dept. of History).
48. Yuan, T., Wang, Z., & Rau, P. L. P. (2023, July). Design of Intelligent Real-Time Feedback System in Online Classroom. In *International Conference on Human-Computer Interaction* (pp. 326-335). Cham: Springer Nature Switzerland.
49. Zawacki-Richter, O., Marín, V. I., Bond, M., & Gouverneur, F. (2019). Systematic review of research on artificial intelligence applications in higher education–where are the educators?. *International Journal of Educational Technology in Higher Education*, *16*(1), 1-27.
50. Zhou, X., Liang, P., Zhang, B., Li, Z., Ahmad, A., Shahin, M., & Waseem, M. (2023). On the concerns of developers when using GitHub Copilot. *arXiv preprint arXiv:2311.01020*
51. Zohuri, B. (2023). Charting the future. The synergy of generative AI, quantum computing, and the transformative impact on economy. *Current Trends in Engineering Science*, *3*(7), 1-4.