AI-Driven Software Development: A New Course Concept and Assessment Model for the Era of Large Language Models

Benedikt Fein
University of Passau
Passau, Germany
benedikt.fein@uni-passau.de

Gordon Fraser
University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

Steffen Herbold
University of Passau
Passau, Germany
steffen.herbold@uni-passau.de

Abstract

The proliferation of Large Language Models (LLMs) is transforming the way software is developed, and software engineering education needs to adapt to this new reality. Traditional grading methods that assess code functionality are undermined by AI-generated code. Meanwhile, students are already using these tools and require formal training on their potential and dangers throughout the software lifecycle. We address this with a new course on *AI-Driven Software Development*. After an initial scaffolded phase covering software development phases and corresponding AI tools, students independently build a complex software system using AI techniques. To evaluate student learning rather than AI capabilities, our grading is based on the final code, the development process, and a reflective report. In our first implementation of this course with 68 students they not only successfully built non-trivial software systems with AI support—our grading scheme also reveals deeper reflections on when and how to effectively use AI tools, which also provide valuable insights informing future courses on AI-driven software development as well as AI tool development in general.

CCS Concepts

• **Social and professional topics** → **Software engineering education**; • **Computing methodologies** → **Artificial intelligence**.

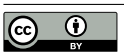
Keywords

CS Education, Large Language Models

ACM Reference Format:
Benedikt Fein, Gordon Fraser, and Steffen Herbold. 2026. AI-Driven Software Development: A New Course Concept and Assessment Model for the Era of Large Language Models. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEET '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3786580.3786962

1 Introduction

The world of software development is in a state of change, caused by the recent proliferation of Large Language Models (LLMs) and development tools that use them to automate even tasks previously unthinkable. As professional software development is adapting to this new reality by actively integrating LLMs and AI-based tools into the workflow [11, 22, 30], there is a growing awareness that software engineering education also needs to adapt [3, 12].



ICSE-SEET '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2423-7/2026/04
https://doi.org/10.1145/3786580.3786962

Students already make heavy use of LLMs, for example to solve coding exercises [1, 4]. This has primarily been considered as an issue of sabotaging potential learning outcomes and complicating grading of student coursework. However, given that graduates will enter a workforce where proficiency with AI tools is the new standard, it becomes important that students are explicitly educated in the different approaches to AI tools and how these can be used to enhance different phases and tasks of software development. They furthermore need to understand the downsides and limitations of AI tools, and how to overcome them. In other words, we need to educate our students not only in classical software development, but in the new paradigm of *AI-Driven Software Development*.

Adapting software engineering curricula presents several challenges. Educators must decide what content to teach, covering both traditional software development phases and the various AI tools that support them. The pedagogical approach must remain hands-on to be effective, while also navigating the rapid evolution and potential costs of state-of-the-art AI techniques. Most critically, traditional assessment methods, which often rely on executing automated tests on student code, are no longer reliable. A central problem becomes how to grade the student's learning and critical thinking, not the generative capabilities of the LLMs they use.

In this paper, we introduce a new course concept on *AI-Driven Software Development* that addresses these issues: It starts with an initial scaffolded phase where students progress through a classical sequence of software development phases such as requirements engineering, prototyping, coding, or testing, all gradually contributing to them building an app. Each of these steps makes use of a different approach to integrating AI tools, such as basic chat interactions with LLMs, IDE-plugins, AI-agents, and API-usage. Following this initial phase, students are tasked to independently implement a more complex software system, thereby applying the AI-based techniques they learned in the first phase, and covering all aspects of the software development process. Halfway through this second phase, the requirements are changed, forcing students to engage with aspects of maintenance and evolution such as code health and technical debt, but also challenging the capabilities of the AI tools.

To assess student learning rather than the capabilities of LLMs, the grading is based not only on submitted code. The students also need to submit a written report in which they justify their design choices and architecture, describe which AI tools and LLMs they used, and provide reflections on where these were helpful and where not so much. The overall assessment then considers the report, the software project, and details of the development process exposed by the issue management and version control systems. To evaluate the feasibility of this course, we conducted it at the University of Passau in the summer semester of 2025 with 68 students participating.

In detail, the contributions of this paper are as follows:

- We introduce an *AI-Driven Software Development* course that covers important aspects of software development as well as the possibilities and techniques to support these offered by AI tools.
- We provide detailed grading rubrics that allow effective assessment of student learning, even in the light of AI-generated components and artefacts.
- We evaluate the course concept at master's level with 68 students, providing insights into learnings and experiences.
- We discuss our experiences and insights, for example on how to cope with budgetary challenges of having students use AI tools.

We find that the students largely succeeded in building non-trivial software systems with AI support but also developed a deeper understanding of when and how to use AI tools effectively. This learning was captured through our grading scheme, and their reflections offer valuable insights for designing and improving future software development courses and AI tools.

2 Background

2.1 AI for Software Engineering

The introduction of LLMs and their applicability to different software artefacts is profoundly affecting the software engineering landscape. Current research suggests LLMs are particularly well-suited for tasks that demand an understanding of syntax, such as code summarization and code repair [31], with less satisfactory performance on tasks that require comprehension of code semantics, such as code generation and vulnerability detection. Research is nevertheless exploring the application of LLMs across the entire software development lifecycle [6], with the majority of studies focusing on development, maintenance, and quality assurance [10]. While researchers are busy exploring different applications of LLMs and methods to assess their performance, software development in practice is simultaneously undergoing a transformation. Despite early claims that LLMs would eventually even completely replace software developers, the consensus currently seems to be that LLMs augment the capabilities of software developers rather than replacing them [25], and human oversight remains crucial [9] as part of an AI-driven approach to software development.

Software developers have a range of possibilities available for integrating LLMs into their workflow, ranging from basic chats to sophisticated IDE plugins. Industrial practice seems to mostly focus on LLM agents [26], i.e., intelligent entities capable of perceiving software artefacts, reasoning using LLMs, and taking actions semi-automatically. Examples include both commercial products integrated into the IDE (e.g., *Cursor*, *JetBrains Junie*, *GitHub Copilot*, *Windsurf*), or IDE-agnostic command-line tools (e.g., *Claude Code*) but also open-source projects (e.g., *aider.chat*, *OpenHands*).[1]

2.2 AI in Software Engineering Education

The transformation of software engineering practice has lead to calls for also adapting software engineering education [12, 18, 24, 25, 32]. A large share of existing work on LLMs in computer science education [21] focuses on Python and Java programming at undergraduate level, rather than more advanced software engineering involving the development of complex software systems and covering the entire software lifecycle. Furthermore, LLMs are often applied for improving teaching of traditional software engineering tasks such as requirements elicitation, UML diagrams, or prototyping [20], or for supporting teachers with the course conduction by generating exercises [23] and automating assessment [2, 8]. It has been shown that students are already using LLMs [28], even without explicit education, which is an issue because inadequate preparation and unrealistic expectations of students about the capabilities of LLMs can even be counterproductive [16].

One way to address this issue lies in observing how students use LLMs and generalising from that [17]. However, there nevertheless is a misalignment as aspects of software engineering en-vogue in LLM-based research [10], such as automated test generation or bug detection/fixing, seldom appear in education-related research context. Even though AI tools have individually been considered in terms of their capabilities to support education [29], there is a gap on how to integrate the AI tools into the curriculum so that students learn to use them by understanding the foundations behind them and their limitations [24]. Consequently, there is a need for a more holistic education approach [12, 14, 15] going beyond individual technical skills, emphasising the importance of adapting and evolving to remain in sync with rapid advancements in AI and LLMs, and adapting expected learning outcomes to a shift in required skill levels for AI-supported software development [32]. This is precisely the aim of the course described in this paper.

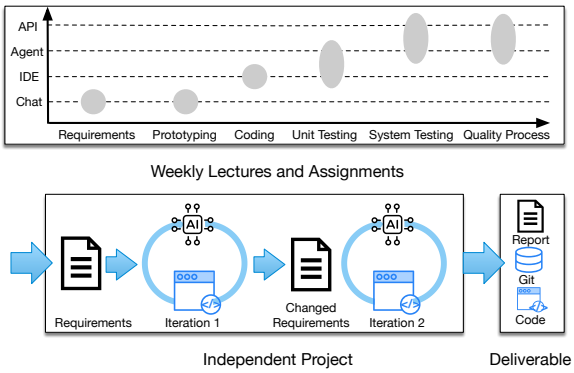


Figure 1: Overview of the design of our course.

3 Course Design

Figure 1 provides an overview of the AI-Driven Software Development course: It consists of an initial scaffolded phase in which there are weekly lectures and exercise sessions (Section 3.1) covering both, different phases of software development, and different types of AI tools. In the second phase the students independently develop a software project over two iterations with changing requirements (Section 3.2). Finally, they have to deliver the code, repository, and a written report (Section 3.3).

[1] Cursor: https://cursor.com/, Junie: https://www.jetbrains.com/junie/, Copilot: https://github.com/features/copilot, Windsurf: https://windsurf.com/, Claude Code: https://claudecode.io/, aider: https://aider.chat, OpenHands: https://www.all-hands.dev/; accessed 2025-09-25

3.1 Weekly Assignments

The course starts with a scaffolded phase of six interactive lectures each introducing one step in a classical sequence of software development: requirements engineering, prototyping, coding, unit and integration testing, system testing, and an overall quality-driven process (e.g., continuous integration with static analysis tools and automated tests, merge requests with code reviews). In each lecture (90 minutes), we first introduce the tasks and outcomes and how they can be achieved using non-AI-based methods, to ensure students are familiar with core software engineering concepts.

3.1.1 Lectures. Each lecture then focuses on examples for AI-based tools suitable for the respective task and presents strategies on how to use the tool by highlighting potential challenges and how to overcome them. For the presented tools, we focus mainly on ones that are available for free for all students, either by being open-source or by offering a special student discount. For example, when first delivering the course all students had free access via the *GitHub* students pack to *GitHub Copilot* both in the *GitHub* web interface and as plugin to their IDE. To support the use of open-source tools, e.g., the *aider* agent, we host a local instance of *Ollama* for API-based access to self-hosted LLMs like *Mistral*, *Devstral*, *Llama 3.1*, *CodeLlama*, and *Gemma 3*. Nevertheless, the students are free to also use other tools not presented in the lectures, including commercial ones like *Cursor* in case they already have a subscription for them.

More specifically, in the first lecture we present a classical design thinking approach to collect requirements and demonstrate how it can be extended by prompting an LLM via its web-chat to suggest additional ideas or to formulate requirements in user story format. We use the demonstration to highlight that the LLMs might sometimes generate generic, too complex, or even nonsensical requirements. Similar demonstrations of limitations continue throughout all lectures to highlight that while the AI can support many development tasks, the human developer ultimately needs to retain both control over the tools and an understanding of the developed system to be able to remediate issues in AI-generated work.

The design thinking approach continues for the prototype, where again the LLM web-chat can be manually prompted to generate user interface mockups either as images or as mostly static HTML files. By prompting the LLM with more technical information about the planned system, it can also be used to suggest data entity models or suitable API endpoints for the interaction between client and server, to extend the prototype into a system skeleton that can later be filled with the implementation of the actual application logic. For this coding phase, the students are usually already familiar with the AI-based autocompletion and prompting tools built into their IDE, so little demonstration, e.g., using *Copilot*, is required. Instead, we focus on possible issues with a pure 'vibe coding' approach, e.g., the introduction of maintenance burden, subtle bugs, or security issues [7, 19] in the generated code due to even the original developer not fully understanding it. We then present prompt strategies to mitigate such issues [27].

To further strengthen the confidence in the quality of the generated code, we next introduce various testing concepts, focusing on automated unit, integration, and mutation testing first. While the LLM context is sufficient to produce unit tests only from the relevant code, it may need additional prompting with the intended behaviour for larger integration tests. Using agentic tools like *aider* that automatically re-run test suites after changes results in a higher confidence in the quality of the generated code, even though generated tests for faulty code might make the wrong assertions. To simulate the user interactions as part of end-to-end system tests, we give an introduction to the *Playwright* library and its Java API.[2] Demonstrating the generation of such tests using *aider* highlights the challenge of context management. By introducing a common abstract base class for tests, the AI tool is guided towards calling reusable helper methods. Furthermore, manually adding relevant HTML templates to the context enables the LLM to use correct identifiers for the interactive user interface elements in the code. Originally, we also intended to task students with an extension for the agent that provides a custom tool hook which can be called by the LLM to specifically request suitable user interface element IDs. However, due to time constraints, we could not realise it and instead plan to revisit this idea in the next course iteration.

Finally, for the quality process lecture, we highlight that, while many tasks can be automated by AI, ultimately the developer still has to retain responsibility for the produced system and has to ensure consistency throughout the development process, e.g., consistency between user stories and user requirements, implementation and user stories, or expected and tested behaviour. We present automatic code review tools like *Copilot* as integrated into the *GitHub* website and *CodeRabbit*.[3] Nevertheless, we recommend students to set up an automatic continuous integration (CI) pipeline that runs code formatters, linters, and the test suite so that core consistency and quality checks are always enforced even without AI support.

3.1.2 Practical Application of the Concepts. To put the presented theoretical concepts into practice, the students are tasked to iteratively develop a small Todo-notes app during this initial course phase as part of weekly assignments. We require them to implement the application with a JVM-based (e.g., using Java or Kotlin) server connected to a database and providing the API for a user interface developed using Java- or TypeScript running in the browser. These technical requirements are a sufficient restriction to allow us to create the lectures around this common baseline, but still enable the students to choose familiar frameworks and libraries for their implementation. We also do not require a specific set of features for this application, but instead let them develop their own in the initial requirements engineering phase. Giving the students this freedom allows them to use the development of the Todo-notes app to collect experience with their preferred tools in preparation for the final project in the second part of the course.

3.2 Final Project

The second part of the course consists of an individual assignment spanning five weeks (e.g., in our case, end of June to end of July). The students are tasked to implement a calendar application with the same basic technical requirements (i.e., JVM server with browser-based user interface and freedom to choose the exact libraries) as the Todo app from the initial weekly assignments. However, for this project we specify some initial functional requirements (event

[2] https://playwright.dev/java/, accessed 2025-09-28
[3] https://www.coderabbit.ai/, accessed 2025-09-28

creation/update/deletion, user management, reminders for events, recurring events, and tagging events with user-defined tags) which the students should extend into full user stories with corresponding acceptance criteria before integrating them into their application.

To simulate a real-world project with changing requirements, we publish additional requirements two weeks after the initial ones. The addition of new requirements also requires the AI tools to be able to support maintenance and evolution, rather than only generating the code of an initial prototype. We require the students to design three new features they would personally find useful and then implement them. Examples for commonly implemented features are a switch between light and dark mode for the user interface or being able to search for events. We also request two more specific requirements that focus on features the development of which might contain subtle challenges for AI tools: An event sharing feature should allow users to share events with others. Depending on the chosen implementation, a sharing link or code has to be handled securely such that only the intended recipient can view or edit only this specific event. As a second feature that introduces many possible edge-cases, users should be able to set a preferred time zone in their user profile such that the calendar view is displaying own events in the correct time and that shared events are shown according to each user's preferred time zone.

The students are not required to follow the exact iterative system development process as in the initial weekly sessions, but they are free to adopt agile-like methods. However, to ensure that they not only focus on the coding but also use the AI tools for other development phases, we require them to create issues in their *GitHub* project with descriptions in the user story format and task them to develop the application iteratively by small merge requests or commits that implement one feature or refactoring at a time. Furthermore, they should add unit and integration tests to both server and client of the system with at least 90 % statement coverage and develop automatic system tests that simulate the interactive interaction via the browser with the full application. The students should add a CI pipeline to their project that verifies the feature integrity after each commit by running automatic tests, automatic linters (e.g., *CheckStyle*, *ESLint*) and static analysis tools to ensure that generated code follows the project's code style guide.

3.3 Final Project Report

To encourage reflection on system design and AI tool usage, the students are required to submit a project report in addition to their implementation and repository. This report should link relevant issues, merge requests, or commits when explaining features or AI tool usage. The suggested structure of the report was as follows.

System architecture and design decisions. The students are asked to give a brief overview over the high-level design decision they made while designing and implementing the application requirements. For example, they could explain technical decisions (e.g., system architecture, library choices), or in case there were requirements for which the specification allowed for multiple different approaches, explain why they selected their chosen alternative.

Design decisions for custom features. In this section the students should give a brief overview over their three chosen custom features developed during the second half of the final project. They should describe each feature and explain the design choices they made when integrating it into their application.

Used LLMs and AI-based tools. This section should contain a summary of the LLMs and AI-based tools they used and how they used them. They should explain for which development scenarios they used which tools. When tools could be used with alternative LLMs (e.g., *GitHub Copilot* supports *GPT* and *Claude* models), they should explain which LLMs they configured in the tool settings.

Where did the AI tools work well? This section should go into more detail over the used tools by demonstrating examples for LLM/tool usage where the tool achieved the intended goal.

Where did the AI tools not work as intended? Similar to the previous section, the students should explain cases where the chosen LLMs/tools did not work as well as expected. They should describe how they adapted their usage of the respective LLM/tool to get more useful responses and if that led to the expected improvements.

Missing AI tools (optional). Since the available AI tools do not cover all possible software engineering and development scenarios, we ask the students to explain if they were missing tool support for some tasks when working on the final project and if so, what an envisioned future tool should be able to do to provide the missing support. They can also mention tools that already exist and they would have liked to use, but were not available to do so, for example due to a complicated setup or the tool's pricing model.

3.4 Grading

The final grade of the course is determined by summing the points for weekly assignments, the final project implementation, the development process followed while implementing the final project, and the project report. All submissions are collected via private repositories on *GitHub* and include the code itself as well as the issues and pull requests created by the students.

For the six initial weekly assignments (cf. Section 3.1), we assign between zero and two points (not at all, partially, or fully completed) each. Similarly, we require the final project to be executable on our machines to verify the feature implementations and also assign between zero and two points per requirement. The grading for these two categories is intentionally coarse and limited in points, since the work of the students and the AI is directly mixed.

The main focus of the grading therefore is on the development process and the final report, since these are not directly the result of the AI, but instead show whether the students used a suitable AI-based process and then explicitly reflected on how they used the tools. To ensure a consistent grading process, we developed grading rubrics for both categories. We subcategorised both and assign between zero and four points for each subcategory.

To grade the development process, we split it into the five subcategories shown in Table 1. For issues and pull requests (or commits) we evaluate that the requirements are specified clearly and are implemented iteratively. The development in self-contained steps is important because it allows both us and students to revisit the development process and allows the students to refer to the commit history in the report when describing how they used the AI tools. The CI pipeline should run all checks as initially specified (cf. Section 3.2). Finally, for both unit and system tests the evaluation

Table 1: Grading rubric for the development process.

Score	Issues	Pull Requests (PRs)/Commits	CI Pipeline	Unit Tests	System Tests
0	No issues	Few large commits implementing the whole project without reference to relevant issues.	Missing	Missing	Missing
1	Issues without description, so no user stories or acceptance criteria.	Some PRs/Commits implementing multiple linked issues at a time.	Compiles the application.	Tests only in server or only in client.	Few system tests cover the happy path of only some features. The server and/or database is mocked.
2	Issues with user stories only for the requested high-level features.	PRs for major features. PRs usually implement one thing (i.e., at most one issue referenced).	Runs unit tests for client and server.	Unit tests for both server and client.	Few system tests cover the happy path of some features.
3	Issues with user stories and acceptance criteria only for the requested high-level features.	PRs for features referencing relevant issues. PRs usually implement one thing.	Runs unit tests and (system tests XOR linters) for client and server.	$\geq$ 80 % coverage for both server and client.	The happy path of all important system features is tested end-to-end (i.e., client connects to actual server and database).
4	Issues not only for the requested high-level features, but also for, e.g., bugfixes or additional smaller features that came up during development.	PRs for features, refactorings, and bugfixes referencing relevant issues. PRs usually implement one thing.	Runs linters, unit tests for both server and client, and system tests.	$\geq$ 90 % coverage for both server and client.	Both the happy path and relevant edge- and failure-cases are tested for all important system features.

Table 2: Grading rubric for the final project report. Missing sections received a score of zero.

Score	System architecture & design	Custom Features	Used LLMs and AI-based tools	Where did AI tools work well?	Where did they not work?
1	Chapter with high level content, but without explanations or design decisions.	Features are incomplete, poorly integrated, or only minimally described.	Barely mentions tool usage or describes only one scenario with little detail or clarity.	Barely identifies any concrete benefits, or examples are unclear or not convincingly linked to tool performance.	Vague or minimal reference to tool failure.
2	Generic explanations, some design decision, no alternatives discussed.	Custom features are present but described in vague or shallow terms, or integration into the core application is not fully clear.	Mentions a few tools but offers only general or vague descriptions. Some tool-task mappings may be unclear or superficial.	Mentions some areas where AI tools worked, but examples may be vague, generic, or lacking context.	Mentions at least one issue, but with limited detail or unclear response strategies.
3	Solid explanation of system architecture and design decisions, alternatives only shortly mentioned.	Required number of custom features are implemented and described reasonably well. Some design choices are explained, though depth or clarity may vary.	Identifies and describes relevant tools used for several tasks. Provides explanations of how and why the tools were used, but lacks details.	Gives a few solid examples with reasonable explanations. May be less in-depth or slightly generalized in places..	Identifies a few meaningful issues and explains how they were handled, though adaptations may be more general or less reflective.
4	Solid explanation of system architecture and design decisions, alternatives discussed in depth.	Meets or exceeds the required number of functional custom features. Features are clearly described, technically sound, and well-integrated into the application. Shows thoughtful design decisions and links to relevant issues/pull requests.	Each tool's use is specifically described, including how it was prompted/integrated, what it contributed, and why it was appropriate for that context.	Provides clear, specific examples of scenarios or development phases where AI tools excelled. Describes why the tools worked well in each case.	Provides specific, well-explained examples where AI tools failed or underperformed. Describes how they recognized the issue and how they adapted their approach.

mainly focuses on presence and sufficient coverage. The high coverage requirement as communicated as part of the final project description can unlikely be achieved by 'vibe coding', but instead requires conscious use of AI tools to obtain tests for edge-cases.

The rubric to grade the final project report (Table 2) is categorised according to the report subsections (cf. Section 3.3). Optional report subsections are not taken into account for the grade. For each category, the students achieve higher marks by showing a thorough reflection on *why* and *how* they made their design decisions and used the AI tools, rather than only describing *what* they used.

4 Evaluation

We conducted the course during the summer semester of 2025 and collected data in order to answer the following research questions:

RQ1 What are the statistical properties of the produced software?
RQ2 Which AI-based tools and LLMs did the students use?
RQ3 How were the students supported by AI tools across different development tasks?

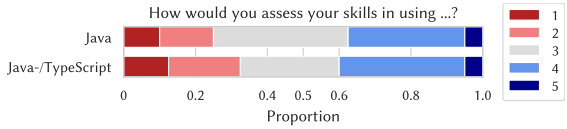


Figure 2: Self-reported programming skills of the course participants (1=Beginner, 5=Expert).

4.1 Experimental Setup

4.1.1 Dataset. In the summer semester of 2025 68 students (2 Bachelor, 66 Master) participated in the course, and 63 completed it.

After the course, we conducted an anonymous online survey using the questions shown in Table 3 among the course participants. The participation was optional. Nevertheless, 40 students participated in the survey, i.e., 63 % of the students who completed the course. As Fig. 2 shows, most survey participants were familiar with the programming languages used in the course. 30 % of the students had between one and three years of software development experience, an additional 55 % more than three years. 55 % stated to have a job besides their studies that involves software development.

Table 3: Post-course survey questions.

How many years of software development experience do you have?	Single choice
How would you assess your skills in using Java?	1–5
How would you assess your skills in using Javascript or Typescript?	1–5
Do you have a job involving software development besides your studies?	Yes/No
For each of *requirements*, *protoyping*, *coding*, *unit testing*, *system testing*, *reviewing*:	
Using AI support helps me to ⟨*perform activity X better*⟩	5-point Likert
Using AI support increases the quality of ⟨*result of activity X*⟩	5-point Likert
I could have ⟨*performed activity X*⟩ without AI support	5-point Likert
I believe I am now better at ⟨*activity X*⟩, even without AI support	5-point Likert
Only for *reviewing*:	
The AI tools followed my project and coding guidelines	5-point Likert
If you used AI support for anything else not covered by this survey, what was it and did it help you?	Free text
Which LLM-based Tools did you use?	Multiple choice
Which LLMs did you use?	Multiple choice

4.1.2 RQ1. To answer this research question, we consider the Git repositories and extracted statistics about the code for both the server and client components.[4] Additionally, we counted the number of tests implemented by the students. We do not consider the test coverage, since this was an explicitly graded metric (cf. Section 3.4) and would therefore be biased towards a higher coverage.

To estimate code quality, we also ran the *PMD* static code analysis tool[5] on the server Java code. We used the provided quickstart configuration,[6] but excluded the *MethodNamingConventions* rule, as many tests followed a 'when… then… should…' or 'test_…' naming scheme with underscores as separators. Such names violate common Java code style guidelines using camel-casing, but nevertheless are often accepted in project-specific code style guidelines.

More than half of the students implemented all 16 required features in their application (mean = 14.5, σ = 2.48). Therefore, we can be reasonably certain that the differences in code metrics likely stem from different implementation design decisions rather than being caused by missing features.

4.1.3 RQ2. As part of the survey, we also collected the used LLMs and AI-based tools (cf. Table 3). The answer options for both questions included both LLMs/tools presented in the lecture and other popular ones. The students could also add other used LLMs/tools in a free-form response field. From the final project reports we additionally obtained more detailed explanations which models and tools were used for which software development tasks and where they worked well or did not work as expected.

Please note that the anonymous survey did not allow us to relate the used LLMs/tools to the produced code or grades, as we cannot identify which student submitted which survey response.

4.1.4 RQ3. To estimate which software development phases were difficult for the students, we use the graded results based on the development process rubric (Table 1). The 'Issues' category corresponds to the requirements engineering phase and the 'Pull Requests' and 'CI pipeline' categories are aspects of the quality process phase. Both the unit and system test categories directly correspond to the respective development tasks. The coding aspect of the development is not represented in the rubric, but instead based on the feature-completeness of the final implementation (cf. Section 3.4).

Using the scores achieved according to the report grading rubric (Table 2), we learn whether students actually reflected on their AI-tool use. The grading rubric is designed such that a higher score requires detailed examples and strategies for successful use of AI tools. The two sections 'Where did AI tools work well?' and 'Where did they not work?' are especially relevant for this evaluation.

To quantify the support the AI tools gave for the different software engineering tasks, we used the survey to propose four statements for each phase, each answered on a 5-point Likert scale (cf. Table 3). The two statements 'Using AI support helps me to … faster.' and 'Using AI support increases the quality of …' explain whether the AI tools worked as expected for this task and supported the students. 'I could have … without AI support.' highlights whether the students were confident in solving the respective task in the development process. Finally, 'I believe I am now better at …, even without AI support.' shows if the students learned development techniques that are useful beyond the AI usage.

4.2 Threats to Validity

Threats to *external validity* arise due to the limited sample size for our data, based on a single instance of one course at one university. However, we already offered two weekly sessions to accommodate for high demand, and since the course is taught in English participants were international and had very diverse prior experiences. While we tried to cover a wide range of AI tools and LLMs, the landscape of AI tools is evolving at a rapid pace, so new tools might become better at certain aspects in future replications of the course. As a threat to *internal validity* there likely also is a bias towards AI tools available at lower costs for students, as we had no budget to cover costly subscriptions and tokens for all students across multiple AI providers. The duration and scope of the project implemented by the students may not be representative of real software development, but we tried to increase realism by adapting the requirements throughout the development. Threats to *construct validity* arise since we use the delivered artefacts as a proxy to estimate student learning. LLMs may not only be used to generate code but also parts of the reports. As it was not possible to track the API requests to the various models, our measurements based on grade/survey/code/repo statistics might generally be inaccurate. However, freedom to make design decisions and about LLM/AI-tool usage is a a core aspect of our course design; since this is not an introductory course we do not want to control such details.

4.3 RQ1: Produced Software

Even though we left the students freedom to choose the libraries used in their implementations, many submissions chose the same frameworks for their final projects. All 63 submissions used the *Spring Boot* framework, which we also used for the demonstrations in the initial lectures, to implement their server. For the client, the demo project demonstrated in the lectures used the *Angular* framework. In their final submissions, 53 students used the *React* framework, five used *Angular*, and the remaining submissions used *Svelte*, *Vue*, or plain JavaScript. In their reports, the students

[4] https://github.com/XAMPPRocky/tokei
[5] https://pmd.github.io/, version 7.17.0
[6] https://github.com/pmd/pmd/blob/9fa1671d4f3a797e3e5f1e3221496349c769f429/pmd-java/src/main/resources/rulesets/java/quickstart.xml

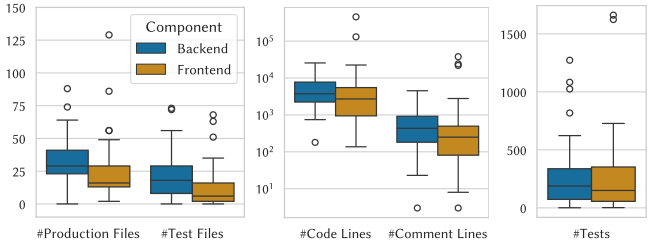


Figure 3: Code statistics of the final project submissions.

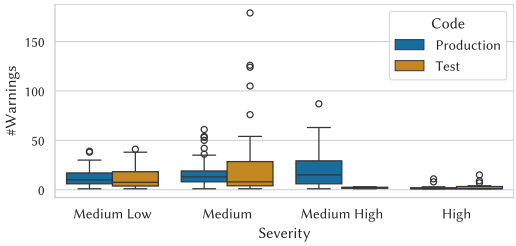


Figure 4: Static code analysis warnings in the final projects.

frequently state that they used these common frameworks due to already being familiar with them from prior projects. Some students also explain that their choice was influenced by the popularity of the frameworks from which they expected improved LLM performance for code generation due to more available training data.

Likely due to the shared implementation framework choices and therefore similar system design, most submissions contained similar amounts of code. Figure 3 shows that submissions typically contained fewer production and test files for the client than the server (client MD = 29, IQR = 60, server: MD = 49, IQR = 35), resulting in 1 000 to 10 000 source lines of code (i.e., excluding comments and empty lines) each. The longer client files likely stem from HTML templates. The median ratio of comment to source code lines is 0.098 5 (IQR = 0.069 0). The smaller amount of comments compared to open source projects [5] is likely a result of the projects' limited scope. Long-term maintainability of the code is also less relevant for this course project.

Figure 3 shows that most students implemented hundreds of tests for each part of the system (unit, integration, and system tests combined; server: MD = 188, IQR = 272 tests, client: MD = 160, IQR = 295). Two students used very fine-granular tests and implemented nearly 1 100 tests for the server and 1 700 for the client.

A static code analysis using *PMD* shows that most submissions contain few warnings (cf. Fig. 4) and that the warnings are not highly critical. The most common 'Medium High' warning contained in the production code concerns the missing conditional evaluation of expressions that are only required as part of logging statements. Other common warnings concern code style (unnecessary imports, 'Medium Low'; missing braces for control statements, 'Medium') or the documentation (empty constructors without comment, 'Medium'). The most common warning in test code are unused imports ('Medium Low'), followed by not using literals on the left side of comparisons ('Medium'), and class attributes that are only used once and could thus be a variable inside a method ('Medium'). Since students should run a linter as part of their CI pipeline, these results indicate that they resolved high-impact warnings already, but ignored lower impact ones. They might also have used different linters or a different linter configuration that does not emit low-impact warnings in the first place.

Most of the observed warnings concern topics regarding code style and maintainability. Only the issue concerning logging might have a slight performance impact during application runtime. The only common warning that could have an impact on the behaviour of the system are the literal comparisons. In case of `null` values such comparisons might throw an exception rather than successfully computing the comparison. However, since these issues are found mostly in test code, we expect the impact to most likely be less helpful error messages for failing test cases.

Overall, the students succeeded in building reasonably complex software systems within the short time given for the final project, with generally high software quality and elaborate test suites.

4.4 RQ2: AI Tool and LLM Usage

Most of the 40 students answering our survey used current and powerful models by *OpenAI* (*GPT-4o*, 29 students) and *Anthropic* (*Claude Sonnet 4*, 30 students) as part of the course-related software development tasks. Other frequently used LLMs are *Claude Sonnet 3.5* (13 students), *Gemini 2.5 Pro* (11), and *GPT-4o-mini* (9). Only one student used a self-hosted LLM (*LLaMA 3*), even though we hosted multiple options on a university server (cf. Section 3.1).

All the frequently used models were available for free to students as part of the *GitHub Copilot* student licence, which we guided them to acquire. We assume that therefore there was little incentive to use the self-hosted LLMs, since the commercial models outperform the smaller local ones. Even when using the open-source tool *aider*, it was possible to configure it to use the *GitHub*-hosted models via an access token extracted from the official *Copilot* IDE plugins.[7]

In their reports, multiple students noted that they switched between models depending on the task. They remarked that *GPT* and *Gemini* tend to work better for natural language tasks, e.g., creating user stories or creating an initial code skeleton from these stories, but then for code-only tasks (e.g., improving the initial skeleton, testing) the *Claude* models tend to work better. This observation is supported by the *SWEBench* and *MMLU* benchmarks.[8]

The most frequently used tool according to our survey is *GitHub Copilot* (30 students), followed by *Cursor* (13), *JetBrains Junie* (6), and *aider* (4). We demonstrated *Copilot* and *aider* as part of the lectures since both are available at no cost to students. Students using *Cursor* paid for a monthly subscription and *Junie* could be used either as part of a trial period or as a paid subscription. Notably, all of these tools have agentic capabilities. These survey results also highlight that most students clearly preferred a direct integration into the IDE (i.e., all tools mentioned above except for *aider*).

While we intentionally designed the course for advanced students and left them the choice of tools and models, when implementing a similar course for less experienced students, e.g., as an undergraduate course where the tool set might be prescribed, the student preferences we observed here can provide guidance.

[7] https://aider.chat/docs/llms/github.html, accessed 2025-09-26
[8] https://www.swebench.com/, https://huggingface.co/spaces/TIGER-Lab/MMLU-Pro

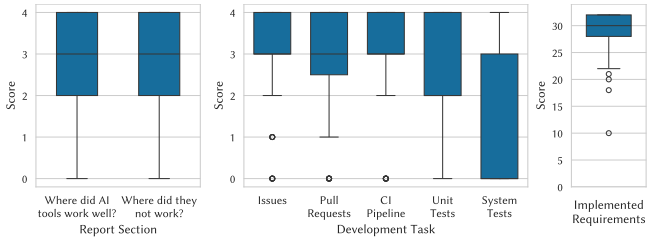


Figure 5: Student course performance.

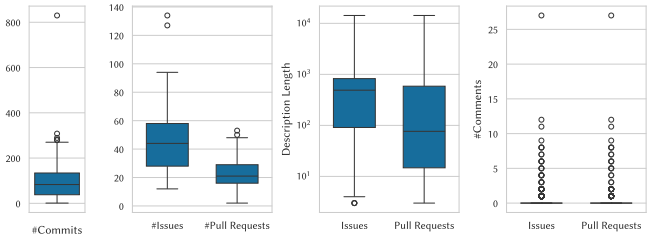


Figure 6: Final project repository content.

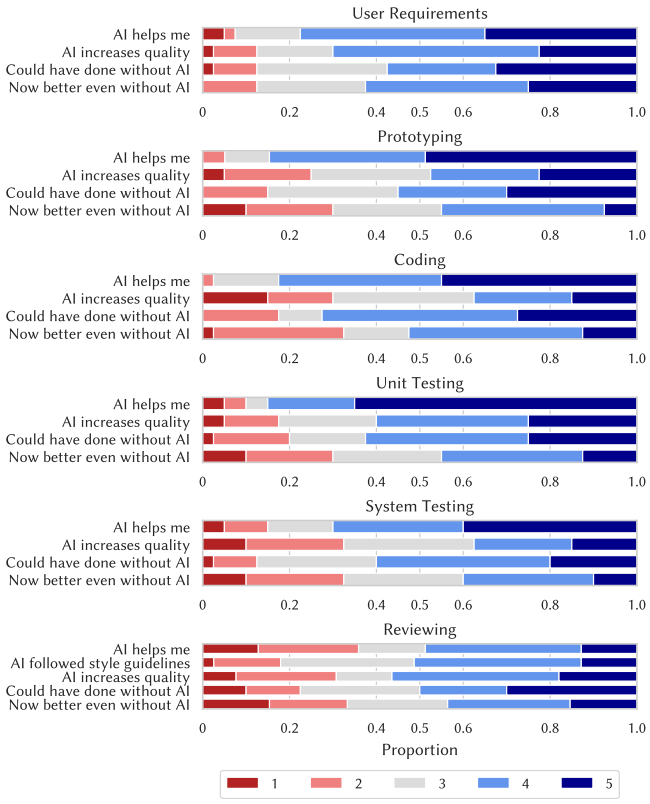


Figure 7: Perceived AI tool support for different software engineering tasks (1=Fully disagree, 5=Fully agree).

4.5 RQ3: AI Tool and Student Performance

As the grading results of the final report show (cf. Fig. 5), most students critically reflected on their AI tool usage, i.e., scoring at least three out of four possible points, both for cases where the tools supported the students and scenarios where they did not. While the survey results in Fig. 7 show that the AI tools supported the students throughout all development phases, there are still considerable differences regarding quality of the produced content.

In the initial requirements engineering phase, the AI tools were clearly helpful (77.5 % of survey respondents), produced high quality output (70.0 %), and many students learned information that is applicable beyond the use of AI tools (62.5 %). These responses align with the contents of the final project repositories. The students created a median of 44 issues (IQR = 30, cf. Fig. 6) to track the system requirements and filled them with a user story in the description that contains a relevant amount of content (MD = 490 characters). This resulted in most (76.2 %) of the students achieving at least three of the four possible points for this development task (cf. Fig. 5).

The following prototyping phase is not associated with a corresponding grading aspect, since any created prototype code could directly be integrated into the final system without requiring an intermediate submission. Since the students could choose their preferred libraries and frameworks for the implementation, they were likely already familiar with them. Thus, the AI tool usage less frequently (45.0 %) taught the students transferrable new skills. Nevertheless, the students clearly considered the AI tools to be helpful in producing the initial boilerplate code (85.6 %).

This trend of subjectively helpful AI tools continues for coding tasks (82.5 %). However, the students are already experienced and could have solved the task without AI support (72.5 %) and therefore clearly noticed cases where the generated code quality was an overall improvement (37.5 %) but also frequently experienced that generated code was not satisfactory (30.0 %). In the grading, this AI tool support with the already high confidence in being able to solve the task regardless of AI support is clearly reflected, since most students successfully implemented the requirements of the final project to achieve 30 of the 32 possible points (MD = 30, IQR = 4).

Unit testing is clearly the category where the AI tools helped the most (65.0 % full agreement, 20.0 % agreement) and also produced high-quality output (60.0 %). However, the students less frequently learned new methods they can now apply even without AI support (45.0 %). A worrisome interpretation of this result might be that they spent less conscious effort on unit testing since the LLMs perform so well at this—which would somewhat defeat the purpose of unit testing. However, more likely this is explained since they were already familiar with the topic and could have implemented tests without AI support (62.5 %). The perceived high quality of the generated tests is also visible in the low number of severe static code analysis warnings for the test code (cf. Fig. 4). Contrary to the perceived helpfulness, however, we can see in the grading that while most students implemented tests in both server and client, the slight majority (32 of 63 students) did not reach a coverage goal of at least 80 % for both components. On the other hand, 22 students managed to cover the required 90 % of the code to achieve a full score. This discrepancy might stem from a possible underrepresentation of students that did not implement any tests (5 students)

or only chose to implement tests for one part of the system (8 students). Using AI to automatically generate the tests might also have been useful even when the students did not achieve a high enough overall coverage to influence the grading.

According to grading results, system testing was clearly the most difficult part of the course. More than half of the students (54.0 %) opted to not implement such tests at all. Not implementing system tests in the first place might have contributed to only 40.0 % of the survey respondents stating that they learned something new about system testing by using the AI. Only 12 students implemented a full end-to-end test suite that not only covers the 'happy path' of important features but also checks edge cases. Nevertheless, still a clear majority (70.0 %) of the survey participants stated that the AI support helped them. Their responses also show, however, that while the AI seems to have produced helpful code, it was of low quality. Only 37.5 % stated that using the AI increased the quality. This is a clear contrast to unit testing. We conjecture that is a result of the difficulty for agentic tools and LLMs to keep relevant files of the system in the context window when generating such tests.

Finally, to ensure code quality, most students created a CI pipeline that at least runs both the server and client unit tests and additionally either linters or system tests (corresponding to a score of at least 3). Such a CI pipeline only is useful when frequently making commits that are checked. Most students followed such an approach, using either atomic commits only (#commits: MD = 83, IQR = 96), or also using pull requests to group related commits (#pull requests: MD = 21, IQR = 13). However, they rarely used automatic code review tools that we can capture from the collected data. Of the 1 286 pull requests made from all students, 1 193 had no comments, even though the *Copilot* integration available on the *GitHub* website could automatically add AI-generated review comments without having to use an external tool. Judging from the survey results, it is also unlikely that many students used automated code review tools in other ways, since compared to the other tasks the AI tools were clearly perceived the least helpful in this one (helpful: 48.7 %, not helpful: 35.9 %) even though the AI tended to adhere to the style guidelines (51.3 %) when it was used. One student mentioned in their report that they at least configured the *aider* agent to run the code formatters and linters after changing the code, then automatically fix any found issues, and only then commit the changes.

While the students overall agree they could have solved all software engineering tasks of the course without the help of AI tools, the majority nevertheless considered the AI tools to be helpful for all tasks except for the reviewing and code quality process. The quality of the produced software together with high degree of reflective thinking exhibited in the reports about the benefits of AI tools (especially for requirements engineering and unit testing) and where AI tools failed to fulfil the quality expectations (e.g., coding and system testing) suggest that, overall, the students successfully learned how to use AI for software development.

5 Discussion

We use this section to highlight challenges during preparation of the course and also use the final project reports to extract concrete examples where students felt supported, some challenges for AI tool usage, and ideas for future tools or improvements to tools.

5.1 Fast-paced Tool Development

The main challenge when preparing the weekly lectures at the beginning of the course was the fast development pace of new AI based tools. Nearly every week new tools, models, or features in existing tools became available. For example, we decided to present agentic tools using *aider* in the lectures since the *GitHub Copilot* IDE plugin did not support such a workflow. However, between the Tuesday and Friday sessions of the lecture, a new update was released with exactly this missing feature. As described in Section 3.1.1, we therefore aim to use the lectures to teach transferrable skills by focusing on generally applicable concepts when using a certain kind of tool, rather than the feature sets of specific tools. Nevertheless, when repeating the course, other exemplary tools that are more suitable to demonstrate the concepts may have become available.

5.2 AI Tool Strengths

Confirming our observations in RQ3, most students also explicitly mention in their reports the requirements engineering (especially writing user stories) and the automatic generation of boilerplate code (e.g., data entity classes, database access methods) as the two areas where AI tools produced high-quality output. As a possible limitation it should be noted, however, that the course considered only a subset of possible requirements engineering tasks, i.e., brainstorming requirements and organising them as user stories. Other aspects like for example the elicitation and prioritisation of requirements given by multiple stakeholders could not be simulated in the course setting. The students also state that both code generation and unit testing for server and client often worked well, not only for the boilerplate code, but also for the implemented features, unless complex logic (e.g., time zone handling) was involved.

A few students explain that LLM generated code sometimes results in discovering new features or library methods. By using the AI chat window in the IDE plugin, the LLM can then also often explain such methods in more detail with generated usage examples that go beyond the method-level documentation that already is natively available when hovering over the method in the code editor.

5.3 AI Tool Shortcomings

In their reports, many students describe system testing as the most challenging aspect both of the course and specifically when using AI tools. They mention that the LLMs tend to generate a plausible-looking test skeleton, which however rarely works due to the identifiers of user interface (UI) elements being often slightly wrong or even clearly non-existing. The AI agents then often failed to find the error location even when analysing the error logs, since they are often unclear due to the mix of errors in the test code, unseen errors in the automated browser, and logging messages of the system's server. Using the AI tools sometimes even made fixing the initially generated test skeletons harder than manually writing system tests. Instead of knowing the intended test steps as a developer and only having to reverse-engineer at which step the test failed, now they also needed to first reverse-engineer the test steps the LLM intended to perform to derive how the test can be fixed (e.g., Did a UI element not load because the test step clicked on the wrong button in the previous step, or did the element not load due to an actual error in the production code?).

A further challenge often encountered when writing system tests, but not limited to this task, is that LLMs tend to rewrite larger pieces of code even though a small fix would have been enough, even when specifically requesting a specific change in the prompt. For example, the LLM could have added some logging statements to a broken test to help both the student and the agent tool to locate the issue, but instead the tool deleted the failing test and added back a different one. Students also report this could sometimes even lead to the agent getting stuck in a rewrite-loop of adding and deleting similar solution attempts repeatedly. As countermeasure, the students started a new conversation/session to start with a fresh context of only few selected files and a precise prompt.

Multiple students mention an issue relating to the training data of LLMs: The generated code often uses deprecated functionalities of libraries, likely due to larger amounts of training code using older versions of the libraries. For example, the LLMs tended to use an older style of the *Spring Boot* server security config, even though the recommended API was changed in the last major version upgrade to the framework in November 2022 already. Similarly, version incompatibilities between suggested JavaScript libraries frequently resulted in invalid import statements or broken builds.

As expected (cf. Section 3.2), the additional requirements handed out during the final project were indeed challenging for LLMs. Multiple students specifically mention in their reports that the LLMs could not resolve edge-cases in the time zone handling, which resulted in having to manually debug and fix them.

5.4 Missing AI Tools and Features

To resolve the described shortcomings, the students mostly envision various enhancements to existing tools rather than entirely new tools. Since most students used agentic tools (cf. Section 4.4), these envisioned features mainly relate to this tool category.

For example, the agentic tools could be improved by having a higher 'state-awareness', retaining some high-level project context like the structure, architecture, or previous design decisions in their context. While some agents can be 'configured' with custom guidelines stored in a specific file in the repository,[9] this does not seem to be sufficient. Constraining the agent by such high-level guidelines could reduce the number of rewrite cycles of code that does not match the intended design, and thus result in less 'wasted' AI budget to invest on actual progress instead. If the agent was able to better follow guidelines across multiple edit steps, it would also allow for the agent to provide a natural language outline of the planned steps beforehand, letting the human approve these steps, and only then automatically perform the steps as planned.

Continuing the theme of improvements to agent context handling, one student suggests that the tool should build some kind of index of the repository and its structure so that it can later automatically load and unload correct files into/from the context. Similarly, two other students suggest an index for libraries and their documentation that is created by the tool provider. By then reading the dependency specification of the project, the agent could internally refer back to the index from the correct version of the library to produce suitable project-specific code instead of just merging together features found in the unstructured training data. This suggests an opportunity for improving agentic tools by integrating approaches like retrieval-augmented generation [13] or knowledge graphs.

To resolve the issue of AI tools rewriting larger pieces of code instead of only applying small fixes, several students suggest putting agents into specific 'modes'. For example, in a coding mode it has more freedom to add new code anywhere, in a testing mode it should focus on test code with only small changes to production code, and in debugging mode it should only make little changes like adding log statements or comments. Such a feature could also improve the refactoring experience. Currently, the tool context is not sufficient to reliably make consistent changes to all affected files. By restricting the agent to make larger changes only in the refactoring target and allowing only necessary fixes in affected files, the target could be retained as main context while the other files are fixed iteratively. Interestingly, the issue of consistent refactorings already appears in the comparatively small course project ($\approx$ 10 000 source lines of code). Such a feature could therefore be especially helpful in industrial use-cases with considerably larger projects.

Overall, these suggestions indicate that the students engaged well with the tools, reflected cricically on their strengths and weaknesses, and came up with concrete ideas for improvements. This shows that the course indeed achieved its aim to teach students how to effectively use AI tools for software engineering.

6 Conclusions

Recent and ongoing developments in AI are disrupting how software is built in practice, and education needs to adapt to these changes to better prepare graduates. In response to this need, we introduced a new course on AI-Driven Software Development that aims to combine a traditional view on phases of the software development process with the many different new possibilities to augment these using AI, while also facing the challenge of how to assess students in a course where courseware is intended to be produced with AI. The course was tremendously popular with students; even though we ran two instances of the course in parallel to accept more participants, many more would have joined if possible. The students' feedback was also consistently positive, while our evaluation demonstrates that the course concept is feasible and students successfully learned how to build software with AI.

Given the success of the course we intend to offer it again in the next academic year, but considering the current rate of change in AI and software development, it is likely that the tool landscape will by then look very different again. It can be hoped that some of the issues and limitations we discussed in Section 5 will be overcome with these changes. However, we—as well as anyone else aiming to replicate a similar course—will need to continuously monitor the landscape of AI tools and update the course material accordingly. We anticipate this effort to be feasible though; since the course is organised around core software engineering phases rather than AI aspects, the overall structure can remain stable.

Acknowledgments

This work is supported by DFG project FR2955/5-1 'Types4Strings'. Thanks to Marko Ivanković and Philipp Straubinger for contributing to the course ideation and design.

[9] e.g., for *Junie*: https://www.jetbrains.com/help/junie/customize-guidelines.html

References

[1] Matin Amoozadeh, David Daniels, Daye Nam, Aayush Kumar, Stella Chen, Michael Hilton, Sruti Srinivasa Ragavan, and Mohammad Amin Alipour. 2024. Trust in Generative AI among Students: An exploratory study. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 67–73. doi:10.1145/3626252.3630842

[2] Rishabh Balse, Bharath Valaboju, Shreya Singhal, Jayakrishnan Madathil Warriem, and Prajish Prasad. 2023. Investigating the Potential of GPT-3 in Providing Feedback for Programming Assessments. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 292–298. doi:10.1145/3587102.3588852

[3] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 500–506. doi:10.1145/3545945.3569759

[4] Ritvik Budhiraja, Ishika Joshi, Jagat Sesh Challa, Harshal D. Akolekar, and Dhruv Kumar. 2024. "It's not like Jarvis, but it's pretty close!" - Examining ChatGPT's Usage among Undergraduate Students in Computer Science. In *Australasian Computing Education Conference (ACE)*. ACM, 124–133. doi:10.1145/3636243.3636257

[5] Tadeusz Chełkowski, Dariusz Jemielniak, and Kacper Macikowski. 2021. Free and Open Source Software organizations: A large-scale analysis of code, comments, and commits frequency. *PLOS ONE* 16, 9 (Sept. 2021). doi:10.1371/journal.pone.0257192

[6] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[7] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2023. Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. (Oct. 2023). arXiv:2310.02059 [cs.SE] doi:10.48550/arXiv.2310.02059

[8] Víctor González-Calatayud, Paz Prendes-Espinosa, and Rosabel Roig-Vila. 2021. Artificial Intelligence for Student Assessment: A Systematic Review. *Applied Sciences* 11, 12 (June 2021), 5467. doi:10.3390/app11125467

[9] Md Asraful Haque. 2025. LLMs: A game-changer for software engineers? *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 5, 1 (March 2025). doi:10.1016/j.tbench.2025.100204

[10] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (Nov. 2024), 1–79. doi:10.1145/3695988

[11] JetBrains. 2024. State of Developer Ecosystem Report. https://www.jetbrains.com/lp/devecosystem-2024/#ai

[12] Vassilka D. Kirova, Cyril S. Ku, Joseph R. Laracy, and Thomas J. Marlowe. 2024. Software Engineering Education Must Adapt and Evolve for an LLM Environment. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 666–672. doi:10.1145/3626252.3630927

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. (May 2020). arXiv:2005.11401 [cs.CL] doi:10.48550/arXiv.2005.11401

[14] Yishu Li, Jacky Keung, and Xiaoxue Ma. 2024. Integrating Generative AI in Software Engineering Education: Practical Strategies. In *International Symposium on Educational Technology (ISET)*. IEEE, 49–53. doi:10.1109/ISET61814.2024.00019

[15] Daniel Mejia, Ernest D.V. Holmes, Jenn Marroquin, and Jamie Gorson Benario. 2025. Bridging Academia and Industry: Leveraging Generative AI in a Software Engineering Course for Practical Industry Experiences. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 507–513. doi:10.1145/3724363.3729036

[16] Simone Mezzaro, Alessio Gambi, and Gordon Fraser. 2024. An empirical study on how large language models impact software testing learning. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 555–564. doi:10.1145/3661167.3661273

[17] Agrawal Naman, Ridwan Shariffdeen, Guanlin Wang, Sanka Rasnayaka, and Ganesh Neelakanta Iyer. 2025. Analysis of Student-LLM Interaction in a Software Engineering Project. In *International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 112–119. doi:10.1109/llm4code66737.2025.00019

[18] Ipek Ozkaya. 2023. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software* 40, 3 (2023), 4–8. doi:10.1109/MS.2023.3248401

[19] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721

[20] Juanan Pereira, Juan-Miguel López, Xabier Garmendia, and Maider Azanza. 2024. Leveraging open source LLMs for software engineering education and training. In *International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 1–10. doi:10.1109/cseet62301.2024.10663055

[21] Nishat Raihan, Mohammed Latif Siddiq, Joanna C.S. Santos, and Marcos Zampieri. 2025. Large Language Models in Computer Science Education: A Systematic Literature Review. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 938–944. doi:10.1145/3641554.3701863

[22] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology* 178 (Feb. 2025), 107610. doi:10.1016/j.infsof.2024.107610

[23] Sandro Speth, Niklas Meißner, and Steffen Becker. 2023. Investigating the Use of AI-Generated Exercises for Beginner and Intermediate Programming Courses: A ChatGPT Case Study. In *International Conference on Software Engineering Education and Training (CSSE&T)*. IEEE, 142–146. doi:10.1109/cseet58097.2023.00030

[24] Michael Vierhauser, Iris Groher, Tobias Antensteiner, and Clemens Sauerwein. 2024. Towards Integrating Emerging AI Applications in SE Education. In *International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 1–5. doi:10.1109/cseet62301.2024.10663045

[25] Wei Wang, Huilong Ning, Gaowei Zhang, Libo Liu, and Yi Wang. 2024. Rocks coding, not development: A human-centric, experimental evaluation of LLM-supported SE tasks. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 699–721. doi:10.1145/3643758

[26] Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. 2025. Agents in software engineering: Survey, landscape, and vision. *Automated Software Engineering* 32, 2 (2025), 1–36. doi:10.1007/s10515-025-00544-2

[27] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2024. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. In *Generative AI for Effective Software Development*. Springer, 71–108. doi:10.1007/978-3-031-55642-5_4

[28] Mounika Yabaku and Sofia Ouhbi. 2024. University Students' Perception and Expectations of Generative AI Tools for Software Engineering. In *International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 1–5. doi:10.1109/cseet62301.2024.10663035

[29] Mounika Yabaku, Nuno Pombo, and Sofia Ouhbi. 2024. Exploring the Potential Use of Generative AI in Software Engineering Education. In *International Conference on Application of Information and Communication Technologies (AICT)*. IEEE, 1–7. doi:10.1109/AICT61888.2024.10740416

[30] Erin Yepis. 2024. Developers get by with a little help from AI: Stack Overflow Knows code assistant pulse survey results. https://stackoverflow.blog/2024/05/29/developers-get-by-with-a-little-help-from-ai-stack-overflow-knows-code-assistant-pulse-survey-results/

[31] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2025. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering* 30, 2 (2025), 50. doi:10.1007/s10664-024-10602-0

[32] Benedikt Zönnchen, Veronika Thurner, and Axel Böttcher. 2024. On the impact of ChatGPT on teaching and studying software engineering. In *Global Engineering Education Conference (EDUCON)*. IEEE, 1–10. doi:10.1109/educon60312.2024.10578680