Industry Team Project: An Experience Report on Apprenticeship-Style Software Engineering Education *

Sarah Van Wart and Kevin R. Sanft
Department of Computer Science
University of North Carolina Asheville
1 University Heights
Asheville, NC, 28804
{svanwart, ksanft}@unca.edu

Abstract

Junior software engineers often begin their careers by joining a project, learning code created by others, and contributing with guidance from experienced developers. With the right support, this apprenticeship-style environment can help novices understand an existing system, extend it incrementally, and ultimately contribute substantive new features. We received a grant to build a web platform for a healthcare client and designed a studio-style undergraduate course around that work, engaging nine students in developing the platform. Students began with a fixed technology stack, an AI-assisted starter codebase, and a structured onboarding period, then assumed greater responsibility for product design and implementation decisions. We found that this structure encouraged attention to quality and sustainability, supported ownership and meaningful participation, but also introduced a steep learning curve, ambiguity, coordination costs, and substantial instructor scaffolding. We also found that new software development tools can reduce some of the effort required to create and support similar apprenticeship-style environments.

1 Introduction

A central challenge in software engineering education is helping students develop the skills required to contribute responsibly to complex, professional-quality software projects. This requires both *professional judgment* – interpreting incomplete requirements, coordinating team-based work, and negotiating priorities for real users – and *technical fluency* – working with current tools, technologies, and development practices that change over time.

At University of North Carolina Asheville (UNCA), a small liberal arts institution, our computer science curriculum attempts to address this challenge. Among other requirements, an upper-level Software Engineering course gives students experience with modern tools and collaborative development through short-term projects. A two-semester capstone sequence gives students experience defining, building, and deploying a substantial software project of their own design independently. However, neither course captures the typical experience of a new software engineer: joining an existing, long-term project shaped by other people's decisions, where senior engineers or technical leads often serve as mentors in an apprenticeship-like setting. Furthermore, we anticipate that advances in AI code generation tools will continue to raise expectations for junior developers. These considerations motivated us to design a course giving students experience with a collaborative, long-term, real-world project in which faculty serve as senior engineers.

We structured this new course, *Industry Team Project* (ITP), like an apprenticeship in which students built a web-based software application for a real client as part of a larger grant-funded collaboration. Importantly, our collaborators needed the software for a planned research study involving real users of the application. Therefore, students' contributions would carry consequences beyond the course, creating a sustained context for developing professional judgment. This experience report describes the first offering of ITP and examines how the course's design supported meaningful participation and attention to quality while also introducing a steep learning curve, ambiguity, coordination costs, and substantial instructor scaffolding.

2 Background

Creating experiences that prepare students for industry is a persistent challenge in computing education. Learning *specific tools and technologies* is part of this challenge. Modern software development relies on frameworks, reusable components, design systems, cloud services, and automated testing and deployment. Design systems, for example, require developers to create and maintain shared components while balancing consistency with the needs of evolving products

[7]. DevOps practices similarly require familiarity with CI, deployment, and related tooling, which can be challenging to incorporate into software engineering courses [4, 12]. Generative AI adds another layer: students can use these tools for programming and software design, but the benefits depend on their ability to evaluate and revise their outputs [11].

Beyond learning to use current tools, students must also learn to *exercise judgment* about when and how to use them within a real system. One way to understand this deeper challenge is through Lave and Wenger's account of situated learning [8], in which competence develops not primarily through instruction but through participation in a community of practice: novices take on real, bounded responsibility for authentic work, and gradually assume greater responsibility as their competence and judgment develop.

Educators have pursued different ways of approximating these forms of participation, each foregrounding different aspects of professional work. Client-based courses and student-run organizations, for instance, emphasize responding to real clients, working through ambiguity, and accountability for quality [2, 5]. Multi-semester and inherited-codebase courses, by contrast, emphasize reading and interpreting existing code, learning unfamiliar conventions, and contributing work others will continue [3, 10, 13]. These dimensions often overlap, but none fully reproduces the sustained, consequential participation Lave and Wenger describe.

2.1 Project Context

The software project for the course was part of a larger collaboration between Emory School of Medicine, Appalachian State University, and UNCA. The Emory collaborators had previously delivered an educational and social-support program for parents of young children with motor delays (e.g., cerebral palsy) through a social media platform [9] and wanted a standalone web application informed by feedback from that study. The proposed application, "3MA," would provide weekly educational modules with slides, videos, and quizzes and support synchronous meetings among parents and healthcare professionals. A two-year, deliverables-based grant required an interactive prototype in spring 2026 and a minimum viable product by mid-summer.

3 Course Design

We designed ITP as an upper-level Computer Science elective, and structured it as a studio-style, apprenticeship-like experience. We gave it an upper-level *Software Engineering* prerequisite, which in turn requires *Data Structures* and *Systems*, so students entered with a strong programming foundation and expo-

sure to collaborative Git workflows, automated testing, linting and formatting, Docker, APIs, React, and databases.

3.1 Technology Stack and Starter Code Repository

We created a starter codebase for 3MA using a conventional architecture and widely adopted tools selected for long-term support rather than novelty (Table 1). This was an intentional design choice: rather than allowing students to choose the system's foundations, we gave them a fixed stack and architecture to inherit. We used Cursor [1] to scaffold boilerplate and configuration from our architectural specifications and establish automated testing, linting, formatting, code-review conventions, and CI/CD.

To test the feasibility of the starter system, we implemented several client features as a proof of concept prior to the semester. We then removed these features, deliberately preserving space for students to build the system's first real features themselves during the onboarding phase. The starter system included user accounts, several data models and API endpoints, basic web and mobile interfaces, automated quality checks, Docker configuration, and a GitHub-based CI and deployment workflow. Students therefore began by extending a working full-stack system rather than assembling it from scratch. We created setup, workflow, and technical guides, along with a shared cheatsheet for common commands and development tasks, drafted with assistance from Cursor.

Table 1: Technology Stack

Function	Technology
Backend / middleware layer	FastAPI with JSON Web Tokens (JWTs)
Data layer	PostgreSQL+SQLAlchemy, S3
Web interface	TypeScript, React, Mantine
Mobile interface	TypeScript, React Native
Development environment	Docker (web and backend)+Expo (mobile)
Testing and static analysis	Pytest, vitest, prettier
CI and deployment	GitHub Actions and Railway
Shared interface design	Figma components, variables, design tokens

3.2 Two-Phase Course Structure

We designed the course to have two phases: (1) shared preparation, followed by (2) specialized work on client-facing features.

Table 2: Course Phases

Weeks	Stage	Units and Assignments
1-7	Shared preparation	Engineering: project and team onboarding; backend testing; data modeling and API design; refactoring; web development; mobile development
8-9	Shared preparation	Product Design: user-centered design; user journeys; prototyping; Figma design systems
10-15	Specialized product work	Role-based work on a Figma prototype and feature implementation, integration, testing, and documentation.

Shared preparation was intended to build technical fluency and a common understanding of the system, product, and development practices before students assumed specialized roles. In the engineering phase of shared preparation, students collaborated in small teams on structured work across the technology stack, largely within predefined scope and requirements. Students coordinated dependent tasks, contributed through pull requests, and worked toward shared expectations for functionality, testing, code quality, documentation, and review. We prohibited AI-assisted development until the final week of technical onboarding so that students could first develop a working mental model of the system, including the structure, behavior, and relationships among its components.

Weeks 8–9 were intended to create more space for students to exercise independent judgment by participating in the user interface design process. With input from the client, students would interpret incomplete requirements, negotiate priorities for real users, and develop user journeys, screen inventories, and prototypes in Figma to guide implementation.

During Weeks 10–15, newly formed teams would complete three two-week sprints focused on distinct vertical feature slices. These sprints aimed to bring professional judgment and technical fluency together under real constraints of scope, quality, and time.

3.3 Course Execution

In spring 2026, nine students enrolled in the ITP course. The students ranged from juniors who had not yet begun capstone work to seniors graduating that semester. The execution of the course followed the original course design, but we adapted the plan as necessary in response to changing project priorities.

The initial seven-week *engineering onboarding* proceeded largely as planned. Through pull requests and peer review, students completed assignments across

the backend, web, and mobile layers. One assignment, for example, required each team to implement functionality for creating and managing educational modules, including API routes and tests. During the first three weeks, we provided detailed feedback on pull requests, peer reviews, testing, documentation, and adherence to project conventions. We also met individually with students to discuss progress and provide targeted technical and collaborative support. As students became familiar with the workflow, we shifted toward in-class coaching and troubleshooting. By the end of onboarding, students had worked across the application's major layers and developed a shared foundation for specialized product work.

In the *product design* (shared preparation) period, students formed three teams focused on the product's primary user groups: parents, healthcare providers, and administrators (content managers). Each team identified the tasks and information its users would need, translated those needs into user journeys, and designed the screens and workflows required to support them. This phase, originally scheduled for two weeks, expanded to about four: the client was unavailable for much of it, and the product was less defined than anticipated, so students worked from an earlier feasibility study and training materials instead.

By the time specialized product work began, only four weeks remained, leaving time for one sprint rather than the three originally planned. Based on responses to a team and feature preference form, students divided into three teams. One team produced a high-fidelity prototype, while the other two implemented video management and user onboarding features.

3.3.1 High-Fidelity Prototype Team

Two students synthesized the class's user journeys, wireframes, and interface decisions into a polished prototype spanning the administrative web interface and the parent-facing desktop and mobile experiences (Figure 1). It showed how staff could manage cohorts, instructional content, quizzes, and meetings, and how parents would move through learning modules and related activities.

3.3.2 Implementation Teams

The other two teams implemented video-integration and user-onboarding workflows. The video team developed functionality through which administrators could manage instructional videos and parents could view them in the web and mobile applications. The onboarding team developed the invitation, registration, and account-activation workflow. In both teams, students divided responsibility across interdependent system components.

Based on pull requests, commit histories, changed files, and automated tests (which do not capture all planning, debugging, peer assistance, or uncommitted

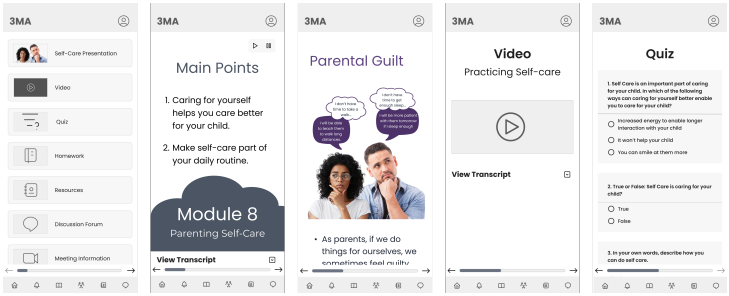

Figure 1: Prototype of the mobile interface for parents

work), both teams completed substantial portions of their workflows. However, the compressed timeline left too little time for full integration and end-to-end testing.

4 Discussion

At the end of the semester, all nine students completed a questionnaire about their preparation, contributions, challenges, and overall experience. We drew on this survey data, students' weekly reflections, their code contributions, and our own teaching observations to understand how the course design shaped their experience. Below, we group our findings by theme, showing how specific design decisions shaped student experience and the tradeoffs they introduced.

4.1 Inheriting an Existing System

Course Decision: ***A Fixed Technical Foundation.*** As noted in Section 3, the starter system shifted students' attention from selecting technologies to understanding system conventions, tracing behavior across layers, extending shared patterns, and producing work others could continue.

Student Experience: ***Emphasis on Quality and Long-Term Sustainability.*** Inheriting a continuing system shaped the design and engineering practices students emphasized. Because others would use, maintain, and extend their work, students placed greater importance on quality, polish, and long-term sustainability. When asked which contributions were most valuable, they described using tests to protect refactoring, documenting API contracts, following established conventions, and reviewing code for consistency and security. Student "S1" explained that because the work "won't disappear after the class ends," it required greater attention to "quality and maintainability."

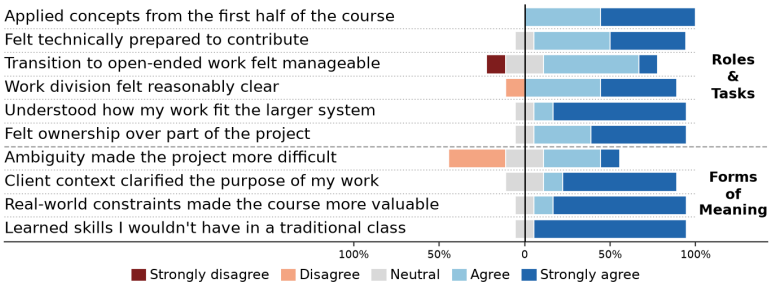

Figure 2: Student Responses to Selected Survey Questions (n=9)

Similarly, S3 described a cohort-management system as "protected against regressions if refactors occur down the road" because "the contract lives in the tests." Unlike conventional assignments, this work needed to be understood, trusted, and extended by teammates and future developers.

Survey responses suggest this understanding didn't always translate into confidence: students generally understood how their contributions fit into the system (Figure 2), but were less confident applying onboarding concepts independently.

Tradeoffs: ***Realism vs. Ease of Entry.*** For students, the fixed technical foundation gave them experience contributing to a realistic, shared system and encouraged them to think more deeply about quality and long-term sustainability. At the same time, it created a steeper learning curve and fewer opportunities to make foundational technical decisions themselves.

For instructors, the fixed foundation made it possible to provide a more complex and production-like system than students could reasonably build from scratch during a semester, but doing so required substantial scaffolding and support. This included simplifying the codebase, staging assignments, documenting conventions, and providing intensive feedback. Cursor helped us build a more capable starting system, but it did not reduce the instructional work required to make that system accessible to students.

4.2 Product Design and Prototyping

Course Decision: ***Students' Responsibility for Product Design.*** Although students inherited the technology stack, architecture, and development processes, substantial product questions remained unresolved. They had to translate broad client goals into user journeys, feature behavior, and technical decisions.

Student Experience: ***Agency, Ownership, and Communication.***

Leaving these decisions open gave students opportunities to exercise professional judgment over important aspects of the product. The shared prototype also supported a related need: communicating and coordinating that emerging direction across roles. During the final four weeks, most students described the shared design artifact as central or useful to their work. S3 said that Figma clarified "a huge amount" of uncertain requirements and "helped me a lot as a developer," while S1 was proud of presenting work that "really helped [the client's] understanding of the project." Consistent with HCI research on prototypes as design tools [6, 14], the prototype functioned as a *boundary object* among students, instructors, and the client. It made workflows, interface states, visual conventions, and unresolved decisions visible without eliminating the implementation choices developers had to make.

Tradeoffs: ***The Work of Navigating Uncertainty.*** Design required more time and support than anticipated, both because of the client's limited availability (Section 3.3) and because students were learning unfamiliar UI/UX methods alongside the necessary Figma skills. The open-ended and iterative nature of the design process also created ambiguity that several students found uncomfortable, but it also gave them opportunities to practice making progress without fully specified requirements or a predetermined solution. Although extending this phase reduced the time available for implementation, it gave students significant influence over the product and a shared understanding of key decisions. Survey responses reflect this tradeoff: four students agreed that ambiguity made the project more difficult, yet eight reported a sense of ownership, seven said the client context clarified the purpose of their work, and eight said real-world constraints made the course more valuable even when they made the work harder (Figure 2).

4.3 Team-Based, Interdependent Work

Course Decision: ***Team-Based Work.*** Students worked in teams throughout onboarding, design, and implementation. Although responsibilities differed within and across teams, the work remained interdependent: students' progress and the quality of the product depended on the decisions and contributions of others.

Student Experience: ***Ownership, Purpose, and Visible Contribution.*** Students generally experienced this structure as giving them a recognizable place within the larger effort: eight of nine reported ownership over part of the project, and open-ended responses showed that they understood their contribution more broadly than completing assigned technical tasks (Figure 2). S1 described serving as an "organizer, encourager, and Figma designer," while S5 identified as the "encourager/vibes guy" and found it valuable to help beyond his assigned role. These roles overlapped with programming and design

work, making otherwise less-visible forms of collaboration recognizable.

This broader range of contributions became important because students' work affected both teammates and the larger product. S5 described the semester as "a true project in which everything had a cause and effect associated with it" and "more like a job in some senses, than a school project"; S4 similarly said that working for a real client gave the work "more purpose."

Tradeoffs: ***Dependency, Pressure, and Coordination Costs.*** The same interdependence that made contributions meaningful also created blocking, uncertainty, and pressure. S8 reported that tasks depended on other teams, were not always connected, and sometimes required decisions before the client's expectations were fully known. S2 described feeling worried, uncomfortable, and at times "inept," even while expressing pride in what they contributed. Team-based, interdependent work made participation consequential, but also required coordination and support when work became blocked or uncertain.

5 Conclusion

This experience report describes one course, one client project, and nine students, so its lessons are necessarily provisional. Even so, the course suggests value in giving students a structured way to enter an existing system, learn its conventions, and gradually assume responsibility for more consequential work. Students' experiences also highlight the tradeoffs of this model: inheriting a sophisticated system created a steep learning curve, open-ended design introduced ambiguity, and interdependent work required substantial coordination and instructor support. At the same time, these conditions gave students opportunities to contribute to work that would continue beyond the course, exercise substantive judgment, and attend to concerns such as maintainability, usability, and quality.

More broadly, new software development tools – including AI-assisted development, design systems, and increasingly capable development platforms – make this kind of course design more feasible by lowering the cost of creating and maintaining a starter system. Cursor helped us scaffold the application boilerplate, development infrastructure, and documentation, creating a solid technical foundation that students could learn from and extend. But the experience also suggests that starting from a higher level of abstraction does not diminish the pedagogical work of apprenticeship: instructors still had to simplify the system, stage students' entry into it, provide intensive feedback, and decide where students should encounter ambiguity and exercise judgment. As these tools expand what instructors can build, an important course-design question is not simply what students can now produce, but what responsibilities, decisions, and forms of judgment we want them to practice.

References

[1] Anysphere. Cursor: The AI code editor. `https://www.cursor.com/`, 2026. Accessed: 2026-07-31.

[2] Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education (TOCE)*, 15(4):1–31, 2015.

[3] Janet Davis and Samuel A Rebelsky. Developing soft and technical skills through multi-semester, remotely mentored, community-service projects. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 29–35, 2019.

[4] Marcelo Fernandes, Samuel Ferino, Anny Fernandes, Uirá Kulesza, Eduardo Aranha, and Christoph Treude. DevOps education: An interview study of challenges and recommendations. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Software Engineering Education and Training*, pages 90–101, 2022.

[5] Mike Holcombe and Andrew Stratton. VICI: Experiences in introducing student run software companies into the curriculum. In Mike Holcombe, Andy Stratton, Sally Fincher, and Gary Griffiths, editors, *Projects in the Computing Curriculum*, pages 103–116, London, 1998. Springer London.

[6] Stephanie Houde and Charles Hill. What do prototypes prototype? In Martin Helander, Thomas K. Landauer, and Prasad Prabhu, editors, *Handbook of Human-Computer Interaction*, pages 367–381. Elsevier, 2nd edition, 1997.

[7] Yassine Lamine and Jinghui Cheng. Understanding and supporting the design systems practice. *Empirical software engineering*, 27(6):146, 2022.

[8] Jean Lave and Etienne Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, Cambridge, UK, 1991.

[9] Nathalie L. Maitre, Larken Marra, William Kjeldsen, Lisa J. H. Pinson, Rachel Byrne, Zhulin He, and Melissa M. Murphy. A social media-delivered intervention for motor delays: stage-ib randomized clinical trial and implementation exploration. *Pediatric Research*, 99(2):693–702, 2026.

[10] Saheed Popoola, Vineela Kunapareddi, and Hazem Said. Developing and sustaining a student-driven software solutions center—an experience report. *Journal of Systems and Software*, 220:112279, 2025.

[11] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. The widening gap: The benefits and harms of generative AI for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1*, pages 469–486, 2024.

[12] Edgar Sarmiento-Calisaya, Alvaro Mamani-Aliaga, and Julio Cesar Sampaio Do Prado Leite. Introducing computer science undergraduate students to DevOps technologies from software engineering fundamentals. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, pages 348–358, 2024.

[13] Anshul Shah, Jerry Yu, Thanh Tong, and Adalbert Gerald Soosai Raj. Working with large code bases: A cognitive apprenticeship approach to teaching software engineering. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pages 1209–1215, 2024.

[14] Susan Leigh Star and James R. Griesemer. Institutional ecology, 'translations' and boundary objects: Amateurs and professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. *Social Studies of Science*, 19(3):387–420, 1989.