software

MDPI

Article

Learning to Code with Context: A Study-Based Approach †

Uwe M. Borghoff *, Mark Minas and Jannis Schopp

Institute for Software Technology, Department of Computer Science, University of the Bundeswehr Munich, Werner-Heisenberg-Weg 39, 85579 Neubiberg, Germany; mark.minas@unibw.de (M.M.); jannis.schopp@unibw.de (J.S.)

* Correspondence: uwe.borghoff@unibw.de

† This paper is an extended version of our paper published in Borghoff, U.M.; Minas, M.; Schopp, J. Generative AI in Student Software Development Projects: A User Study on Experiences and Self-Assessment. In Proceedings of the 6th ACM European Conference on Software Engineering Education (ECSEE 2025), Seeon/Bavaria, Germany, 2–4 June 2025.

Abstract

The rapid emergence of generative AI tools is transforming software development. Consequently, software engineering education must adapt to ensure that students not only learn traditional development methods but also understand how to use these new technologies effectively and responsibly. In particular, project-based courses provide an effective setting in which to explore and evaluate the integration of AI assistance into real-world development practices. This paper presents our approach and a user study conducted in the context of a university programming project in which students collaboratively developed computer games. The study investigates how participants used generative AI tools across different phases of the software development process, identifies the tasks for which these tools were perceived as most useful, and analyzes the challenges students encountered. Building on these insights, we further examine a repository-aware, locally deployed large language model (LLM) assistant designed to provide project-contextualized support. The system employs retrieval-augmented generation (RAG) to ground its responses in relevant documentation and source code, thereby enabling a qualitative analysis of model behavior, parameter sensitivity, and common failure modes. These findings deepen our understanding of context-aware AI support in educational software projects and inform the future integration of AI-based assistance into software engineering curricula.

Keywords: software development project course; software engineering education; generative AI; repository-aware LLM; retrieval-augmented generation; qualitative analysis

Academic Editors: Hongyu Zhang and Mirko Viroli

Received: 16 May 2026
Revised: 14 June 2026
Accepted: 18 June 2026
Published: 21 June 2026

1. Introduction

The programming project at our university is a software development course in which teamwork is a central component. Similar courses are likely part of the curricula of most bachelor's degree programs in computer science worldwide.

In this course, teams of approximately seven students collaborate to develop a computer game following a structured software development process. The main objective is to enable participants to gain practical experience in team-based software development, including core activities such as planning, programming, testing, and documentation, while also strengthening essential soft skills such as teamwork and communication.

The widespread availability of generative AI tools, such as ChatGPT and GitHub's Copilot [1], is also affecting software development practices. However, these tools have

not yet been formally integrated into the curriculum of our programming project. In informal conversations, students frequently report using them, although they are often dissatisfied with the quality of the responses. To address this largely unguided use of AI and to promote its responsible application, we plan to provide participants in the programming project with a chat-based AI tutor in the coming years. This tutor will support students in software development, particularly in programming-related tasks. To mitigate the widely discussed effect of *skill degradation* [2,3], it will not simply provide complete solutions; instead, like a human tutor, it will offer hints and guide students toward solving problems independently.

In recent months, we have pursued an empirical approach to inform the implementation and deployment of the AI tutor. This paper describes our multi-stage approach and the results obtained at each stage. In this approach, we distinguish between the freely available AI tools examined in the user study, the repository-aware LLM assistant evaluated in the case study, and the planned AI tutor that motivates the long-term development of tutoring-oriented support.

In the *first step*, we empirically investigated how our students use freely available AI tools. To this end, we conducted a user study among participants in the previous year's programming project, which took place from October to December 2024. From the beginning of the course, students were permitted to use freely available AI tools as optional aids. Their use was neither required nor formally integrated into the course activities, although individual examples of appropriate use, such as documentation support, were occasionally discussed when students requested guidance. At the end of the programming project, we conducted a study in the form of a structured questionnaire that captured their experiences with AI tools across the different phases of the project and in relation to the various activities involved.

The user study revealed several limitations of freely available AI tools in this application domain. In particular, these tools lack access to the documentation and source code of the sample project provided to participants, although students must become familiar with these materials during the course. Consequently, when students asked questions in this context, the tools were rarely able to generate helpful responses. To overcome this limitation, our planned AI tutor must have access to relevant project documents and source code and must be able to incorporate this information appropriately into its responses.

In the *second step* of our empirical approach, we conducted a case study in which we implemented a repository-aware, locally deployed LLM assistant designed to provide project-contextualized support. The system employs RAG to ground its responses in relevant documentation and source code. This assistant enabled us to investigate model behavior, parameter sensitivity, and common failure modes. To assess the correctness of generated solutions and analyze model behavior under controlled conditions, the assistant evaluated in this case study deliberately produced ready-made solutions. Unlike the planned AI tutor, it was not intended to provide pedagogically guided assistance.

A further important aspect of these investigations was the requirement that our planned AI tutor be operated locally rather than relying on commercial API services as the primary model-serving layer. This requirement was motivated by data protection considerations and by the need to retain control over the data exchanged during repository-aware interaction, including prompts, project documents, retrieved passages, source-code snippets, tool outputs, and interaction logs. To ensure sufficiently fast response times, even when multiple students use the tutor simultaneously, the underlying LLM should be as small as possible. The case study therefore sought to determine which LLMs are capable of answering the selected programming queries correctly and to what extent smaller models with fewer parameters can do so.

https://doi.org/10.3390/software5020027

1.1. Contributions

This paper makes the following contributions:

- We provide empirical evidence on how students use and experience generative AI in a realistic software engineering project course, based on a structured user study across different development phases.
- We design and systematically evaluate a repository-aware, locally deployed LLM assistant based on RAG, and analyze how context integration affects model behavior and response quality in relation to the limitations observed in the user study. The resulting combination of user-centered and system-level analysis offers a perspective that is less commonly explored in existing RAG-based work.

1.2. Organization

The rest of the paper is organized as follows. Following the review of related work in Section 2, we introduce the context of the study, focusing on our programming project course in Section 3. Sections 4 and 5 present the user study, which analyzes how students use and experience generative AI tools across different phases of the software development process and identifies key limitations of current approaches. Based on these findings, Sections 6–8 describe the design and systematic evaluation of a repository-aware, locally deployed LLM assistant, focusing on how contextual grounding affects model behavior, response quality, and typical failure modes. We conclude in Section 9.

2. Related Work

Recent research highlights the growing role of artificial intelligence (AI) in both software engineering practice and education. Rane et al. [4] investigate how AI technologies, including machine learning and natural language processing, enable adaptive systems that tailor instruction based on learner feedback and progress. Nitzl et al. [5] likewise show that AI improves performance in complex analytical tasks, such as intelligence analysis, by supporting data interpretation and decision-making.

In software engineering, researchers widely investigate conversational and generative AI models [6,7]. Surveys and Special Issues provide comprehensive overviews of this rapidly evolving field [8,9]. Prior work consistently finds that AI and deep learning enhance efficiency, accuracy, and productivity in development processes [10–14]. At the same time, researchers emphasize that these advances reshape required competencies and demand new forms of human–AI collaboration [15–17].

Within education, AI-based tools support learning, tutoring, and assessment. For example, ChatGPT generates natural-language explanations and provides personalized tutoring for software engineering students [18]. Automated systems also assist with program evaluation, grading, and code generation, thereby improving feedback quality and reducing instructor workload [19]. More broadly, prior studies show that AI supports novice programmers by offering guidance, explanations, and code suggestions [20].

Automatic assessment systems, or *autograders* [21], have long played an important role in programming education. Test-based systems execute student submissions against instructor-defined test cases, a practice that dates back to the 1960s [22]. Contemporary examples include ArTEMiS (https://github.com/ls1intum/ArTEMiS, accessed 11 March 2026) [23], which provides an online coding environment with immediate feedback, and Praktomat (https://www.waxmann.com/automatisiertebewertung/, accessed 11 March 2026), which supports Java- and JUnit-based testing while integrating plagiarism detection via JPlag (https://github.com/jplag/JPlag, accessed 11 March 2026) and style checking through Checkstyle (https://checkstyle.org/, accessed 11 March 2026).

More recent systems employ machine learning for automatic grading [24] and adaptive evaluation [25,26].

Advances in deep learning and large language models (LLMs) further transform software development. Models such as DeepMind's AlphaCode [27], Google's PaLM [28], Microsoft's CodeBERT [29], and OpenAI's ChatGPT and Codex [30] generate syntactically correct and functionally meaningful code from natural-language prompts. These capabilities drive the emergence of AI-based development assistants that support code generation, explanation, refactoring, and testing, thereby improving accessibility and productivity for developers [20].

Table 1 summarizes representative systems used in educational and professional contexts.

Table 1. Representative AI-based systems for code generation and assistance.

System	Key Features
AlphaCode	DeepMind's competitive-programming model generating solutions from natural-language problem statements (https://deepmind.google/blog/competitive-programming-with-alphacode/, accessed 20 June 2026).
PaLM	Google's LLM, capable of reasoning and text-to-code generation across domains (https://ai.google/discover/palm2/, accessed 9 March 2026).
CodeBERT	Transformer pretrained on paired natural language and source code for code understanding and synthesis (https://github.com/microsoft/CodeBERT, accessed 9 March 2026).
ChatGPT/Codex	OpenAI's conversational and code-oriented LLMs for generating, explaining, and refactoring code (https://openai.com/de-DE/index/introducing-codex/, accessed 9 March 2026).
CodeWhisperer	Amazon's AI assistant, providing full-function suggestions and built-in security checks in real time (https://workshops.aws/categories/CodeWhisperer, accessed 9 March 2026).
GhostWriter	Replit's in-editor assistant, offering code completion, transformation and inline explanations (https://blog.replit.com/ai, accessed 9 March 2026).
Codota/Tabnine	Predictive code-completion tools trained on open-source corpora (https://www.tabnine.com/blog/codota-is-now-tabnine/, accessed 9 March 2026).
Cody	Sourcegraph's context-aware assistant, supporting repository navigation and refactoring (https://about.sourcegraph.com/cody, accessed 9 March 2026).
Diffblue Cover	AI-driven test-generation tool for Java code, automating unit test creation and improving coverage (https://www.diffblue.com/, accessed 9 March 2026).
Qodo (aka CodiumAI)	AI platform focused on code generation and quality assurance, with context-aware review and test scaffolding features (https://qodo.ai/, accessed 9 March 2026).
Cursor AI	Coding assistant indexing code bases and enabling natural-language prompts for code generation and modification (https://cursor.com/, accessed 9 March 2026).

Gao et al. [31] compare retrieval-augmented generation (RAG) with fine-tuning and prompt engineering and show that RAG more effectively incorporates external knowledge and adapts to new contexts. Similarly, Ovadia et al. [32] find that combining retrieved information with model generation improves performance over unsupervised fine-tuning. Recent surveys further emphasize the importance of contextual awareness in LLM-based

systems [33,34]. These insights motivate repository-based approaches that integrate project-specific knowledge into AI-assisted workflows.

Beyond code generation, researchers examine how LLMs influence programming practice and education. Manakina and Lung [35] show that LLMs support curriculum design and provide structured guidance for assignments. Raman and Kumar [36] analyze the coding process using a six-step framework and evaluate automation tools such as GitHub's COPILOT [1], concluding that software engineering education should increasingly emphasize code comprehension [37]. Moroz et al. [38] further observe that developers often use these tools to explore multiple solution strategies rather than to obtain a single correct answer.

Despite these benefits, researchers also identify important limitations. Choudhuri et al. [39] report no significant improvements in productivity or self-efficacy when using CHATGPT and highlight issues such as incomplete guidance and fabricated responses. Pudari and Ernst [40] note that, while AI tools handle syntax-level tasks effectively, they struggle with higher-level design and idiomatic reasoning. AlOmar [41] shows that students remain cautious about relying on CHATGPT for tasks such as code review. Finally, Li et al. [42] and Garousi et al. [43] warn that excessive reliance on generative AI may hinder long-term learning, reduce critical thinking, and lead to skill degradation.

Overall, prior work establishes that AI significantly enhances software engineering practice and education, while also introducing challenges related to trust, learning outcomes, and human–AI collaboration. However, existing work does not sufficiently combine (i) empirical evidence on how students actually use and experience generative AI in realistic project settings with (ii) a concrete, repository-aware LLM architecture that is evaluated under controlled conditions with respect to context integration, model behavior, and configuration effects. In particular, the interplay between user behavior, missing project context, and the performance of locally deployed, RAG-based assistants remains underexplored. This paper addresses this gap by linking a user study of AI usage in a software engineering project course with the design and systematic evaluation of a repository-aware, locally deployed LLM assistant.

3. The Programming Project

For most of our students, this programming project was their first software development project. In this course, they work in teams of approximately seven students to develop a complete computer game in Java. The course carries nine ECTS credits, corresponding to a workload of approximately 270 h during the period from October to December.

We have offered this course for many years and have continuously refined it. The following sections describe the course structure used in the 2024 and 2025 academic years.

3.1. Basic Organization

Each team is assigned a tutor, a supervisor, and a one-page sketch of the game they are expected to develop. The tutors are students who completed the course in the previous year, while the supervisors are members of the academic staff. The assigned game is a multiplayer 3D game with network support, based on a well-known board or card game (e.g., Monopoly, Ludo, or Risk). Games are assigned according to the teams' preferences, and no game is assigned to more than one team.

The course begins with a *kick-off meeting*, during which all students, supervisors, and tutors gather in the lecture hall to discuss organizational matters. In the following weeks, each team works independently on its project, following the traditional waterfall process model described below.

This process model is used for both didactic and organizational reasons. First, it is simple and clearly structured, which is particularly important because, for most students, this is their first experience working on a software development project as part of a team. Consequently, they are generally unfamiliar with the various phases and artifacts involved in software development. Second, the course lasts only 11 weeks, leaving little time to learn and apply a more modern agile process model. For the purposes of the present study, this structure also offers an analytical advantage: because the project separates *Preparation*, *Analysis*, *Design*, and *Coding & Testing* into distinct phases, AI usage can be reported and interpreted separately for each phase of the development process.

At the end of the term, the teams present their games to the other teams and invited guests, such as fellow students, during a *public game presentation*. This event also includes a competition in which all participants, including guests, have the opportunity to evaluate each game, and the winning team receives a prize.

The project schedule is organized into four sequential phases, as shown in Figure 1 and described below.

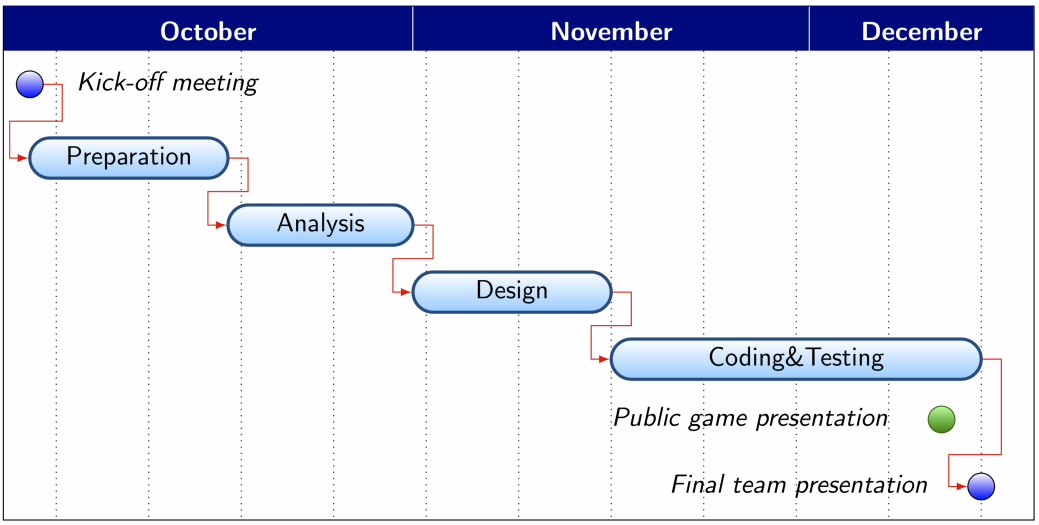

Figure 1. Schedule of the programming project.

Each team meets weekly for one hour with its supervisor and tutor. During these meetings, team members present their progress and discuss problems and possible solutions. Each phase concludes with a milestone, and teams may proceed to the next phase only after receiving approval from their supervisor. As a result, the duration of individual phases may exceed the planned schedule, causing teams to fall behind. In the previous year, this occurred for several teams, particularly during the *Design* phase. Nevertheless, all teams were able to catch up by the end of the term and participate in the *public game presentation* with an executable game.

3.2. Project Phases

The project consists of the following phases. The first phase is the *Preparation* phase (2 weeks), during which each team member completes training tasks and practices relevant programming techniques. This phase is independent of the specific game that will be developed later. Instead, an implementation of the simple game *Battleship* is provided to all teams from the outset. Like the games developed during the project, it is a multiplayer 3D game with network support and is implemented using the freely available 3D game engine JMONKEYENGINE. *Battleship* was designed to illustrate the architecture and core concepts of 3D games and also serves as a starting point and blueprint for the teams' own

developments. The goal of the *Preparation* phase is therefore to familiarize students with *Battleship*, its architecture, and its extension.

The following three phases follow the waterfall process model. During the *Analysis* phase (2 weeks), teams create a product requirements specification based on the one-page sketch of their assigned game. They produce artifacts such as an extended textual game description, a data model represented as a class diagram, a list of use cases and acceptance tests, and an initial version of the user manual.

In the subsequent *Design* phase (2 weeks), the teams create a software specification document. This document contains a detailed description of the game specification, including the chosen software architecture and the communication protocol required for network interaction. Recommended diagrams include class diagrams, BPMN diagrams, sequence diagrams, and state diagrams. In addition, the system, integration, and unit tests are specified in a test-case table. During the *Coding & Testing* phase (approximately 3 weeks), the games are implemented and tested in accordance with the software specification documents. Any deviations from these documents must be documented, and the final version of the user manual is completed and delivered.

A team is considered to have successfully completed the course once its supervisor has approved the final product and the team has presented its project and game during a *final team presentation*. After these requirements have been fulfilled, all team members receive the course credits.

3.3. Course Evolution

Our programming project has evolved over the years, as described by Borghoff et al. [44], although its basic structure has remained unchanged: teams develop games in Java, follow the waterfall process model, and present their results during the *public game presentation*. The latter has proven to be an essential component of the course for the following reason. Students receive credit for successful participation, but they do not receive a grade because individual performance is difficult to assess within a team setting. In the absence of the incentive provided by a higher grade, students may be less motivated to contribute beyond the minimum required effort. The public presentation of each team's game, and in particular, the competition for the prize awarded to the most appealing and entertaining game help counteract this tendency.

This is reflected in the fact that, in every iteration of the programming project, almost all teams completed their games in time for the public presentation, even when they had fallen behind schedule because one or more phases took longer than expected. Furthermore, many students reported that presenting their game publicly was one of the highlights of the course and an especially enjoyable experience.

Recently, students have increasingly incorporated generative AI tools into their workflows. Although these tools have not been formally integrated into the course curriculum, their use has been permitted for specific tasks, such as code documentation and text proofreading, in both the 2024 and 2025 iterations of the course.

4. User Study

At the end of the 2024 term, a user study was conducted during the final presentations to investigate the extent and nature of AI tool usage. Understanding how students adopt and apply general-purpose AI tools, primarily LLMs, during the programming project can provide valuable insights for the development of dedicated AI tools to better support our software engineering course. However, because these tools are neither formally integrated into the course curriculum nor openly discussed by students, further investigation is needed to assess their usage and impact.

Rather than advocating a specific approach to AI integration, we examine how students choose to incorporate AI tools into a software development project when their use is permitted but not formally prescribed. Our aim is to establish a baseline understanding of AI adoption in order to determine what forms of support, tutoring, or other AI-driven interventions may be needed in the future.

A recent experiment by Rapaka et al. [45] examines two versions of an educational game. One version followed a traditional design, whereas the other incorporated an intelligent, AI-based process. The experiment demonstrates that immersive and AI technologies can serve as valuable tools in the development of educational games and entertainment applications. At the same time, previous research by Choudhuri et al. [39] highlights that the use of AI in learning software engineering can significantly increase frustration levels.

To investigate these questions, we conducted a user study that captures a snapshot of AI adoption among the student body. The goal of the study was to determine which aspects of the course students engaged with using AI tools, to identify successful applications, and to uncover remaining challenges.

4.1. Demographics

The study included 38 participants, representing approximately 78% of the 49 students enrolled in the course. As shown in Table 2, most participants were Computer Science students, followed by students of Business Informatics and Mathematical Engineering. The gender distribution and degree-program composition of the study participants closely mirrored those of the overall course population.

The participant population was predominantly male, reflecting the composition of the course itself. Therefore, the study does not provide sufficient data to analyze potential gender-related differences in AI adoption, usage patterns, or perceptions. No conclusions should be drawn in this regard.

Table 2. Demographics of course and study participants.

Characteristic	Course Participants ($n = 49$)	Study Participants ($n = 38$)
Gender		
Male	46	35
Female	3	3
Academic programs		
Computer Science	34	26
Business Informatics	13	10
Mathematical Engineering	2	2
Participation rate	–	≈78%

Of the 49 students enrolled in the course, 43 were in the middle stage of their bachelor's program, while 6 were approaching its completion. At the time of the study, all participants had completed the required undergraduate courses in computer science and object-oriented programming.

4.2. Methodology

Data for the study were collected using a structured questionnaire distributed during the final team presentation of the programming project. This timing allowed participants to reflect on their experiences across all phases of the project.

Participation in the study was voluntary, and students were informed of their right to withdraw at any time. Data were collected anonymously, and participants were given the

opportunity to request the deletion of their data. The study complied with general data protection regulations to ensure the ethical handling and protection of personal data.

The questionnaire was designed to provide a comprehensive evaluation of the adoption and use of AI tools during the programming project. Its design was informed by the *Technology Acceptance Model* (TAM) [46], particularly the constructs of perceived usefulness, ease of use, and behavioral intention. However, the questionnaire was not intended as a validated TAM instrument and does not implement the standard TAM scale structure.

4.3. Questionnaire

We invited all students attending the final presentations to participate in the study, regardless of whether they had used AI tools. Including students who had not used AI was essential, as their perspectives provide valuable context for developing an authentic picture of AI adoption. Those who had not used AI were asked to explain their reasons for not doing so, thereby providing insight into barriers to adoption and other influencing factors.

1. *General question:*
   "Did you use AI tools during your programming project?"
   Options: Yes, No. If "No", participants provided reasons such as lack of knowledge about available tools, no access to suitable tools, privacy or trust concerns, and other (open-ended response).

Perceived usefulness was incorporated into the study by asking students to assess how AI tools supported their work throughout the different phases of the programming project, namely *Preparation*, *Analysis*, *Design*, and *Coding & Testing*. The questionnaire captured several dimensions relevant to understanding the adoption and use of generative AI tools. Participants were asked to describe the tasks they performed with AI support and to identify the platforms they used. Particular attention was given to students' perceptions of AI support for debugging, documentation, and code generation, in order to evaluate the extent to which these tools were perceived to support task completion and efficiency in specific project contexts.

2. *Repeated questions for each project phase:*
   (a) *"Did you use AI tools during this phase?"*
   Options: Yes, No.
   (b) *"What specific AI platforms did you use during this phase?"*
   Examples: ChatGPT, Copilot, Claude, Gemini, LLaMA, or other (open-ended response).
   (c) *"What tasks have you worked on using AI tools?"*
   Participants could choose from predefined tasks or provide their own description in an open-ended response.
   (d) *"How helpful was AI in this phase?"*
   Scale: 1 (not helpful) to 5 (very helpful).
   (e) *"How much time did the AI save you during this phase?"*
   Options: None; Less than 1 h, 1–3 h, 4–6 h, 7–9 h, 10+ h.
   (f) *"Did you face challenges when using AI tools?"*
   Options: Yes, No. If "Yes", participants could specify challenges such as unclear results, difficulty integrating tools, and other (open-ended response).

Responses concerning helpfulness were recorded using a five-point Likert scale ranging from "not helpful" to "very helpful". Ease of use was examined through questions about the challenges associated with integrating AI tools into participants' workflows. Open-ended questions invited participants to describe barriers to usability, such as unclear

or inaccurate results. This approach provided qualitative insights into the effort required to effectively use AI tools in the context of the programming project.

Responses to open-ended questions were analyzed qualitatively. Task descriptions, reported challenges, and other free-text responses were grouped into categories using an iterative coding process. Rather than being predefined, the categories emerged from the responses and were refined as recurring themes became apparent during the analysis. The coding was performed by one researcher, and ambiguous cases were discussed with the co-authors until a consensus was reached.

To assess the overall usefulness of these tools, participants were additionally asked to provide a holistic rating of their experience throughout the project. The survey also examined participants' behavioral intentions, including their willingness to use AI tools in future projects.

3. *Behavioral intentions and final reflections:*
   (a) *"How do you rate the overall usefulness of AI tools in the programming project?"*
   Scale: 1 (not helpful) to 5 (very helpful).
   (b) *"Would you use AI tools in future programming projects?"*
   Scale: 1 (very unlikely) to 5 (very likely).
   (c) *"Could you have achieved similar results without AI?"*
   Options: Yes (without extra effort), Yes (with more effort), No (AI was essential).
   (d) *"What new or additional AI tools or features would you find helpful?"*
   Open-ended response for suggestions and improvements.
   (e) *"Do you have any additional comments or feedback regarding the use of AI in your project?"*
   Open-ended response for qualitative feedback.

5. User Study Results

Of the 38 participants in this study, 34 (89.4%) reported using AI tools during the programming project. The remaining four participants stated that they did not use AI tools, citing either a lack of need or personal reservations. In addition, one participant reported being unaware of the AI tools available.

As shown in Figure 2, the use of AI tools varied substantially across the four phases of the programming project. Adoption was highest during the *Coding & Testing* phase and lowest during the *Design* phase, with intermediate levels observed during the *Preparation* and *Analysis* phases. This pattern suggests that students perceived greater value in AI support for implementation-related activities than for design-oriented tasks. Notably, ChatGPT was used by all students who reported using AI tools, regardless of the project phase. The second most frequently used tool was GitHub's Copilot, particularly during the *Coding & Testing* phase, whereas other tools were used only rarely.

5.1. Preparation Phase

During the *Preparation* phase, 19 responses were collected regarding the tasks participants completed using AI tools. Of these, 18 responses could be assigned to specific task categories, while one response was too vague to be included in the analysis. The categorized responses illustrate the different ways in which students used AI tools in their work, highlighting both the diversity of applications and the challenges encountered.

The most frequently reported task category was comprehension ($n = 9$). Students relied on AI tools to understand an existing code base that served as the foundation for their later project work. This included engaging with the jMonkeyEngine, design patterns, and architectural decisions, all of which required AI support to explain code

snippets and clarify technical concepts. However, comprehension tasks were also often associated with challenges. Of the nine participants who used AI for comprehension, six reported encountering "inaccurate results", while two mentioned "missing context", referring to responses that did not sufficiently account for the existing code base or domain-specific frameworks.

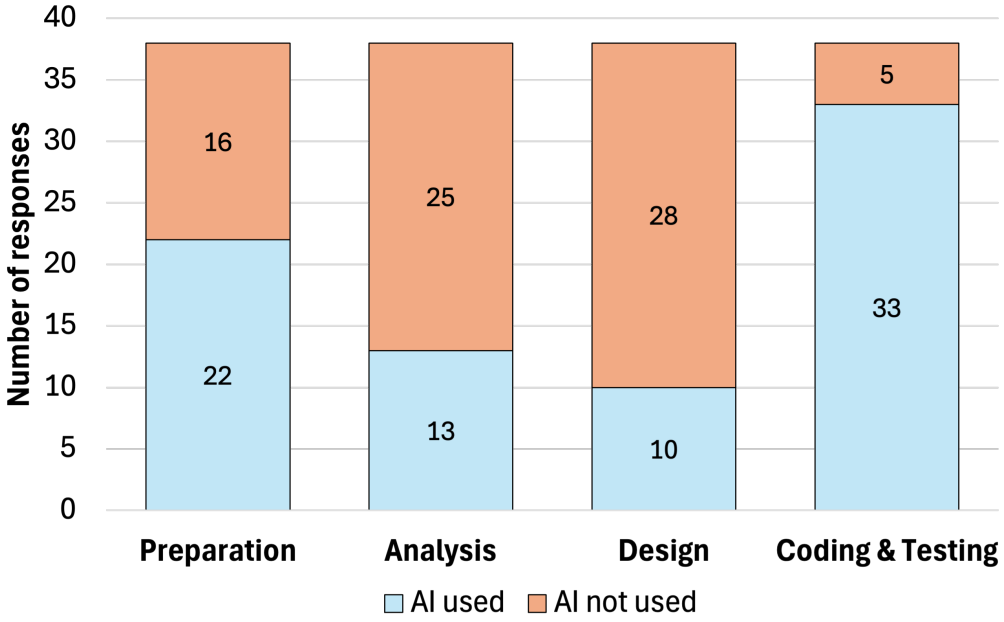

Figure 2. AI tool adoption across the project phases. Bars show the number of respondents who reported using or not using AI tools in each project phase ($n = 38$).

Code generation tasks ($n = 4$) were closely related, as students were required to implement additional features in the existing project. Of these participants, three reported problems with "inaccurate results", which required substantial correction of the AI-generated code snippets. This finding points to a broader limitation of AI tools in producing reliable code when the underlying logic or requirements are highly specific.

Similarly, code improvement ($n = 3$) emerged as a recurring theme, as students used AI to incorporate feedback from their supervisors into their solutions. However, two participants reported frustration with generic or irrelevant suggestions that could not be meaningfully integrated into the existing code.

The use of AI for documentation ($n = 3$) was also notable and is consistent with the fact that staff had indicated AI could be used for creating and refining code documentation. Although documentation tasks were comparatively straightforward, one participant nevertheless reported challenges related to "inaccurate results". Debugging ($n = 2$) was surprisingly underrepresented, despite being an introductory task in this phase in which students were asked to identify bugs in the provided code base. Both participants who used AI for debugging reported difficulties caused by the AI's inability to account for the broader project context, which resulted in ineffective or misleading suggestions. Finally, one participant mentioned image generation ($n = 1$), although the specific artifact produced remains unclear.

These findings highlight the multiple roles that AI tools played during the *Preparation* phase, while also revealing substantial limitations in their contextual understanding and output reliability when applied to unfamiliar code. The recurring issues of inaccuracy, lack of context, and overly generic suggestions indicate that, although AI tools were often

perceived as enhancing productivity, their use requires careful oversight and thoughtful integration into specific workflows.

5.2. Analysis Phase

During the *Analysis* phase, 13 participants reported using AI tools, with 11 of them describing specific tasks that could be assigned to five different categories. None of the responses had to be excluded because of vagueness or lack of interpretability.

Requirements specification ($n = 8$) was the most frequently reported task. In this activity, students created their own version of the requirements document, describing how the game to be developed should function. We assume that AI was helpful in this context because text generation is one of its major strengths, and many AI systems are familiar with popular games that students adapted into software during the project. Since no complaints were reported, this appears to have been a successful application of AI tools for supporting the formulation of precise and comprehensive requirements. This was followed by use case generation ($n = 4$), in which AI tools supported the creation of structured descriptions including the use case ID, priority, and key elements such as purpose, actors, and conditions. This repetitive and systematic task benefited from AI support by streamlining the process. However, one participant noted that the results were unreliable and required manual correction.

Documentation tasks ($n = 4$) focused primarily on the creation of user manuals. Among other aspects, these tasks required detailed descriptions of the game's components, controls, and gameplay elements. As this was likewise a text-intensive activity, it benefited from the same strengths of AI observed in the requirements specification task discussed above. No complaints were reported.

Diagram generation ($n = 2$) in this phase involved attempts to create class diagrams that were relatively simple in structure and complexity, as they mainly described the preliminary classes and overall structure of the project. However, both responses were accompanied by complaints that the generated diagrams were either incorrect or insufficiently detailed. The difficulty of AI tools in producing meaningful diagrams therefore became quickly apparent to the students. Finally, GUI mockups ($n = 2$) were used to design and present the user interface at this early stage and to support discussions with the "customer" about the intended appearance of the program. Here, the strength of AI in image generation likely played an important role by enabling students to visualize and iteratively refine their designs. Despite the small number of explicit mentions, several groups used AI-generated images in their projects, which was evident in the final presentations.

Although only about one-third of the participants reported using AI in this phase, its overall helpfulness was rated positively and only a few problems were reported. Text-intensive tasks such as requirements specification and documentation clearly benefited from the strengths of AI tools. At the same time, challenges such as insufficient detail and unreliable output underline the importance of critically evaluating and refining AI-generated content to ensure alignment with project requirements.

5.3. Design Phase

Of the 10 participants who reported using AI tools, 9 provided specific tasks that could be categorized into four groups. Once again, no response had to be excluded because it was uninterpretable.

The most frequently reported task was the creation of diagrams and charts ($n = 5$), which in this phase included flowcharts, class diagrams, package diagrams, sequence diagrams, and state diagrams. These artifacts are intended to visualize the structural and behavioral aspects of the software and to serve as a transition from conceptual work

to concrete implementation. However, as already observed in the previous phase, this finding must be interpreted in light of the limitations of AI support for such artifacts. Two participants reported that the generated diagrams were insufficient, and one student specifically noted a lack of relevant content in the generated diagram.

Another important task category was the evaluation and comparison of design approaches ($n = 4$). This involved assessing alternative design strategies in order to select effective solutions, such as thick-client and thin-client models in a network-based architecture. One participant reported the difficulty of formulating a prompt with an appropriate amount of context to elicit a satisfactory response.

The optimization of design prototypes ($n = 3$) was also mentioned, with students asking AI tools to improve their ideas. No problems were reported for this task category.

Finally, the creation of images ($n = 1$) was mentioned, as one student reportedly continued refining GUI elements. No problems were reported in this case.

Although most of the reported tasks were completed without explicit problems, these findings must be interpreted in light of the relatively low adoption of AI tools in this phase. Only 10 of the 38 students who participated in the study reported using them. This may be attributable to the strong emphasis on diagram-based artifacts, for which students reported insufficient AI-generated results. Diagram-based design artifacts have requirements that differ from those of text-intensive tasks. For such artifacts, the AI tool must generate a fluent description, but then also select and relate the elements for the respective diagram type, such as classes, states, interactions, or communication paths. These elements must remain consistent with the requirements developed in the *Analysis* phase and with the architectural decisions and communication protocol described in the software specification document, because the diagrams are intended to guide the subsequent implementation. Moreover, diagrams are used as visual representations of structural and behavioral aspects of the software. If related elements are not grouped appropriately, relations are difficult to follow, or the layout is visually cluttered, the diagram becomes hard to read and discuss even when parts of the underlying design description appear plausible. As a result, AI-generated diagrams may be incomplete, incorrect, visually unclear, or insufficiently aligned with the actual project structure, which makes them less reliable for students during the *Design* phase. Because such artifacts play a central role in communicating, reviewing, and refining software designs, these limitations represent an important challenge for the effective integration of AI into software engineering education.

5.4. Coding & Testing Phase

This phase exhibited the highest adoption of AI tools, with 33 of the 38 participants reporting their use. All 33 participants identified at least one task for which AI provided support. These tasks were grouped into six categories. At the same time, problems were widespread: 27 participants reported difficulties, and 24 mentioned one or more specific challenges in their responses.

The tasks of code generation, code improvement, and code understanding, which were grouped together because of their thematic overlap and frequent co-occurrence ($n = 28$), were widely reported by participants. Because this phase focused primarily on implementing the designs developed in the previous phase, students encountered a variety of challenges. The most frequently reported problem, mentioned by 10 participants, was incorrect code. The generated code often failed to integrate with the existing code base, either because of logical errors or because existing classes and methods were overlooked or misinterpreted.

One participant explicitly reported encountering “hallucinations”, in which the AI generated incorrect code based on framework methods that did not exist. Six participants

noted difficulties arising from the fact that some AI models were not trained on domain-specific frameworks or libraries, such as JMONKEYENGINE or LEMUR, which resulted in output that was incompatible with their projects. Three participants reported that AI tools struggled to handle the complexity of their projects because they were unable to provide the entire code base or large numbers of classes as input to the system.

As a result, the generated output often conflicted with the existing code and did not integrate seamlessly. Finally, two participants highlighted prompting-related difficulties, attributing their problems to an inability to communicate their requirements effectively to the AI. Similar challenges were observed across tasks, particularly in code understanding and code improvement, as the AI performed better with standard Java but often failed when specialized frameworks were involved.

Documentation tasks ($n = 24$) were also among the most frequently reported. Participants used AI tools to create or refine project documentation. At this stage, documentation primarily involved writing JAVADOCS for classes. Many students reported that AI was particularly effective in analyzing methods and generating comprehensive, contextually appropriate documentation. Only one participant reported an erroneous AI-generated document, suggesting that the tools performed well in this context. However, this observation should not be generalized to programming-related tasks, which involved substantially greater contextual and integration challenges.

Debugging tasks ($n = 20$) were closely related to code generation, as both required AI tools to operate within existing code bases and project frameworks. Five participants reported that the AI had difficulty with unfamiliar frameworks or libraries, similar to the issues observed with code generation. Three participants reported incorrect output during debugging, including one case in which the AI "hallucinated" by generating nonsensical code that was incompatible with the project context. In other cases, the proposed solutions failed to address the actual problem or conflicted with the existing implementation, thereby complicating integration. One participant also noted prompting-related difficulties, particularly in describing complex code problems or requesting contextually relevant methods.

Test generation ($n = 7$) was another area in which AI tools were used, specifically for testing the model developed within the *Model–View–Controller* architecture [47]. Although this use was generally effective, some problems were reported. One participant noted that poorly specified tasks led to outputs that did not meet the intended goals. Another participant observed that, in some cases, the AI generated nonsensical tests. Finally, general questions ($n = 1$) and music generation ($n = 1$) were also mentioned, although no problems were reported for these tasks.

Although the *Coding & Testing* phase showed the highest adoption of AI tools among participants, it also revealed substantial challenges, particularly in code generation, debugging, and the improvement of existing implementations. The tools proved valuable for automating repetitive tasks and generating initial solutions, but recurring problems such as incorrect output, difficulties with domain-specific frameworks such as JMONKEYENGINE, and limitations in handling complex code bases underscored the need for careful oversight. Despite these challenges, participants reported particularly positive experiences with documentation and, to a lesser extent, test-generation tasks. These activities are comparatively well-bounded and text-oriented and therefore differ from programming and debugging tasks, where context awareness and integration with the existing code base proved substantially more demanding.

Overall, these findings indicate that the high adoption of AI tools during the *Coding & Testing* phase reflects both their perceived usefulness and the substantial support needs associated with implementation tasks. At the same time, the large number of reported

difficulties demonstrates that widespread use should not be equated with straightforward effectiveness. Rather, the results highlight a tension between the demand for AI assistance and the current limitations of AI tools in handling project-specific context, framework dependencies, and integration challenges.

5.5. Additional Observations

In addition to analyzing specific project phases, participants were asked whether they believed they could have achieved the same results without using AI tools. Seven participants indicated that they could have achieved comparable results without AI, while twenty-four stated that doing so would have required significantly more effort. Notably, three participants reported that AI had been critical to their success. These responses underscore the substantial role AI played in supporting participants throughout the project. Although some participants were confident that they could have achieved similar results without AI, the majority acknowledged the considerable time and effort saved by integrating these tools into their workflows, as illustrated in Figure 3. A smaller but still notable group regarded AI as essential.

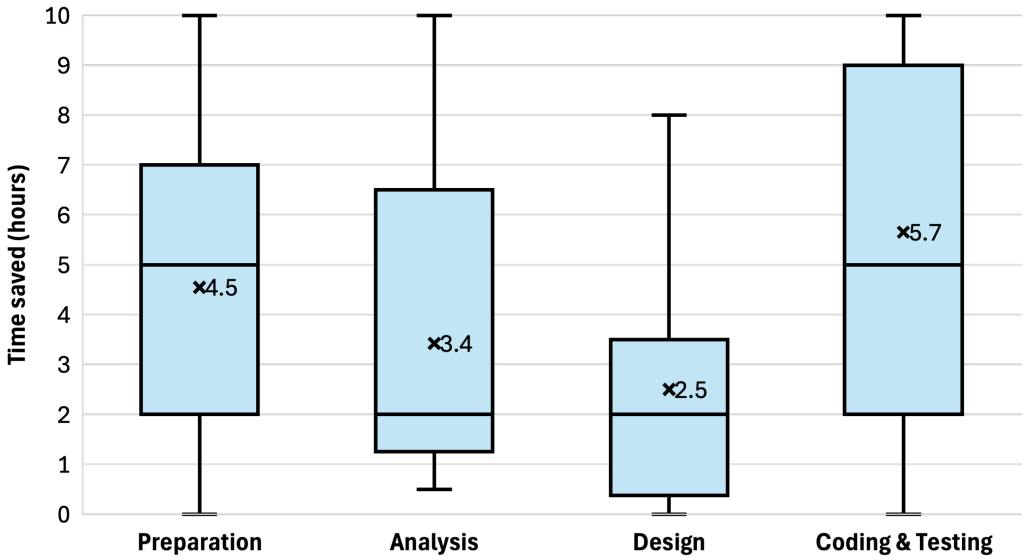

Figure 3. Self-reported time savings by project phase. Boxplots summarize phase-specific responses in hours for participants who reported AI use in the respective phase (Preparation: $n = 22$; Analysis: $n = 13$; Design: $n = 10$; Coding & Testing: $n = 33$). Horizontal lines indicate medians, and crosses indicate mean values.

Interestingly, this broadly positive perception of AI contrasts sharply with the gap analysis emerging from the study. While participants generally acknowledged the productivity potential of AI tools and expressed a strong willingness to reuse them in future projects, the reported benefits were unevenly distributed across development phases and consistently constrained by practical limitations. Self-reported time savings varied substantially, with the largest gains observed in the *Coding & Testing* phase and the smallest in the *Design* phase. At the same time, participants repeatedly reported inaccurate or incomplete outputs, integration difficulties, and the limited ability of current AI tools to cope with complex project-specific contexts [48]. Taken together, these findings suggest that the current value of AI lies less in end-to-end problem solving than in accelerating well-structured, repetitive, and locally bounded tasks, such as documentation and test generation, while leaving higher-level design integration and context-sensitive reasoning as persistent gaps.

The waterfall process model used in the course helps to interpret these differences, but the relevant distinction is not between the process models. In our setting, the development process is organized into project phases, which allows us to relate the observed patterns to the tasks and artifacts associated with the *Preparation*, *Analysis*, *Design*, and *Coding & Testing* phases. This task-oriented interpretation is consistent with recent findings showing that developers use AI-based coding assistants unevenly across software development activities and stages [49]. Work on generative AI in agile software development makes a related distinction below the overall process model by discussing support for recurring development activities such as *Planning*, *Implementation*, *Testing*, *Maintenance*, and *Retrospectives* [50]. Although these settings differ in how they organize development work, they share the goal of distinguishing the tasks and artifacts for which AI support is used. In our study, AI support was perceived as most useful for text- and code-oriented tasks, whereas design artifacts such as diagrams remained more challenging because the generated results often lacked correctness. These patterns give an initial indication as to where a future tutor should offer more targeted support, particularly through repository grounding for code-related tasks and stronger assistance in creating, checking, and refining design-related artifacts without replacing the students' own design work.

Although the study is conducted in a waterfall-structured setting, the identified patterns are task-dependent and are therefore expected to occur in other development models as well.

6. Case Study: Repository-Aware LLM Assistant

The user study identified limited access to project-specific context as a major obstacle when applying AI tools to code comprehension and implementation tasks. As a result, generated responses often failed to align with the existing code base, leading to inaccuracies and integration problems. These findings suggest that effective AI support in software projects requires contextual grounding through direct access to relevant project artifacts.

To address this issue, we develop a repository-aware LLM assistant that retrieves relevant documentation and source code from the project repository before generating responses. Unlike the planned AI tutor introduced in Section 1, the assistant evaluated in this case study serves as a research platform for investigating repository-aware support and therefore generates complete solutions rather than pedagogically guided assistance.

This grounding is intended to improve the accuracy, coherence, and practical usefulness of the generated output. The system is deployed locally to preserve data control and to keep the evaluation reproducible and traceable. This is particularly important in a repository-aware setting because the model-serving request may include retrieved passages, source-code snippets, tool outputs, and interaction history. The local setup also allows us to control and document the model versions, sampling parameters, retrieval settings, and repository–tool interactions used in this evaluation. The design continues our broader goal of advancing context-sensitive AI support for software engineering education, here realized in the form of a solution-oriented assistant rather than a tutor.

This section outlines the architecture and local deployment of the assistant, which together form the technical foundation for the evaluation presented in the following sections.

6.1. Architecture

Our system is designed for the programming project environment, with a particular focus on the *Preparation* phase, in which students work with and extend the existing *Battleship* code base. Typical tasks in this phase include adding sound effects, introducing new game items, and extending model components with corresponding behavior and visualization. The challenges students reported when using AI—such as limited code understanding

and weak integration with existing artifacts—motivate an architecture that grounds model responses in project code rather than relying solely on prompt-based interaction.

We organize the assistant around a chat interface, a model-serving layer, and two complementary grounding paths. The first, *document-level RAG*, uses project documentation and auxiliary materials as the preferred knowledge sources. All documents are embedded for semantic retrieval so that, at query time, relevant passages can be provided as additional context. This mechanism directly addresses the text-heavy tasks and missing-context issues observed in the user study.

The second, *code-level lookup*, provides targeted access to the project's Git repository. Guided by the user query and, where applicable, the document-level retrieval results, the system locates relevant classes or files in the repository tree, retrieves their content through the repository API, and can optionally extract method-level snippets to support more focused reasoning. Repository access is strictly read-only and, at present, pinned to a fixed branch. This path is invoked only when a prompt explicitly refers to code artifacts, thereby improving integration fidelity and consistency in code-related responses.

Both grounding paths feed into a unified *prompt* $\rightarrow$ *retrieval* $\rightarrow$ *generation* pipeline. At runtime, this pipeline retrieves relevant document passages, invokes repository tools when code artifacts are referenced, appends the resulting context to the user prompt, and submits the assembled request to the model-serving endpoint through the OpenAI-compatible CHAT COMPLETIONS API. This design enables the evaluation of different model families or decoding parameters without requiring modifications to the system interfaces. For auditability and qualitative analysis, the system logs all prompts, retrieved passages, repository–tool inputs and outputs, and generated responses, while routing all traffic through an HTTP proxy that captures complete request traces for inspection and debugging.

6.2. Deployment

This subsection describes the implementation of the architecture and outlines the on-premises setup together with its operational controls for auditability and reproducibility. Figure 4 provides an overview of the local system stack and the main data flows between the chat interface, document grounding, model serving, and repository access.

All components run as Docker containers on a dedicated on-campus NVIDIA DGX-H100 system. Containerization provides isolated and reproducible execution while simplifying configuration management across study runs. The use of controlled hardware supports long-context workloads and ensures stable performance, which is critical for processing complex prompts that combine retrieved documents with multi-file code inputs.

The inference engine VLLM (https://docs.vllm.ai, accessed 9 March 2026) exposes an OpenAI-compatible CHAT COMPLETIONS (https://platform.openai.com/docs/api-reference/chat, accessed 9 March 2026) endpoint. We configure an extended context window to fully utilize the available GPU memory, thereby enabling the processing of retrieved context and large source files. Tensor parallelism is applied across multiple GPUs to maintain low latency for long-context workloads. Sampling parameters can be adjusted either through the user interface or directly within the inference engine.

Open WebUI (UI) (https://openwebui.com/, accessed 9 March 2026) connects to this endpoint and organizes study runs through model-chat configurations. As shown in Figure 4, project documents are processed by a content-extraction service, Docling (https://github.com/docling-project/docling, accessed 9 March 2026), and embedded for semantic retrieval. The resulting knowledge base is linked to each model-chat configuration, which appears as a selectable chat entry. Each configuration defines the base model, the sampling preset and model variant, the attached knowledge base containing our project

documentation, and the enabled tools. The setup also includes a short system prompt instructing the assistant to respond in German and to initialize the session once by calling `load_battleship_json` in order to load the repository's class inventory. Subsequent calls to this method are disabled. At runtime, the UI assembles an OpenAI-compatible CHAT COMPLETIONS request by combining the user prompt, retrieved context, conversation history, and the enabled tools. Each request includes the tool names, natural-language descriptions, and parameter definitions. The UI then submits the request to the endpoint, executes any tool calls returned by the model, and appends the tool outputs to the conversation state.

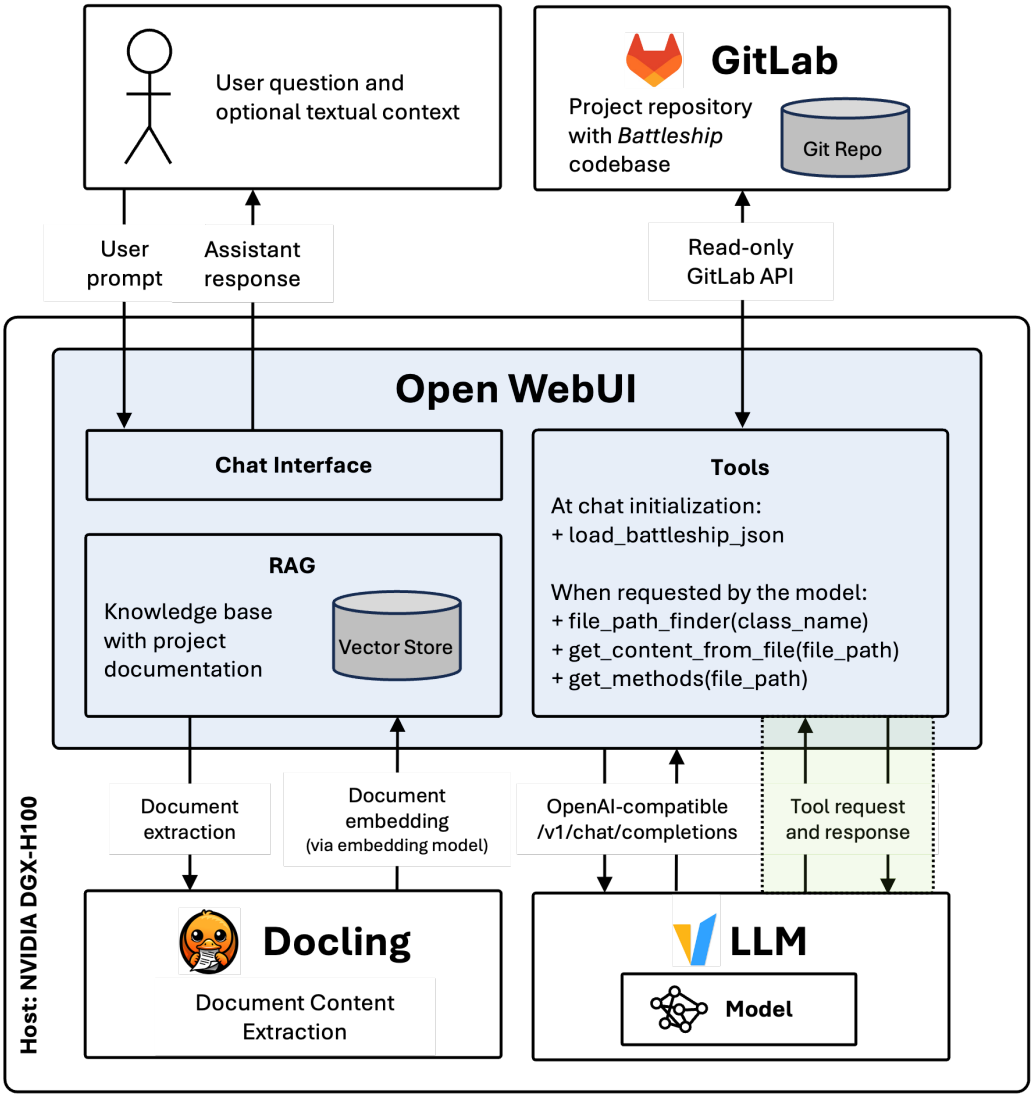

Figure 4. Local deployment stack and data flow for the repository-aware assistant, showing the Open WebUI (chat, knowledge, tools), document grounding via Docling, model serving via vLLM (OpenAI-compatible API), and repository adapters for GitLab, all running on a DGX-H100 host.

Table 3 summarizes the application stack and retrieval settings used in the case study.

Within this setup, access to the repository is provided through three dedicated tools that interface with the GitLab API. All calls target a fixed branch reference and rely exclusively on `GET` endpoints. The first tool, `file_path_finder`, searches the repository tree to locate relevant files. Given a class name as input, it returns the path of the corresponding class in the repository. Once the file has been identified, `get_content_from_file` retrieves and decodes its content, performing minimal header cleanup when necessary.

Finally, `get_methods` uses Tree-Sitter-based analysis to enumerate and inspect method names within a class, thereby enabling fine-grained reasoning about source code structure.

When a prompt references a specific class or method, the assistant first resolves the class path within the repository tree using `file_path_finder` and then retrieves the corresponding file content through `get_content_from_file`. This tool–call pipeline is illustrated in Figure 5.

Table 3. Technical configuration used in the case study.

Configuration Item	Setting
Application stack	
Chat interface	Open WebUI 0.6.5
Document extraction	Docling 2.30.0
Model serving	vLLM 0.8.5.post1
Serving parameters	`max_model_len` = 93,000; `tensor_parallel_size` = 8
Repository access	GitLab 17.10.1; read-only access via `GET` endpoints
Retrieval settings	
Embedding model	`sentence-transformers/all-MiniLM-L6-v2`
Text splitting	Character-based; `chunk size = 1000`; `overlap = 100`
Retrieval settings	Top-k = 3; hybrid search disabled; full-context mode disabled
Vector store	Chroma, persistent local client
Similarity metric	Cosine distance
Reranking	None

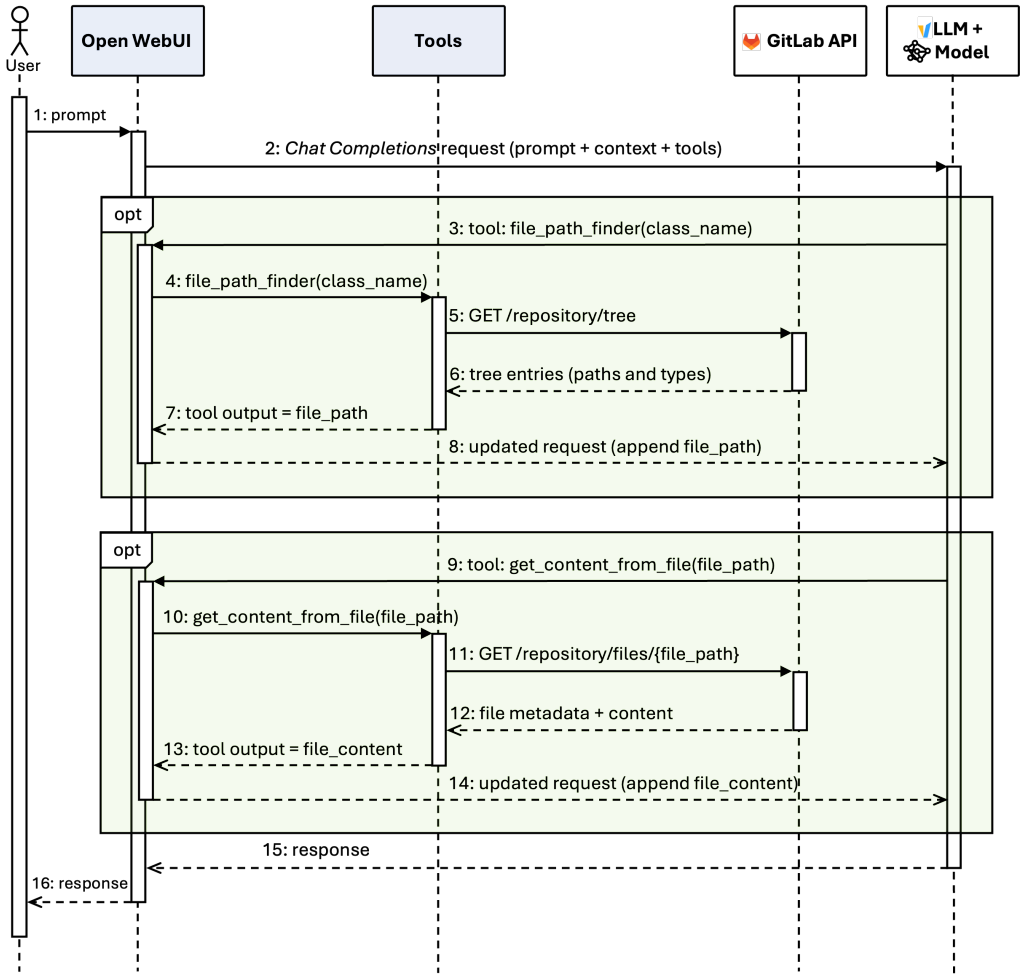

Figure 5. Exemplary tool–call pipeline: user prompt → repository path discovery (`file_path_finder`) → file retrieval (`get_content_from_file`) → grounded response.

The retrieved source code, together with any relevant document passages, is appended to the prompt as contextual input to enable the model to generate a grounded response. If the prompt refers to a particular method, the system first enumerates the method names in the corresponding class via `get_methods` to verify its existence before retrieving the code. This lookup process may be repeated across multiple turns when additional files are referenced.

All prompts, retrieved document passages, tool inputs and outputs, and model responses are logged within the UI. In addition, all traffic is routed through an HTTP proxy to capture detailed traces for debugging and auditing. For reproducibility, both the container images and the repository reference are version-pinned. The Supplementary Material contains the project context, the exported chat runs, the repository-access tool and the corresponding tool log.

We conducted the deployment and evaluation on our dedicated hardware, the NVIDIA DGX-H100, which provides substantially more computational resources than are available at many educational institutions. Although the architectural concepts presented in this paper are not tied to this specific hardware configuration, serving large models with extended context windows and tensor parallelism may not generalize directly to environments with fewer resources. Practical deployments may require smaller models, shorter context windows, reduced concurrency, or simplified retrieval configurations. Therefore, the following empirical evaluation should not be considered representative of all deployment settings.

7. Empirical Evaluation of the Case Study

This section evaluates the effectiveness of the locally deployed, repository-aware assistant in generating project-specific responses and examines how different configuration choices affect response quality. Two main factors are considered: decoding settings, represented by sampling parameters, and differences across model families and sizes. The evaluation is conducted on the system stack shown in Figure 4. The following subsections describe the evaluation design, the experimental procedure, and the defect catalog used for the qualitative assessment of the generated outputs.

7.1. Evaluation Design

The evaluation focuses on two representative tasks from the *Preparation* phase of the *Battleship* project. As described earlier, these tasks are intended to help students understand the existing code base and become familiar with the project's framework and overall architecture. The evaluation runs were conducted using the original German task formulations. For this publication, we provide faithful English translations.

Task 1 (3D Models for Ships): "Ships are currently rendered as boxes whose length corresponds to ship size. Add a 3D model representation that assigns models based on ship length. If no model exists for a given length (or if parameters change), revert to the box representation to keep the game functional."

This task requires an understanding of the existing size-dependent ship-rendering implementation and the introduction of a length-to-model mapping that preserves functionality when assets are missing or parameters change. The *Battleship* project includes ships of various lengths; therefore, the mapping must select the appropriate model for each length and provide a reliable fallback to ensure playability under different configurations. The solution should integrate into the existing rendering pipeline by identifying the relevant class, reusing existing components, and adding only minimal control logic to check model availability and activate the fallback when necessary. A dynamically sized box representation is already available and should serve as the default. This task was selected because of its moderate complexity and because a 3D model for ships of length

four already exists in the project, allowing the evaluation to focus on repository awareness and small, safe adaptations rather than extensive rewrites. Given this scope, we used the task description verbatim as the initial prompt (*Prompt A1*).

Task 2 (Background Music): "The game already provides sound effects. Add suitable background music that can be enabled and disabled in the menu independently of the sound effects. In addition, the volume of the background music should be adjustable in the menu using a slider."

This task focuses less on isolated code snippets and more on correct architectural integration. Background playback should be managed by a persistent application-state component that participates in the game's update lifecycle rather than being triggered ad hoc. It should register a dedicated music channel that is decoupled from the existing sound-effects path so that user controls can operate independently. Within the user interface, the menu serves as the container for the graphical elements: a checkbox should toggle the background-music state, while a slider should be bound to a small state model (e.g., the volume level) maintained by the audio controller in order to preserve a clear separation between view and logic. A clean solution therefore spans multiple classes and requires repository awareness to identify and extend the relevant code paths.

During preliminary testing on our deployed stack, early pilot runs consistently failed to produce a functional slider implementation. We therefore excluded the slider subtask in order to avoid introducing an additional UI pathway that would not affect the comparative conclusions. Accordingly, the evaluation was operationalized using two prompts issued within the same chat: the first established continuous background playback, and the second implemented an independent on/off control in the menu.

Prompt B1: "How can I implement background music in the game?"

Prompt B2: "How can I implement a way to enable or disable the background music independently of the sound effects? Use the class `Menu.java`."

7.2. Experimental Procedure

Each evaluation run is defined as a new chat instance corresponding to a specific combination of model, sampling configuration, and task, using the technical configuration summarized in Table 3. No re-runs are performed. To ensure comparability across runs, we fix the repository snapshot and document index and enable the same tool set in all sessions.

During each interaction, knowledge retrieval is enabled for the associated workspace knowledge base. Repository lookups are performed either when explicitly requested by the model or when code artifacts are required. The available tools are accessible to the model at every prompt and, when invoked, are executed through the user interface. The resulting outputs are then incorporated into the conversation state to inform subsequent model responses. This orchestration follows our *prompt* $\rightarrow$ *retrieval* $\rightarrow$ *generation* pipeline.

Configuration effects are examined in two stages. In the first stage, a sampling sweep (summarized in Table 4) is conducted using the *Mistral-Large-Instruct-2411* model, with the parameters *temp*, *top_p*, and *min_p* systematically varied in order to explore the trade-off between diversity and stability in code generation.

In the second stage, a model sweep is conducted with the sampling parameters fixed at $temp = 0.5$, $top\_p = 0.95$, and $min\_p = 0$, while varying model families and model sizes. These values were selected a priori, informed by a brief pre-study, in order to balance response consistency with controlled variability in the generated outputs. A temperature of 0.5 reduces the stochastic variation compared with the default setting. This is important for code-centric tasks where small deviations can lead to incorrect methods, APIs, or integration steps. The *top-p* value of 0.95 applies only mild nucleus filtering because it excludes the least likely tail of the token distribution, but it also preserves enough flexibility for explanatory

and code-generating responses. We did not tune these parameters separately for individual models, because the purpose of the model sweep was to compare model families and sizes under identical decoding conditions. The model lineup spans multiple families and sizes, including both code-specialized and general-purpose variants. To ensure comparability, the task prompts remain identical across both stages.

Table 4. Sampling configurations used in the evaluation.

	temp	**top_p**	**min_p**
Standard (no change)			
Default	1.0	1.0	0.0
Isolated change			
temp 0.5	0.5	1.0	0.0
temp 0.3	0.3	1.0	0.0
top_p 0.8	1.0	0.8	0.0
top_p 0.5	1.0	0.5	0.0
min_p 0.1	1.0	1.0	0.1
min_p 0.3	1.0	1.0	0.3
Combined change			
temp 0.5 + min_p 0.1	0.5	1.0	0.1
temp 0.5 + min_p 0.3	0.5	1.0	0.3
temp 0.3 + min_p 0.1	0.3	1.0	0.1
temp 0.3 + min_p 0.3	0.3	1.0	0.3
top_p 0.8 + min_p 0.1	1.0	0.8	0.1
top_p 0.8 + min_p 0.3	1.0	0.8	0.3
top_p 0.5 + min_p 0.1	1.0	0.5	0.1
top_p 0.5 + min_p 0.3	1.0	0.5	0.3

We evaluated the following models.

- *DeepSeek-Coder-V2-Instruct* (236B parameters) (https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Instruct, accessed 15 May 2026)
- *Devstral-Small-2505* (24B parameters) (https://huggingface.co/mistralai/Devstral-Small-2505, accessed 15 May 2026)
- *Mistral-Large-Instruct-2411* (123B parameters) (https://huggingface.co/mistralai/Mistral-Large-Instruct-2411, accessed 15 May 2026)
- *Mistral-7B-Instruct-v0.3* (7B parameters) (https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3, accessed 15 May 2026)
- *Qwen3-235B-A22B* (235B parameters) (https://huggingface.co/Qwen/Qwen3-235B-A22B, accessed 9 March 2026)
- *Qwen3-30B-A3B* (30B parameters) (https://huggingface.co/Qwen/Qwen3-30B-A3B, accessed 15 May 2026)

To ensure comparability, we disabled model-native *function calling*, where available, and routed all tool use through the same UI-controlled path for all models. Each configuration–task combination was executed exactly once. Task 1 was issued as a single prompt, whereas Task 2 used its two-step prompt sequence within the same chat. Beyond these predefined steps, no restarts or re-runs were performed.

For auditability, we recorded all user prompts, retrieved document passages, repository tool inputs and outputs, and model responses from the user interface logs. These data were complemented by HTTP-level traces captured through the proxy to support transparency and reproducibility.

The generated model responses were then analyzed qualitatively using the defect catalog described below. These recorded materials served as contextual evidence for

interpreting response failures, for example, cases in which retrieved context was available but not incorporated into the generated response. The defect catalog was developed during the pre-study, refined during the evaluation and then used in its finalized form for coding the generated responses. One researcher coded the sampling-sweep outputs and another researcher coded the model-sweep outputs. Ambiguous cases were interpreted in light of the recorded prompts, retrieved passages, and repository-tool outputs. When uncertainty remained, the other researcher was consulted.

7.3. Defect Catalog

The finalized defect catalog used for the qualitative analysis groups the observed defect types into four broader defect families: *grounding and tool-use defects*, *task- and requirement-level defects*, *repository integration defects*, and *code-level implementation defects*. The catalog comprises the following defect types, grouped by defect family.

Grounding and tool-use defects

1. *Hallucination.* The model refers to or invokes code artifacts that do not exist in the repository but are assumed to be available. Typical cases include:
   (a) Referencing undefined or nonexistent types.
   (b) Invoking methods that are not defined for an existing class or type.
2. *Tool misuse.* Incorrect invocation or handling of repository tools. Examples include:
   (a) Using a repository tool with an incorrect target (e.g., the wrong class or method).
   (b) Retrieving the correct artifact but failing to incorporate the result into the generated response.

Task- and requirement-level defects

3. *Task misunderstanding.* Partial or incorrect interpretation of the user's request. For instance, the model may implement back-end logic but fail to expose the corresponding functionality in the user interface.
4. *Insufficient robustness.* The proposed code lacks defensive checks for missing or unloadable resources and fails to degrade gracefully under error conditions. Common examples include:
   (a) Constructing a resource path dynamically without validating its existence.
   (b) Mentioning a fallback mechanism but omitting exception handling, thereby leaving failures unhandled.

Repository integration defects

5. *Missing resource entry.* Adding new non-code resources without registering them properly. For example, this may involve adding a new UI element with a text label while omitting the corresponding localization key or value in the resource bundle.
6. *Integration omission.* Failing to reuse existing code paths or helper functions when extending functionality. This often leads to incomplete or inconsistent behavior, such as omitting calls to necessary utility methods.

Code-level implementation defects

7. *Use-before-initialization.* Accessing objects that have not yet been constructed or fully initialized. Typical occurrences include:
   (a) Passing a `null` reference to a method that expects an instance.
   (b) Invoking a method on a field before its initializer has been executed.
8. *Duplicate variable declaration.* Declaring the same identifier multiple times within the same scope, such as redeclaring a local variable within a method body.

9. *Type-mismatched comparison.* Using comparison operators on operands of incompatible data types. For instance, this may involve comparing a method's enum return value with a constant from a different enumeration.
10. *Invalid invocation.* Performing illegal or context-inappropriate method calls or field accesses. A common case is attempting to access instance fields from a static context.
11. *Wrapper-only method.* Creating a method that merely delegates to another method without adding meaningful logic. A typical example is calling an overloaded method with mapped parameters and immediately returning its result.
12. *Code duplication.* Repeating the same logic across multiple methods with only minor variations. For example, two methods may share nearly identical bodies that differ only in a constant or parameter.

8. Results of the Case Study Evaluation

This section presents the results of the case study following the two-stage evaluation. In total, 42 runs were conducted, comprising 15 sampling configurations and 6 model variants, each evaluated on two tasks. Across both stages, the repository-aware setup consistently produced usable, repository-aligned outputs. In most runs, the generated artifacts could be integrated into the existing code base with only limited follow-up corrections. This contrasts with the generic assistants observed in our user study, which frequently lacked sufficient project context to support seamless integration.

All generated outputs were analyzed using the defect catalog. Each observed issue was assigned to a defect type and, where applicable, to a case variant. Tables 5 and 6 group these defect types by the defect families introduced in the catalog. The following subsections first discuss the results of the sampling sweep, then those of the model sweep, and finally the identified threats to validity.

8.1. Sampling Sweep

Table 5 summarizes the defects observed across the 15 sampling configurations.

A clear pattern emerged with respect to *Hallucination*, which was the most frequent defect, occurring in 14 of the 15 runs and clustering most prominently at $temp = 1.0$. Even when the token selection range was restricted using $top\_p \in \{0.5, 0.8\}$ or $min\_p > 0$, hallucinations still occurred—typically as isolated instances rather than as the multiple occurrences (two or three) observed at higher temperatures.

Beyond *Hallucination*, the other defect types did not occur consistently across different levels of decoding noise. The defect *Missing resource entry* was unaffected by sampling variation and recurred even in otherwise clean configurations. Because our system logs all prompts, responses, tool operations, and retrieved document passages, we can confirm that the required resource files were present. This pattern indicates a retrieval-to-action gap, in which relevant context is successfully retrieved but not incorporated into the generated output. Apart from *Hallucination* and *Missing resource entry*, the remaining observed defects were too sparse, given the single-run design, to support parameter-specific conclusions. Defects such as *Type-mismatched comparison* and *Invalid invocation* occurred only sporadically. *Wrapper-only method* and *Code duplication* were observed exclusively at $temp = 1.0$, $top\_p = 1.0$. We therefore interpret these as low-base-rate phenomena rather than systematic effects. Additional repetitions of the runs would be required to assess variance and confirm the stability of these findings.

In conclusion, the configuration $temp = 0.5$, $top\_p = 1.0$, $min\_p = 0.0$ was the most stable configuration in our evaluation, yielding the fewest defects across all runs. Configurations combining $temp = 1.0$ or a positive min_p with additional filtering (e.g.,

$(1.0, 1.0, 0.1)$, $(1.0, 0.8, 0.1)$) consistently produced multiple issues. Accordingly, we adopt a conservative default configuration for code-centric applications.

Table 5. Results of the sampling sweep. The upper blue entries indicate defects observed in Task 1, whereas the lower red entries indicate defects observed in Task 2. Letters *a* and *b* refer to the case variants defined in the defect catalog. Entries marked with `x` indicate a defect type without a specific case variant, while `x2` and `x3` denote repeated occurrences. Empty cells indicate that the corresponding defect type was not observed.

			Grounding and Tool-Use Defects		**Task- and Requirement-Level Defects**		**Repository Integration Defects**	**Code-Level Implementation Defects**						
temp	**top_p**	**min_p**	**Hallucination**	**Tool Misuse**	**Task Misunderstanding**	**Insufficient Robustness**	**Missing Resource Entry**	**Integration Omission**	**Use-Before-Initialization**	**Duplicate Variable Declaration**	**Type-Mismatched Comparison**	**Invalid Invocation**	**Wrapper-Only Method**	**Code Duplication**
Standard (no change)														
1.0	1.0	0.0	b				x	x						
Isolated change														
0.5	1.0	0.0					x							
0.3	1.0	0.0	b	a			x							
1.0	0.8	0.0	a b			a	x							
1.0	0.5	0.0	a b				x							
1.0	1.0	0.1	b x2				x		x					x
1.0	1.0	0.3	b x2				x						x	
Combined change														
0.5	1.0	0.1	b			a	x		b		x			
0.5	1.0	0.3	b				x	x						
0.3	1.0	0.1	b	a			x	x						
0.3	1.0	0.3	b				x			x	x			
1.0	0.8	0.1	b x3			a	x							
1.0	0.8	0.3	b x2				x							
1.0	0.5	0.1	b				x					x		
1.0	0.5	0.3	b			a	x							

8.2. Model Sweep

Table 6 presents the defect profiles of six models evaluated under fixed decoding parameters and identical task prompts. A consistent pattern was observed across model families: in the initial prompt, the models often identified the correct class, but hallucinated its content. After being instructed to retrieve the corresponding file, the generated outputs were adjusted accordingly, and the hallucinations did not recur. Therefore, these early but self-corrected instances were not included in the defect catalog.

Table 6. Results of the model sweep. The upper blue entries indicate defects observed in Task 1, whereas the lower red entries indicate defects observed in Task 2. Letter *b* refer to the case variants defined in the defect catalog. Entries marked with `x` indicate a defect type without a specific case variant, while `x2` denotes repeated occurrences. Empty cells indicate that the corresponding defect type was not observed.

	Grounding and Tool-Use Defects		**Task- and Requirement-Level Defects**		**Repository Integration Defects**			**Code-Level Implementation Defects**				
Model (Abbrev.)	**Hallucination**	**Tool Misuse**	**Task Misunderstanding**	**Insufficient Robustness**	**Missing Resource Entry**	**Integration Omission**	**Use-Before-Initialization**	**Duplicate Variable Declaration**	**Type-Mismatched Comparison**	**Invalid Invocation**	**Wrapper-Only Method**	**Code Duplication**
DeepSeek-Coder-V2	b	b			x							
Devstral-Small-2505	b				x							
Mistral-Large	b				x							
Mistral-7B			x		x	x						x
Qwen3-235B-A22B	b								x			
Qwen3-30B-A3B	b x2			b	x							

Two notable tendencies emerged despite the single-run design. First, *Hallucination* was not confined to a particular model family or size, as every model except *Mistral-7B* exhibited at least one marked instance. For *Mistral-7B*, the *Hallucination* defect type remained unannotated only because the run did not produce complete solutions for either task. Second, the *Missing resource entry* defect recurred across almost all models. As in the sampling sweep, this defect appears to arise not from decoding noise but from a retrieval-to-action gap. Once again, the system logs confirmed that the relevant documentation had been successfully retrieved, yet the required property entry was never written.

Among the large models, *DeepSeek-Coder-V2-Instruct* ultimately solved both tasks with repository-aware code, although it required an explicit follow-up prompt before identifying the correct classes. *Mistral-Large-Instruct-2411* produced a clean design and demonstrated coherent linking between components. *Qwen3-235B* generated the most comprehensive rationale and was the only model to recognize the need to update the resource file, although it still required corrections to produce a fully functional solution.

Among the mid-sized models, *Devstral-Small-2505* performed well relative to its size: it successfully completed the mapping task, and, uniquely, identified persistent user preferences in Task 2. *Qwen3-30B* completed the ship-model mapping but failed to connect all relevant components. The smallest model, *Mistral-7B*, proved unsuitable in this context: multiple follow-up interactions were required, partial code was generated without integration into the existing workflow, and the run did not reach completion.

Two exploratory observations can be drawn from these results. First, within the scope of this evaluation, model size alone did not predict success in our repository-aware setup: for example, the 23.6B *Devstral-Small-2505* produced the most usable outputs despite its medium size. One important difference was whether a model used the available repository context to produce repository-aligned outputs. The tasks required more than plausible code snippets, because the model had to incorporate the retrieved information while preserving the existing project structure. The availability of relevant context alone did not prevent failures when retrieved information was not carried through to the generated response, as reflected in the recurring retrieval-to-action gap. Conversely, a mid-sized model produced usable outputs when it used the retrieved information to ground the generated response. Second, the recurring defects across models indicate that improving tool use has a greater impact than switching model families when operating under similar constraints. The evaluation does not isolate training data, model architecture, or instruction tuning as separate causes. These factors must be considered together with model size and tool use when interpreting the observed differences.

8.3. Threats to Validity

The results of this study should be interpreted in light of its limitations. First, the empirical evaluation is based on a single course at one university, with a sample size of 38 participants. This limits the external validity and generalizability of the findings; while the study design provides in-depth insights into a realistic, project-based educational setting, the observed effects may not directly apply to other institutional contexts or learner populations.

The user-study data are based on retrospective self-reports collected during the final project presentations. Consequently, participants had to recall their AI usage, perceived usefulness, time savings, and the challenges they encountered across the different project phases. This may introduce recall bias and self-reporting bias, as participants may not accurately remember all interactions with AI tools or may overestimate or underestimate their effects. Although the questionnaire was informed by TAM, it was not based on a validated TAM instrument and no psychometric construct analysis was performed. Consequently, the results should be interpreted as descriptive indicators of student perceptions rather than as validated measurements of TAM constructs.

Contextual factors such as course design, student background, prior experience with AI tools, and instructional setting likely influence usage patterns and the perceived usefulness of repository-aware AI support. These factors may introduce biases that are difficult to disentangle in a single-site study. Therefore, to validate the findings and assess their robustness in more diverse educational environments, future studies should replicate and extend this research across multiple courses, programming languages, project types, institutions, and larger student populations.

Additionally, using only a single run for each configuration–task combination limits the reproducibility and robustness of the observed effects in a stochastic setting. The analysis relies on manual coding based on the defect catalog described earlier. As a result, the analysis emphasizes the occurrence and distribution of defect types rather than their relative impact on task completion or code quality. Future work could complement the

present catalog with severity ratings to support more fine-grained comparisons of defect consequences. The generated outputs were not independently double-coded, and no formal inter-rater reliability metric was computed. Future studies should complement the present approach with repeated runs for selected configurations and independent double-coding in order to assess result stability and coding reliability more systematically. Although the logs capture prompts, tool invocations, and retrieved passages, the absence of a standardized benchmark introduces subjectivity and the potential for coder bias.

We did not conduct a separate quantitative evaluation of retrieval accuracy. Therefore, the assessment of retrieval behavior is based on the qualitative interpretation of the generated responses, considering the recorded prompts, the retrieved passages, and the repository-tool outputs. Consequently, no retrieval-specific metrics such as precision, recall, or top-k relevance were computed. Evaluating retrieval performance independently of the generation component would require a dedicated benchmark with annotated relevant passages and repository artifacts, which was beyond the scope of the present study.

The retrieval-to-action gap identified in this study was inferred from a qualitative analysis of prompts, retrieved passages, tool outputs, and generated responses, while several cases indicated that relevant information was retrieved but not incorporated into the final response, the study was not designed to isolate the causes of this behavior. Potential contributing factors include prompt structure, context length, model-specific reasoning behavior, and the interaction between retrieved content and tool outputs. Future work should investigate these factors through targeted experiments designed specifically to analyze retrieval-to-action failures.

Again, external validity is limited because the evaluation covers only two tasks from one project, with a restricted number of models and no repeated runs, while the standardized UI-controlled tools and simple retrieval pipeline support comparability, they may underrepresent models with stronger native tooling. This could cause some failures to be attributed to retrieval granularity rather than model capability.

9. Conclusions and Future Work

This paper presents an empirical study of the use of generative and repository-aware AI in student software development projects. Conducted within a university programming project, the study combines quantitative insights from student experience with qualitative observations of a locally deployed retrieval-augmented LLM assistant. The objective is to examine how AI tools can effectively support software engineering education and ways to improve their integration beyond general-purpose use. The results show that students perceive generative AI as valuable, particularly for text-heavy and structured tasks such as documentation, requirements formulation, and test generation. However, persistent challenges remain in code comprehension and implementation, where AI-generated outputs often fail to align with the project's structure and dependencies. This integration weakness, previously identified by Borghoff et al. [48], is partially addressed in our repository-aware setup through the use of project context and dedicated retrieval tools. Grounding model responses in actual repository artifacts enables more coherent, contextually aligned, and reproducible outputs.

The study shows that the advantages and limitations of AI assistance vary by phase of the software project. Tasks that rely primarily on text benefit most from generative support. However, diagram- and design-related artifacts, such as class and sequence diagrams, still lack adequate support. Therefore, the current emphasis on text-based work remains appropriate, while extending AI support to visual and conceptual design activities remains an important direction for future research. Qualitative analysis of the repository-aware assistant reveals recurring defects in grounding, tool use, repository integration,

and code-level implementation, including hallucinations and retrieval-to-action gaps. These findings highlight the need for more precise retrieval mechanisms, stronger prompt control, and more refined tool integration. Nevertheless, the repository-aware setup mitigates key context and integration limitations observed for general-purpose AI tools in the user study. It produces repository-aligned outputs and makes the generation process more traceable, thereby transforming the interaction from isolated prompt–response exchanges into a more integrated and inspectable workflow.

In summary, we provide exploratory evidence that, within the qualitative case study reported here, combining generative capabilities with repository-based retrieval can make generated outputs more contextually aligned and the generation process more traceable in software engineering education. Future work will focus on refining this approach to create a reflective, tutoring-oriented model that supports reasoning rather than replacement. Additionally, the contextual grounding will be extended to include visual and design artifacts. Our long-term goal for the programming project is to develop an AI system that acts as a tutor, assisting students with their programming assignments in an adaptive and pedagogically meaningful way. For an analytical review of barriers and operational requirements of adaptive learning, see [51].

Future work will expand the scope of repository awareness from solution-oriented assistance to contextual scaffolding. We plan to investigate tutor designs that use internally generated provisional solutions to derive hints and guiding questions, as well as designs that support exploratory problem solving without first forming such solutions. In both cases, repository grounding will help tutors refer students to relevant project artifacts and next steps without providing complete implementations. To evaluate this effect, dedicated learning-outcome studies are required. These studies should include pre/post assessments, suitable comparison groups, and task-based measures of coding accuracy, conceptual understanding, knowledge retention, and independent problem-solving ability. These studies should also distinguish between educational and efficiency gains to determine if AI-assisted tutoring improves learning or merely accelerates task completion.

These deployments will also require explicit governance of student interaction data, including rules for storage, access, retention, research use, consent, and use in model training.

Author Contributions: Conceptualization, U.M.B., M.M. and J.S.; methodology, U.M.B., M.M. and J.S.; software, M.M. and J.S.; validation, U.M.B., M.M. and J.S.; formal analysis, U.M.B., M.M. and J.S.; resources, J.S.; data curation, J.S.; writing—original draft preparation, U.M.B., M.M. and J.S.; writing—review and editing, U.M.B., M.M. and J.S.; visualization, U.M.B., M.M. and J.S.; supervision, U.M.B. and M.M.; project administration, J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: The study involving humans was approved by the Ethics Committee (IRB) at University of the Bundeswehr Munich under *Ethics Committee Approval—EK UniBw M 25-50*. The study was conducted in accordance with local legislation and institutional requirements for data protection.

Informed Consent Statement: A total of 49 students were enrolled in the course, of whom 38 voluntarily participated in the user study. All participants signed a written consent form agreeing to take part in the study. They also consented to the anonymized use of their collected data for research and publication purposes. Written consent was obtained from all participants prior to the start of the study.

Data Availability Statement: The original German task formulations and the datasets generated and/or analyzed during the current study are available from the authors.

Acknowledgments: This paper builds on and extends the work of Borghoff et al. [48], which is distributed under the Creative Commons Attribution International 4.0 License. Compared to our conference paper (Sections 3–5), this journal version introduces new Sections 6–8 that present a repository-aware, locally deployed LLM assistant, as well as a qualitative analysis of its behavior in a programming project setting. This extension is a direct follow-up on the results of our user study, addressing the identified limitations of general-purpose AI tools for code comprehension and integration within software development projects. A preprint of this article has been published on arXiv by Borghoff et al. [52]. We would like to thank our best students for their entertaining and creative game projects, which we continue to enjoy testing at the Institute's traditional Christmas event. We also wish to express our sincere gratitude to our students Vincent Bongiorno and Lucien Heller for their outstanding thesis work, whose ideas and technical contributions have influenced this paper. Their efforts have significantly enriched both the system design and the methodological approach presented here. Finally, we acknowledge the use of DeepL Translator, DeepL Write, Grammarly, and ChatGPT 5.4 for language improvement and translation support. These tools were used to refine the text and to translate selected sections from the original German version. The paper, however, remains an accurate representation of the authors' original research and contributions.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

GUI	Graphical User Interface
LLM	Large language model
RAG	Retrieval-augmented generation
TAM	Technology Acceptance Model

References

1. Mastropaolo, A.; Pascarella, L.; Guglielmi, E.; Ciniselli, M.; Scalabrino, S.; Oliveto, R.; Bavota, G. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *Proceedings of the 45th International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, 14–20 May 2023*; IEEE/ACM: Sydney, Australia, 2023; pp. 2149–2160. [CrossRef]
2. Crowston, K.; Bolici, F. Deskilling and upskilling with AI systems. *Inf. Res.* **2025**, *30*, 1009–1023. [CrossRef]
3. Macnamara, B.N.; Berber, I.; Çavuşoğlu, M.C.; Krupinski, E.A.; Nallapareddy, N.; Nelson, N.E.; Smith, P.J.; Wilson-Delfosse, A.L.; Ray, S. Does using artificial intelligence assistance accelerate skill decay and hinder skill development without performers' awareness? *Cogn. Res. Princ. Implic.* **2024**, *9*, 46. [CrossRef] [PubMed]
4. Rane, N.; Choudhary, S.; Rane, J. Education 4.0 and 5.0: Integrating artificial intelligence (AI) for personalized and adaptive learning. *SSRN Electron. J.* **2023**, *1*, 29–43. [CrossRef]
5. Nitzl, C.; Cyran, A.; Krstanovic, S.; Borghoff, U.M. The Use of Artificial Intelligence in Military Intelligence: An Experimental Investigation of Added Value in the Analysis Process. *Front. Hum. Dyn.* **2025**, *7*, 1540450. [CrossRef]
6. Russo, D. Navigating the Complexity of Generative AI Adoption in Software Engineering. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 135:1–135:50. [CrossRef]
7. Sengul, C.; Neykova, R.; Destefanis, G. Software engineering education in the era of conversational AI: Current trends and future directions. *Front. Artif. Intell.* **2024**, *7*, 1436350. [CrossRef] [PubMed]
8. Kokol, P. The Use of AI in Software Engineering: A Synthetic Knowledge Synthesis of the Recent Research Literature. *Information* **2024**, *15*, 354. [CrossRef]
9. Carleton, A.D.; Falessi, D.; Zhang, H.; Xia, X. Generative AI: Redefining the Future of Software Engineering. *IEEE Softw.* **2024**, *41*, 34–37. [CrossRef]
10. Yang, Y.; Xia, X.; Lo, D.; Grundy, J.C. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* **2022**, *54*, 206:1–206:73. [CrossRef]

11. Martinovic, B.; Rozic, R. Perceived Impact of AI-Based Tooling on Software Development Code Quality. *SN Comput. Sci.* **2025**, *6*, 63. [CrossRef]
12. Nascimento, N.; Alencar, P.; Cowan, D. Comparing Software Developers with ChatGPT: An Empirical Investigation. *arXiv* **2023**, arXiv:2305.11837. [CrossRef]
13. Piscitelli, A.; Costagliola, G.; Rosa, M.D.; Fuccella, V. Influence of Large Language Models on Programming Assignments—A user study. In *Proceedings of the 16th International Conference on Education Technology and Computers, ICETC 2024, Porto, Portugal, 18–21 September 2024*; ACM: New York, NY, USA, 2024; pp. 33–38. [CrossRef]
14. Waseem, M.; Das, T.; Ahmad, A.; Liang, P.; Fahmideh, M.; Mikkonen, T. ChatGPT as a Software Development Bot: A Project-Based Study. In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2024, Angers, France, 28–29 April 2024*; Kaindl, H., Mannion, M., Maciaszek, L.A., Eds.; SciTePress: Setúbal, Portugal, 2024; pp. 406–413. [CrossRef]
15. Bhandari, K.; Kumar, K.; Sangal, A.L. Artificial Intelligence in Software Engineering: Perspectives and Challenges. In *Proceedings of the 2023 Third International Conference on Secure Cyber Computing and Communication (ICSCCC)*; IEEE: New York, NY, USA, 2023; pp. 133–137. [CrossRef]
16. Borghoff, U.M.; Bottoni, P.; Pareschi, R. An Organizational Theory for Multi-Agent Interactions Integrating Human Agents, LLMs, and Specialized AI. *Discov. Comput.* **2025**, *28*, 138. [CrossRef]
17. Borghoff, U.M.; Bottoni, P.; Pareschi, R. Beyond Prompt Chaining: The TB-CSPN Architecture for Agentic AI. *Future Internet* **2025**, *17*, 363. [CrossRef]
18. Daun, M.; Brings, J. How ChatGPT Will Change Software Engineering Education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2023, Turku, Finland, 7–12 July 2023*; Laakso, M., Monga, M., Simon, Sheard, J., Eds.; ACM: New York, NY, USA, 2023; pp. 110–116. [CrossRef]
19. Borghoff, U.M.; Minas, M.; Mönch, K. Automatic Program Assessment, Grading and Code Generation: Possible AI-Support in a Software Development Course. In *Proceedings of the Artificial Intelligence and Soft Computing—23rd International Conference, ICAISC 2024, Zakopane, Poland, 16–20 June 2024*; Lecture Notes in Artificial Intelligence; Rutkowski, L., Scherer, R., Korytkowski, M., Pedrycz, W., Tadeusiewicz, R., Zurada, J.M., Eds.; Springer: Cham, Switzerland, 2024; Volume 15166, pp. 39–51. [CrossRef]
20. Kazemitabaar, M.; Chow, J.; Ma, C.K.T.; Ericson, B.J.; Weintrop, D.; Grossman, T. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, 23–28 April 2023*; Schmidt, A., Väänänen, K., Goyal, T., Kristensson, P.O., Peters, A., Mueller, S., Williamson, J.R., Wilson, M.L., Eds.; ACM: New York, NY, USA, 2023; pp. 455:1–455:23. [CrossRef]
21. Ala-Mutka, K. A Survey of Automated Assessment Approaches for Programming Assignments. *Comput. Sci. Educ.* **2005**, *15*, 83–102. [CrossRef]
22. Douce, C.; Livingstone, D.; Orwell, J. Automatic test-based assessment of programming: A review. *ACM J. Educ. Resour. Comput.* **2005**, *5*, 4. [CrossRef]
23. Krusche, S.; Seitz, A. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education, SIGCSE 2018, Baltimore, MD, USA, 21–24 February 2018*; Barnes, T., Garcia, D.D., Hawthorne, E.K., Pérez-Quiñones, M.A., Eds.; ACM: New York, NY, USA, 2018; pp. 284–289. [CrossRef]
24. Srikant, S.; Aggarwal, V. Automatic Grading of Computer Programs: A Machine Learning Approach. In *Proceedings of the 2013 12th International Conference on Machine Learning and Applications, Miami, FL, USA, 4–7 December 2013*; IEEE: New York, NY, USA, 2013; Volume 1, pp. 85–92. [CrossRef]
25. Chrysafiadi, K.; Virvou, M.; Tsihrintzis, G.A. A fuzzy-based mechanism for automatic personalized assessment in an e-learning system for computer programming. *Intell. Decis. Technol.* **2022**, *16*, 699–714. [CrossRef]
26. Hidalgo-Suarez, C.G.; Bucheli, V.A.; Ordoñez, H. Automatic Assessment of Learning Outcomes as a New Paradigm in Teaching a Programming Course: Engineering in Society 5.0. *Rev. Iberoam. Tecnol. Aprendiz.* **2022**, *17*, 379–385. [CrossRef]
27. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A.D.; et al. Competition-level code generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [CrossRef] [PubMed]
28. Chowdhery, A.; Narang, S.; Devlin, J.; Bosma, M.; Mishra, G.; Roberts, A.; Barham, P.; Chung, H.W.; Sutton, C.; Gehrmann, S.; et al. PaLM: Scaling Language Modeling with Pathways. *J. Mach. Learn. Res.* **2023**, *24*, 240:1–240:113.
29. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16–20 November 2020*; Cohn, T., He, Y., Liu, Y., Eds.; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; Volume EMNLP 2020, pp. 1536–1547. [CrossRef]
30. Finnie-Ansley, J.; Denny, P.; Becker, B.A.; Luxton-Reilly, A.; Prather, J. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proceedings of the ACE '22: Australasian Computing Education Conference, Virtual Event, Australia, 14–18 February 2022*; Sheard, J., Denny, P., Eds.; ACM: New York, NY, USA, 2022; pp. 10–19. [CrossRef]

31. Gao, Y.; Xiong, Y.; Gao, X.; Jia, K.; Pan, J.; Bi, Y.; Dai, Y.; Sun, J.; Wang, M.; Wang, H. Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv* **2024**, arXiv:2312.10997. [CrossRef]
32. Ovadia, O.; Brief, M.; Mishaeli, M.; Elisha, O. Fine-Tuning or Retrieval? Comparing Knowledge Injection in LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, 12–16 November 2024*; Al-Onaizan, Y., Bansal, M., Chen, Y., Eds.; Association for Computational Linguistics: Stroudsburg, PA, USA, 2024; pp. 237–250. [CrossRef]
33. Huang, Y.; Huang, J. A Survey on Retrieval-Augmented Text Generation for Large Language Models. *arXiv* **2024**, arXiv:2404.10981. [CrossRef]
34. Ni, B.; Liu, Z.; Wang, L.; Lei, Y.; Zhao, Y.; Cheng, X.; Zeng, Q.; Dong, L.; Xia, Y.; Kenthapadi, K.; et al. Towards Trustworthy Retrieval Augmented Generation for Large Language Models: A Survey. *arXiv* **2025**, arXiv:2502.06872. [CrossRef]
35. Manakina, O.; Lung, C. Designing Reusable LLM-Enhanced Assignments: A Quality-Oriented Framework for Software Engineering Education. In *Proceedings of the 49th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2025, Toronto, ON, Canada, 8–11 July 2025*; Shahriar, H., Alam, K., Ohsaki, H., Cimato, S., Capretz, M., Ahmed, S., Ahamed, S., Majumder, A., Haque, M., Yoshihisa, T., et al., Eds.; IEEE: New York, NY, USA, 2025; pp. 2212–2217. [CrossRef]
36. Raman, A.; Kumar, V. Programming Pedagogy and Assessment in the Era of AI/ML: A Position Paper. In *Proceedings of the COMPUTE 2022, Jaipur, India, 9–11 November 2022*; Choppella, V., Karkare, A., Babu, C., Chimalakonda, S., Eds.; ACM: New York, NY, USA, 2022; pp. 29–34. [CrossRef]
37. Izu, C.; Schulte, C.; Aggarwal, A.; Cutts, Q.I.; Duran, R.; Gutica, M.; Heinemann, B.; Kraemer, E.T.; Lonati, V.; Mirolo, C.; et al. Fostering Program Comprehension in Novice Programmers—Learning Activities and Learning Trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR 2019, Aberdeen, Scotland, UK, 15–17 July 2019*; Scharlau, B., McDermott, R., Pears, A., Sabin, M., Eds.; ACM: New York, NY, USA, 2019; pp. 27–52. [CrossRef]
38. Moroz, E.A.; Grizkevich, V.O.; Novozhilov, I.M. The Potential of Artificial Intelligence as a Method of Software Developer's Productivity Improvement. In *Proceedings of the 2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*; IEEE: New York, NY, USA, 2022; pp. 386–390. [CrossRef]
39. Choudhuri, R.; Liu, D.; Steinmacher, I.; Gerosa, M.; Sarma, A. How Far Are We? The Triumphs and Trials of Generative AI in Learning Software Engineering. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*; IEEE: New York, NY, USA, 2024; pp. 184:1–184:13. [CrossRef]
40. Pudari, R.; Ernst, N.A. From Copilot to Pilot: Towards AI Supported Software Development. *arXiv* **2023**, arXiv:2303.04142. [CrossRef]
41. AlOmar, E.A. Nurturing Code Quality: Leveraging Static Analysis and Large Language Models for Software Quality in Education. *ACM Trans. Comput. Educ.* **2025**, *25*, 1–36. [CrossRef]
42. Li, S.; Liu, J.; Dong, Q. Generative artificial intelligence-supported programming education: Effects on learning performance, self-efficacy and processes. *Australas. J. Educ. Technol.* **2025**, *41*, 1–25. [CrossRef]
43. Garousi, V.; Jafarov, Z.; Movsumova, A.; Namazov, A.; Mirzayev, H. Encouraging Students' Responsible Use of GenAI in Software Engineering Education: A Causal Model and Two Institutional Applications. *arXiv* **2025**, arXiv:2506.00682. [CrossRef]
44. Borghoff, U.M.; Minas, M.; Mönch, K. Using Automatic Program Assessment in a Software Development Project Course. In *Proceedings of the 5th European Conference on Software Engineering Education, ECSEE 2023, Seeon/Bavaria, Germany, 19–21 June 2023*; Mottok, J., Ed.; ACM: New York, NY, USA, 2023; pp. 22–30. [CrossRef]
45. Rapaka, A.; Dharmadhikari, S.C.; Kasat, K.; Mohan, C.R.; Chouhan, K.; Gupta, M. Revolutionizing learning—A journey into educational games with immersive and AI technologies. *Entertain. Comput.* **2025**, *52*, 100809. [CrossRef]
46. Davis, F. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Q.* **1989**, *13*, 319–340. [CrossRef] [PubMed]
47. Krasner, G.E.; Pope, S.T. A cookbook for using model-view-controller user interface paradigmin smalltalk-80. *J. Object-Oriented Program.* **1988**, *1*, 26–49.
48. Borghoff, U.M.; Minas, M.; Schopp, J. Generative AI in Student Software Development Projects: A User Study on Experiences and Self-Assessment. In *Proceedings of the 6th European Conference on Software Engineering Education, ECSEE 2025, Seeon/Bavaria, Germany, 2–4 June 2025*; Mottok, J., Hagel, G., Eds.; ACM: New York, NY, USA, 2025; pp. 161–170. [CrossRef]
49. Sergeyuk, A.; Golubev, Y.; Bryksin, T.; Ahmed, I. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Inf. Softw. Technol.* **2025**, *178*, 107610. [CrossRef]
50. Cornide-Reyes, H.; Monsalves, D.; Durán, E.; Silva-Aravena, F.; Morales, J. Generative Artificial Intelligence in Agile Software Development Processes: A Literature Review Focused on User eXperience. In *Proceedings of the Social Computing and Social Media*; Coman, A., Vasilache, S., Eds.; Springer: Cham, Switzerland, 2025; pp. 228–246. [CrossRef]

51. Dobrovsky, A.; Hofmann, M.; Borghoff, U.M. The Adoption Gap in Adaptive Learning Path Generation: An Analytical Review of Barriers and Operational Requirements. *Front. Comput. Sci.* **2026**, *8*, *accepted*.
52. Borghoff, U.M.; Minas, M.; Schopp, J. Learning to Code with Context: A Study-Based Approach. *arXiv* **2025**, arXiv:2512.05242. [CrossRef]