

DOI: https://doi.org/10.48009/2_iis_2026_121

Using AI for prototype development in the systems analysis and design course

John E. Anderson, *Utah Valley University*, janderson@uvu.edu
Daniel McDonald, *Utah Valley University*, Daniel.McDonald@uvu.edu

Abstract

This exploratory study examines the feasibility of using generative AI to support prototype development in the Systems Analysis and Design (SAD) course. It highlights the importance of the SAD course within Information Systems curricula, defines the requirements-to-prototype challenge, and considers the emerging practice of “vibe” coding as a means of developing prototypes. Four generative AI tools (Lovable, Replit, Cursor, and Claude Code) were evaluated, with their respective strengths and limitations analyzed. The findings indicate that these tools can produce functional and usable prototypes, exceeding initial expectations. The study also underscores the importance of structured analysis techniques in improving the quality of AI-generated prototypes. Overall, the results inform instructional practices by identifying opportunities to enhance the SAD course.

Keywords: vibe coding of prototypes, systems analysis and design, requirements specifications document

Introduction

The Systems Analysis and Design (SAD) course is a cornerstone of the Information Systems (IS) program and has been a key component since the program’s inception. In the SAD course, students complete a semester-long project with three main deliverables: a Business Case, a Requirements Specification Document, and a Solution Design Document. The follow-on course, Systems Design and Implementation, requires students to build what they designed. A key challenge is the difficult transition from conceptual design to implementation, which often leads to failures. To address this, faculty are exploring whether AI-assisted prototyping using detailed requirements documents can improve design quality and better support implementation. Since Generative AI (GenAI) has the potential to produce an app with just a text description, could it be used in the Systems Analysis and Design course to create working prototypes? We decided to do an exploratory feasibility study of using GenAI to create prototypes using the Requirements Specification Document (RSD) that students write in the SAD course.

Background and literature review

A review of the literature found that the role of a systems analyst is a key career in computing and continues to expand and that the SAD course is one of the most important courses in the Information Systems program. In the course, students learn a structured approach to analysis from which design follows. GenAI

enabled “Vibe” coding has proven to be an effective, accessible approach to software development with a low barrier to entry. “Vibe” coding thus has the potential to help students create high-fidelity working prototypes to validate and improve their designs. However, to create quality prototypes, requirements must be specified using a carefully structured process.

Importance of System Analysis and Design in Industry

The U.S. Bureau of Labor Statistics estimates there are currently over a half a million computer systems analysts and that employment is projected to grow 9 percent from 2024 to 2034 (while the average growth rate of all occupations is 3 percent) with about 34,000 openings each year. (Bureau of Labor Statistics, 2025) The computer systems analyst is one of the ten main computing jobs tracked by the bureau.

Importance of the SAD course in the Information Systems Program

The Systems Analysis and Design (SAD) course has long served as a cornerstone of IS curricula, emphasizing structured problem solving, requirements analysis, and iterative system development. Since IS’97, analysis and design have been recognized as central competencies, and one of the two primary domains of the IS discipline. Notably, analysis and design skills ranked among the most critical capabilities for IS graduates, reflecting their importance in bridging organizational needs with technical solutions (IS’97, IS 2002, IS2010, and IS2020).

A persistent theme across IS curriculum models is the role of prototyping in supporting iterative and exploratory design. IS’97 explicitly called for students to “use rapid prototyping and other similar alternative mechanisms for rapid development of information systems” (IS’97, p. 83). This emphasis continued in IS 2002 and IS 2010, where prototyping was embedded within requirements analysis, user interface design, and high-level system modeling (IS 2002; IS2010). By IS 2020, prototyping was further associated with emerging technologies, reflecting the increasing importance of applying advanced tools to solve business problems (IS2020).

Experiential Learning and Use-Case-Driven Modeling in the SAD Course

Our SAD course emphasizes experiential learning (Kolb, 1984) through a semester-long, hands-on project where students gain concrete experience, reflect on their learning in discussions, bring abstract concepts together as the project progresses, and actively experiment with many SAD methods. The importance of experiential learning in Information Systems education is widely recognized and recommended by professionals, accrediting bodies such as ABET and AACSB (ABET Computing Accreditation Commission, 2026; AACSB, 2026), and curriculum guidelines such as IS 2020 (IS2020). It is regarded as a critical pedagogical approach that facilitates “learning by doing,” helping students translate theoretical knowledge into practical skills and promoting lifelong learning outcomes. The provided sources highlight three primary areas in IS where experiential learning is successfully integrated: AI and Machine Learning (Halaweh, 2025), IT Project Management (Javadi et al., 2025), and Enterprise Resource Planning (ERP) (Dorey et al., 2026). Both Halaweh (2025) and Dorey et al. (2026) found that experiential learning improved academic performance, increased student interest, and helped students move beyond remembering to higher forms of Bloom’s Taxonomy.

One of the experiential learning methods used in our SAD course is a use-case driven modeling approach for a real client problem. Use-case-driven modeling follows a deliberate, sequential workflow that moves from behavioral requirements to structural design. The process begins with use-case diagrams to define system scope and user goals, then progresses through detailed use case specifications, and finally scoped ERDs that define persistent data. Each artifact is derived from the previous one, creating a strong traceability chain that links every database table back to a specific user requirement. This approach prioritizes use cases over structural models to avoid analysis paralysis, over-engineering, and

inconsistencies caused by parallel modeling. By working in focused “use case slices,” teams can iteratively design, validate, and implement functionality with clear justification. (Go-UML, 2024)

A “Vibe” Coding Solution for Prototype Development in SAD

In our SAD course, the last part of the team project is to create a prototype of the new system. This usually has been wireframe based and has been of highly varied quality. In the next course the student teams implement the project, also with outcomes of varying quality, because the students have varying levels of software development expertise. Could using GenAI “vibe coding” to create prototypes be a possible solution? The term “vibe coding,” coined by Andrej Karpathy in a February 2025 X post, has since gained considerable momentum (Karpathy 2025). Vibe coding is a fast, improvisational approach to software development in which a developer collaborates conversationally with a large language model acting as a pair programmer. An important advantage of vibe coding is its low barrier to entry, it doesn’t require formal programming expertise. While traditional development skills are beneficial, a user can participate through a chat-based interface, making it comparable to no-code or low-code approaches. Unlike those, however, vibe coding still generates real source code that can be extended through conventional methods. (Borg et al, 2025)

Prototyping is an iterative, hands-on way of exploring and refining ideas by creating tangible representations that can be evaluated and improved through feedback. It is widely used across design and software disciplines, with prototype scope and fidelity shaping both cost and the kind of insights gained. Low-fidelity prototypes like sketches or wireframes are inexpensive and effective early on, while high-fidelity, executable prototypes are costlier but enable deeper validation of behavior, interactions, and quality attributes. Vibe coding has the potential to significantly lower the cost of creating such executable prototypes. This is especially valuable in startup contexts, where working software helps establish credibility with customers and investors, but traditional engineering skills may be scarce. (Borg et al, 2025)

Bjarnason et al 2023 found that prototyping plays five key roles in product and software development: communication, validation, incremental development, exploration, and learning. Prototypes help with external communication by conveying product vision to customers, users, and investors. They help with internal communication by aligning understanding across teams, product owners, and managers. Prototypes help with product refinement through validation and testing by focusing on gathering early feedback from internal stakeholders and external users and enable realistic usability testing. Research shows that executable prototypes can serve as seeds for later implementation. Prototyping allows teams to test assumptions, explore alternatives, and improve product and market fit at low cost. At the same time, it raises new challenges: throwaway prototypes may not meet the demands of maintainability or security, and the shift toward rapid hi-fi prototyping may reduce the kind of creative exploration low-fi sketches encourage. However, generative AI introduces new forms of synergistic creativity and may enhance divergent thinking. Learning how to effectively harness this human/AI “vibe” for co-creation remains an important area for future research. (Bjarnason et al 2023)

High Quality Specifications Necessary for AI Prototyping

At the AI Engineer World’s Fair 2025, OpenAI’s Sean Grove claimed that whoever writes the specification is now the programmer since AI can take it from there because AI can generate both the code and other downstream artifacts, such as test cases and documentation. (Grove, S. 2025)

Others argue that requirements alone are not enough, that a developer does not go from requirements to code directly. Critical knowledge such as domain expertise, design decisions, risks, and edge cases cannot be fully expressed in source code. Instead, it lives in layered artifacts (requirements, analysis, and design documents) that bridge the gap between real-world needs and functioning software. When automated tools

ignore these abstractions, the resulting code may initially work but fail to embody this accumulated knowledge. Over time, this leads to growing technical debt, loss of maintainability, and eventually stalled progress. Designing abstractions that capture real-world complexity while remaining manageable is still a uniquely human skill, supported by our ability to reason across large contexts and cope with complexity. Abstractions are the foundation of software engineering. Requirements must be systematically translated from the problem domain into the solution domain through successive refinements. Skipping this step undermines the entire development process and has not proven effective. (Steghofer, 2026)

Methodology

In the SAD course we currently teach, students do a semester-long project with the following deliverables: Business Case (includes the strategy analysis and feasibility analysis), Requirements Specification Document (includes use case diagram, fully-dressed use cases, ERD), and a Solution Design Document (includes prototypes of a report, user flow diagram, wireframes of screens, help and error messages). The next follow-on course is Systems Design and Implementation where the students implement the project they just created. One of the recurring challenges is the heavy lift from the conceptual/logical design to implementation which has a high failure rate, especially in IS programs where students have less programming in the curriculum compared to Computer Science. We, the faculty, wondered if doing the prototype using GenAI would help make better solution prototypes (wireframes, reports, etc.) and smooth the implementation for the next course. We also wondered if the Requirements Specification Document, because of its detail, and common use in development, would be a tool that might yield good prototype results. Thus, we decided to do a feasibility study as to the usefulness of GenAI tools for this challenge.

We began by reviewing all widely used AI development tools (in the popular professional literature) and selected four, each aligned with a different level of student expertise in software development. As we looked at possible GenAI development tools it became apparent that they grouped into three general categories. The first category is App Builders. These tools are often online, with immediate cloud deployments. App Builder tools do not require local installations and have more limitations regarding the types of coding languages and platforms. Lovable and Replit fall into this category. Lovable was chosen as the most beginner-friendly option, with Replit as an intermediate-skill-level cloud-based platform. Both are considered major players in the AI coding space with Lovable having 8 million users and Replit having over 35 million users. The next category of AI tools is the Integrated Development Environment (IDE) tools. These tools often fork VS Code and integrate their AI tool with the fork. The IDE tools require a local installation of their tool. Cursor as an example of this accessible client-side IDE. Finally, there are AI solutions referred to as extensions or agents. These tools can browse code repositories and make changes. They not only operate outside of code editors but also work within windows of many different IDEs. We chose Claude Code as an example of a coding agent/extension. This type of productivity tool is designed for experienced programmers. Thus, we selected Lovable as a beginner-level tool, Replit as an intermediate option, Cursor as an IDE-based tool, and Claude Code as an advanced coding agent, ensuring representation across different skill levels.

We then used a representative student-created Requirements Specification Document (RSD) which included an introduction, use case diagram, fully-dressed use cases, and an ERD along with a prompt “Create a web app using the attached pdf as a requirements document,” as input for each tool. The RSD was for a student project called Coach Tim which was a solution for tracking athlete data. The document contained six fully-dressed use cases: athlete logs in, athlete enters data, athlete submits data, coach logs in, coach views/edits data, coach runs reports.

We evaluated each tool using generic software evaluation criteria including functional fit, usability, technical attributes, community, cost/licensing, and organizational fit. We then developed a weighted ranking matrix, with two faculty evaluators independently conducting initial ratings before collaborating to reach consensus on the final rankings. The scoring followed 1-5 with 5 being the best. The weighting of each criteria was jointly agreed upon by the two faculty evaluators with an eye toward weights that would match a public regional university.

The evaluation criteria included functional fit, which assesses whether the tool supports required capabilities and integrates with necessary languages, data formats, or platforms; usability and developer experience, which consider the learning curve, clarity of documentation, and ease of use for students and collaborators; and technical attributes, such as performance, scalability, security, and extensibility through plugins or APIs. Additional criteria include ecosystem and community, focusing on community size, update frequency, and availability of tutorials and libraries; cost and licensing, including whether the tool is open-source or proprietary and its suitability for university use; and organizational fit, which examines alignment with existing expertise, compatibility with current infrastructure, and availability of IT support.

We assigned the greatest emphasis to functional fit (30%), defined as the tool's ability to support the development of a high-quality prototype from a RSD, and usability (25%), reflecting how quickly students could learn to use the tool. These criteria were prioritized because the tools are intended for use in a SAD course rather than a programming course. Within this context, technical factors (such as performance, scalability, and security) were weighted at 20%, while ecosystem considerations were given a smaller weight of 5%. We note that in a programming-focused course, these technical and ecosystem factors would likely be rated higher. Cost was weighted at 10%, with a preference for free or low-cost tools to ensure accessibility for students. Finally, organizational fit was assigned a weight of 10%, reflecting how well each tool aligns with the programming knowledge and skills developed within our current curriculum. Alternative weights may yield different results. We also jointly prepared qualitative assessments to support the evaluation of each tool.

Results

All four tools were more than capable of creating working prototypes of web applications given only the requirements document as input. The tools were fast (all finished in under 15 minutes) and produced acceptable web applications, especially from a prototype standpoint. Differences between the tools included differing setup requirements, differing flexibility of choice in coding languages and AI models, differing levels of adherence to only the requirements document, and differences in styling and usability of the website.

Lovable

Lovable required basically no setup before getting started. The default AI model is currently Google Gemini Flash but does allow some prompt-driven changes, which are mostly hidden. It runs on the Web and produces a cloud app from nothing more than the requirements document. After being told to start, Lovable worked for 8 minutes and built a login page, athlete dashboard, and coach dashboard, which included user management, athlete data (with seed data), and reports. The dashboard screen is shown in Figure 1. The datastores initially did not persist in the cloud or anywhere. We were able to type "Enable Cloud" and the cloud backend was added. There were a few minor errors, such as the workout dates were off by a day when recorded by the athlete. Other than that, all the functionality worked. The reports displayed colored line graphs and bar charts as opposed to just numbers. Reports, however, did not aggregate workouts that took place on the same date. Each workout was listed as a separate entry in all the reports.

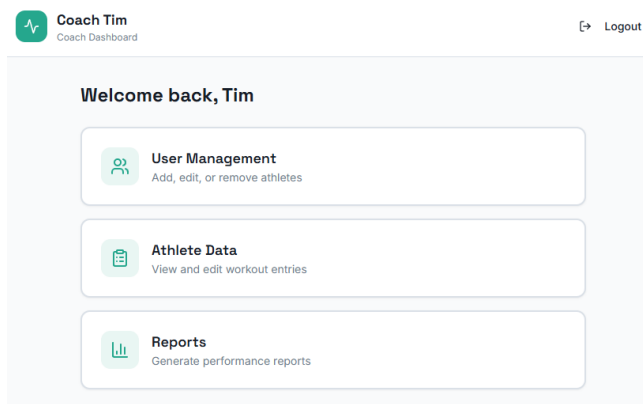


Figure 1 – The dashboard for app created by Lovable

Lovable was opinionated on the software stack. While Lovable, chose a good modern stack of React + Vite + Tailwind on the front end connecting to a Supabase backend using PostgreSQL DB, selecting a different stack would have required heavy customization. We were able to see and edit the code directly if needed. Overall, the web application was styled using different colors, curved boxes, and buttons using the Tailwind CSS library. It adhered closely to the requirements document, not adding additional functionality. In addition, the immediate cloud deployment makes it easy to get quick client feedback to iterate the prototype. Lovable created good documentation as part of the development, it displayed useful steps and enumerated and summarized what it was going to do. It was also helpful that datastores were seeded with sample data. With no installation and an easy-to-use website and documentation, the learning curve was very manageable. Lovable seemed to use good security practices by not displaying passwords and not storing them as text in the database. Lovable is proprietary, but low cost and free for students to use for prototyping. Lovable works well with Information Systems who typically have less programming experience.

Due to the off by one recording date error and the lack of aggregated reports, It was scored a 3 for functional. With the immediate deployment capabilities of the website, the lack of any installation requirements, and no need to specify AI models or API choices, we rated Lovable a 5 for usability. For technical, we rated Lovable a 4 because it implemented basic security practices but was quite rigid in the deployment language and tools. Lovable was scored a 3 for ecosystem because of the limited control of infrastructure and the lower level of flexibility in choosing coding languages and APIs. Lovable was ranked 5 for cost because we were allowed to get a working prototype up without paying any money. Lovable also matched our organizational needs because we have students with limited programming background and was therefore rated as 5.

Replit

Replit produced the most functional website of all the tools. A screenshot is shown in Figure 2. For example, instead of just showing data at the lowest granularity, it included overview summaries of athletes, which included total number of sessions, distance traveled, and average workout duration. This functionality contrasts with Lovable that did not aggregate any data, even on the report screens. Replit graphs were more interactive when you mouse over data points. Icons were also used making it easy to identify the different functionality. The web app not only used the list of requirements from the document, but also existing templates that were considered best-practice for similar types of apps. Replit also added a view of “recent activity” of workout entries from the athletes. These additional features alone added usefulness to the “Coach Dashboard”. Replit did, however, have some omissions. For example, it excluded the workout attributes of “max heart rate” and “resting heart rate” that were both listed in the requirements

document. In contrast, Lovable had correctly included max and resting heart rate. Also, on the line graph, workout dates moved from right-to-left on the x-axis as opposed to left-to-right, which is customary. Despite these omissions, we scored Replit a functional score of 5, the highest of any website.

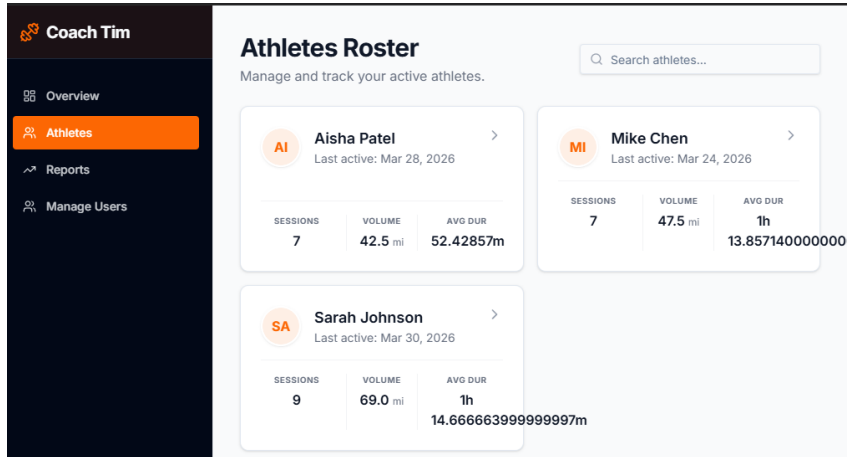


Figure 2 – Screenshot of the application created by Replit

Related to usability, Replit required no client-side installation, and it was easy to log onto the site using just Google Authentication. After uploading the requirements document and starting the AI, the app crashed after working for 8 minutes. Fortunately, Replit was able to restart the app, diagnose the error causes and fix the error. While we did not have to edit any code, the crash itself made the process seem more confusing than Lovable as we did not know whether a fix was going to be possible. Compared to Lovable, it was harder to find the actual code that was produced creating a steeper learning curve. Replit did create a DB backend immediately as opposed to waiting for us to prompt it. Replit had a very feature rich but complicated canvas editing tool. The user could draw on the screen using a mouse and Replit would translate the drawing into user interface components. It was able to use AI to take screenshots and then evaluate the picture to fix errors or implement a screen. Overall, Replit supported more design features than any of the tools we evaluated. Because of the crash, however, we rated Replit a 4 for usability.

Replit created a Typescript/NodeJS project by default but could have created a Python or PHP project if so desired. Replit included a persistent data store PostgreSQL by default and used hashed passwords and the crypt library for the security model. Lovable felt better for beginners because it had fewer options, but Replit felt stronger on the technical side and we rated it a 5. Replit also had better model selection tools compared to Lovable, but not as flexible as Cursor.

We rated Replit as 5 for ecosystem because of the greater flexibility in choosing AI models and platforms and APIs. There was no charge to create the prototype web application in Replit, so we rated it as 5 for cost. Also, because the app required no local installation and was able to repair errors, we rated Replit a 5 for organizational fit as well.

Cursor

Overall, Cursor created 10 PHP files, a total which is significantly smaller than the Lovable and Replit JavaScript solutions. The simplicity of the solution makes it easier for the developer to edit, but the user interface uses much less CSS and styling than the JavaScript solutions. A screenshot is shown in Figure 3. Also, the reporting features lacked visual representations. The Cursor solution did not include any graphs or charts to show workout totals or averages. The user interface felt very much like lists of data from tables as opposed to summarized views. The Cursor application felt very close to the requirements document as

it did not expand on functionality. The Cursor solution may be desired if developers know PHP and there is an existing web server environment that supports PHP. In addition, the Cursor app included a mistake. An attribute from the workout, listed as workout time in the requirements document, was supposed to be interpreted as workout duration. In the description of the attribute, it was described as the duration of the workout. Cursor used a date/time datatype to capture the data and therefore misinterpreted the duration. The attribute should have been stored as int and converted to minutes and seconds on the interface when needed. The cursor solution was awarded a score of 3 for functional.

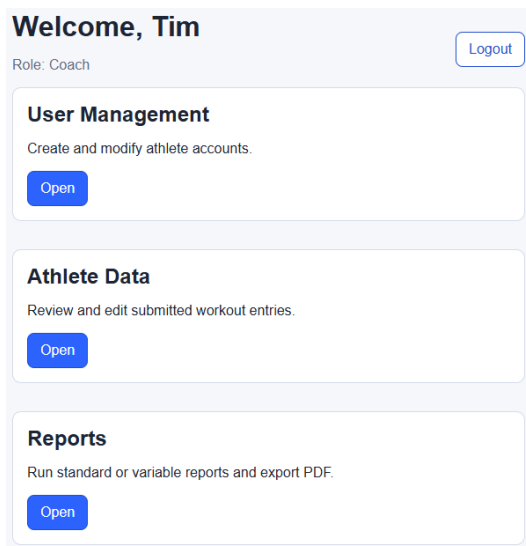


Figure 3 – The Cursor dashboard screen

Cursor is a fork of the Visual Studio Code codebase and therefore requires client-side installation of the integrated development environment (IDE). However, the installation is straight forward. Once installed, cursor gives the user many options of AI models to use. This greater AI model flexibility requires the user to be a bit more experienced. While the requirements document does not specify a programming language to use, Cursor found an XAMPP server on the client machine and implemented a PHP solution with a sqlite datastore. In addition, because Cursor is installed locally, it prompted every time it needed to run a command to create a folder or run the PHP debugger. This extra user involvement, including creating an initial environment for the project would be harder for a beginning programmer. To run the application, the local XAMPP server needed to be started, which also required some user experience. In addition, as the code is only running locally, it would need to be deployed to the Web for end users to evaluate the prototype. Due to these extra requirements, we awarded the Cursor solution a usability of 3.

Cursor did include a persistent data store (sqlite) compared to Lovable that initially only had RAM data storage. In terms of scalability, JavaScript solutions scale better than PHP, but the language choice may have been driven by the client environment more than a choice by the AI. Cursor implemented security using web sessions to keep users logged in. Passwords were hashed in sqlite. The solution did not implement much of a framework and therefore was less extensible but easily understood. Due to its lack of extensibility and lack of a framework, we awarded Cursor a technical score of a 3.

Cursor gave the user many choices regarding which AI models to use, i.e. GPT-5.5 (Open AI), Codex 5.3 (Open AI), Sonnet 4.6 (Anthropic), Opus 4.7 (Anthropic), Gemini 3 Flash (Google), and Composer 2 (Cursor). We liked this flexibility. We awarded Cursor a 4 for ecosystem.

Cursor was free for our limited chat agent usage, and it is free for students, so we awarded it a 5 for cost. As an editor, Cursor is friendly to programmers but may be intimidating to beginners. Because students in the SAD course have not had many programming courses, Cursor did not score as high on organizational fit as the other easier-to-use tools and was awarded a 3.

Claude Code

We used the Claude Code extension to Visual Studio Code as the fourth tool. Of the PHP solutions, Claude Code produced the most highly styled user interface and the most-organized source code. Claude Code used the Bootstrap CSS library to produce a styled user interface (shown in Figure 4) that nicely transitioned between overview and detail screens. There were navigational icons that helped the user stay oriented in the app. There was a nice dashboard with image icons that made it easy to see and understand the functionality of the app. The interface used a lot of JavaScript features despite being a PHP application, which made it look more modern and polished. These interface features were missing in the Cursor solution. Being just behind the Replit solution, we awarded the Claude Code solution a functional score of 4.

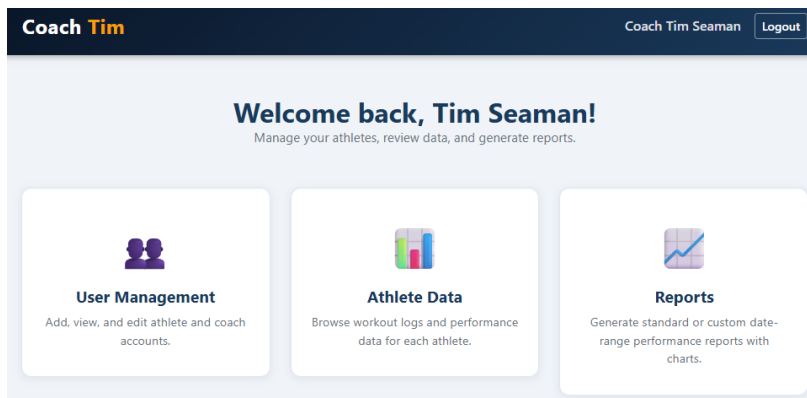


Figure 4 – The dashboard created by Claude Code

Setup and usability was more work with Claude Code. We initially installed the command-line tool as an agentic coding tool. Claude Pro, however, was required to use Claude Code as an extension in VS Code, which required a paid account. The complexity of linking our Claude Code account to VS Code was the most difficult setup of all the tools we used. Once setup, however, Claude Code worked very nicely and worked a lot like the Cursor tool. Because of the challenge in getting it set up, we awarded Claude Code a usability rating of 2.

While no formal framework was used, Claude Code produced code that was organized around the tables it created in MariaDB. In addition to the app, Claude Code created a setup script to initialize the MariaDB database and added seed data to all the tables. It added the most seed data of any of the tools we used. Unlike the other three tools, we did not find any errors or discrepancies from the requirements document. Claude really read the requirements document, seemed to understand it, and followed it closely. Claude used a good security model in the application, including using hashed passwords, session guards, and the use of prepared statements in SQL. Due to the lack of a framework, we awarded the Claude Code solution a technical rating of 4.

Unlike the other tools we evaluated that use AI models, Claude Code is made by Anthropic that specializes in creating AI models that assist programmers. Claude's models, for example, can be used within Cursor. In addition, Claude Code can be implemented as an agent to automate tasks, scan code repositories for changes or errors, and help deploy applications. Due to the greater number of areas where Claude Code can be used, we awarded it a 5 for ecosystem.

Because we needed a subscription to Claude Pro to use the Claude Code extension to VS Code, we gave Claude Code a cost rating of 3. While Claude Code has the power and flexibility many programmers have embraced, students in the SAD course do not have that level of expertise. Given that our target organization was the SAD course students, we awarded Claude Code an organizational score of 3.

Overall Results

Weighted Ranking Matrix for AI Dev Tools (1-5 scoring)

Criterion Fit	Weight	Lovable	Replit	Cursor	Claude Code
Functional	30%	3	5	3	4
Usability	25%	5	4	3	2
Technical	20%	4	5	3	4
Ecosystem	5%	3	5	4	5
Cost	10%	5	5	5	3
Organizational	10%	5	5	3	3
Weighted Total		4.1	4.75	3.25	3.35

As mentioned in the method section we created a weighted ranking matrix based on hands-on evaluations, prioritizing functional fit (30%) and usability (25%) given the course's focus on design rather than programming. Technical factors were weighted at 20%, cost and organizational fit at 10% each, and ecosystem considerations at 5%. The faculty raters felt that these weights made sense for a public regional university. What follows is a discussion of the matrix above.

Replit produced the strongest overall prototype, while Claude Code delivered the most robust PHP implementation. Lovable and Cursor were roughly tied for third place. But they all did an astonishingly good job. In terms of usability, Lovable was the easiest to learn, followed by Replit, then Cursor, with Claude Code presenting the steepest learning curve. From a technical standpoint, Replit again performed the best, followed by Lovable and Claude Code, while Cursor delivered only a basic, minimal application.

In terms of ecosystem and community, Replit has the largest and most mature user base, supported by extensive tutorials and resources on platforms such as YouTube and Stack Overflow. Claude Code also benefits from a large and rapidly growing community, distinguished by highly structured learning resources, including formal courses developed by Anthropic. Cursor's community is somewhat smaller but highly engaged, with support primarily through GitHub examples and a largely self-directed learning approach. Lovable currently has the smallest community, though it is growing quickly and offers a strong template gallery along with concise quick-start guides.

From a cost perspective, all tools were free to use for our SAD prototype exercise, except for Claude Code, which requires a \$20 per month subscription. Finally, in terms of organizational curriculum fit, both Replit and Lovable align well with our existing SAD course structure, whereas Cursor and Claude Code would require more significant adjustments to effectively integrate into the curriculum.

Thus, as seen in the matrix above, both Replit and Lovable emerge as the most suitable tools for students to use in our SAD course when developing prototypes.

Discussion

We were pleasantly surprised at the ability of generative AI tools to create high quality prototypes from a requirements specification document. Our discussion below highlights some key issues, including the need for high-quality requirements documentation, the ways in which GenAI-developed prototypes can enhance the SAD course, and ongoing concerns about the use of AI.

High Quality Requirements are important in GenAI Prototyping

It became apparent that the quality of the generated prototype relied on the quality of the RSD. Even with our high quality RSD, the input requirements still had some shortcomings that were reflected in the prototypes. While some GenAI tools that “thought” beyond the requirements developed more error free solutions, those that held tightly to the requirements produced the errors in the requirements.

Recently as vibe coding has matured past the early ad hoc or bank page stage of total freedom several authors are calling for a more structured approach to vibe coding development. DeepLearning.AI created a new course named “Spec-Driven Development with Coding Agents” because while vibe coding is fast it often produces output that is not what was expected. The course teaches how to “write clear specs that give your coding agent the context it needs to build intentional, maintainable software. ... the focus is on writing a clear, scoped, and tested spec that your coding agent can follow.” (DeepLearning.AI, 2026, p1)

A new book, *Beyond Vibe Coding* (Osmani, 2026) proposes to help us “learn how to formulate clear goals and constraints for the AI.” The author suggests that software developers be the architect and editor by translating “complex requirements into effective prompts or specifications for an AI assistant,” building domain expertise, learn system design, and focus on reviewing/testing/verification. He also points out the importance of using structure when implementing components from descriptions.

The book additionally includes a chapter on “AI-Driven Prototyping” that allows developers to move from concept to working model in hours instead of days, enabling faster and deeper user feedback. Prototyping using vibe coding can feel like brainstorming, high-level, fast, and conversational. Lastly, he points out the prototype is the first draft, the code must be hardened during the transition to production. (Osmani, 2026)

GenAI “Vibe” coded prototyping can have a positive impact on the SAD course

GenAI prototyping can positively impact the SAD course in several ways:

Emphasizes high-quality RSDs The effectiveness of the generated prototype depended on the quality of the requirements specification document, highlighting the need to emphasize the development of high-quality RSDs in the SAD course. Historically human programmers varied in how close they would follow RSDs, with AI, the ASD literally becomes the blueprint.

Better validation of design Prototyping is faster, more comprehensive, and more informative than relying on wireframe sketches alone. While there may still be value in creating wireframes and incorporating them into the generative prototyping process, generating prototypes directly from a RSD already offers great speed and depth of understanding. This approach enables students to evaluate and validate their designs more holistically and supports more realistic usability testing.

Iterations lead to higher quality AI has lowered the cost of iteration, making mistakes in requirements less consequential. As a result, refining prototypes has become more affordable, encouraging greater flexibility in adapting requirements. In the past, developers were often less receptive to user feedback, but easier iteration now supports continuous improvement and higher-quality outcomes. This aligns with James

Clear's emphasis on repetition and quantity as key to developing skills and achieving higher quality, suggesting that repeated prototyping can lead to more effective and polished system solutions. (Clear, ch11, 2018)

Increased student motivation Generative prototypes may motivate students to engage more deeply in the SAD course and take it more seriously. When students recognize the importance of crafting effective prompts by using industry-standard communication tools such as use cases, ERDs, and other structured analysis techniques, they can gain greater ownership of the development process. With AI, students increasingly take on the combo role of programmer/analyst. In the past, programming often limited creativity due to its complexity, but AI reduces that barrier, empowering analysts and designers. For example, in industry settings like Overstock, both product managers and coders are assuming the AI development role.

IS students can be good AI developers Information Systems students may be well positioned to become effective AI developers due to their strength in structured analysis. While IS programs emphasize business processes and domain understanding more than technical programming skills, AI tools can help bridge that gap by providing strong technological capabilities. As a result, IS students' expertise in business problem-solving and requirements specification becomes a significant advantage.

Creating usable products We have found that many of our students want to fully implement their projects. Another advantage of generating prototypes from structured requirements documents is the potential to produce a usable product more quickly, enabling faster and higher-quality evolution from prototype to fully developed system.

Concerns of Using AI in IS Education

Of course, there are valid concerns about using AI in information systems education including student overreliance on AI, reduced programming comprehension, and authorship and accountability issues. The growing use of AI prototyping tools introduces both opportunities and risks for student learning. On the positive side, AI tools shift the focus from syntax to higher-level thinking. Students can concentrate more on modeling problems, understanding processes, and translating requirements into structured representations that AI can use to generate code. This encourages deeper engagement with domain knowledge and reinforces the importance of clearly defined requirements. It also aligns with industry trends, where the ability to guide AI through precise specifications is becoming a valuable skill. Additionally, students can develop stronger quality assurance capabilities, as AI-generated outputs still require validation, debugging, and refinement. In this sense, AI use promotes skills in evaluation, testing, and iterative improvement rather than rote coding.

However, there are potential drawbacks. Heavy reliance on AI for code generation may reduce students' understanding of fundamental programming concepts, such as control flow, data structures, and system design. Without this foundation, students may struggle to diagnose errors, assess scalability, or adapt solutions beyond the AI's output. There is also a risk that students become overly dependent on AI tools, treating them as "black boxes" rather than engaging critically with the underlying logic of the systems they create. This could limit their ability to work effectively in situations where AI tools are insufficient, unavailable, or produce flawed results.

Implications for Programming Education in IS Programs

We argue that this study shows that programming education is being reshaped by AI tools that can generate and debug code, shifting the emphasis from learning syntax to understanding systems, processes, and requirements. We hypothesize that students will need stronger domain knowledge and the ability to model

applications clearly so AI can produce accurate solutions. Additionally, as AI-generated code is often imperfect, quality assurance and validation skills are becoming increasingly important. Thus, curricula should prioritize programming concepts, software design, and frameworks over learning multiple languages, while advanced courses focus more on building and deploying complete, scalable applications rather than writing algorithms by hand. These are issues yet to be studied and resolved.

Conclusion

This study explored how generative AI can support prototype development in Systems Analysis and Design courses, finding that tools like Lovable, Replit, Cursor, and Claude Code can produce functional prototypes that often exceed expectations, while also emphasizing that AI-supported prototyping increases the importance of structured requirements in SAD education.

GenAI prototyping can enhance the SAD course by reinforcing the importance of high-quality requirements specification documents, as better inputs lead to better prototypes. It enables faster, more detailed design validation compared to traditional wireframes, supporting more comprehensive evaluation and usability testing. The reduced cost of iteration encourages continuous refinement and higher-quality outcomes, while also increasing student motivation and engagement by giving them greater ownership of the development process. Additionally, IS students are well positioned to leverage these tools due to their strengths in structured analysis, allowing them to bridge gaps in technical skills. Lastly, generating prototypes from structured requirements can accelerate the creation of usable products, helping students more easily transition from design to implementation.

As for the GenAI tools we evaluated for a Systems Analysis and Design course in an IS program, Replit and Lovable are the most effective primary tools for student prototyping. Claude Code and Cursor seem better suited for more advanced, code-intensive courses. Replit balanced functionality, usability, and learning support. Lovable is the easiest to learn but lacks in scalability and technical rigor. Claude Code is the most capable but is harder to use. Cursor is also powerful but has a steeper learning curve and lacks learning resources.

Another way to frame this comparison is in terms of scenarios. When a developer is iterating closely with a client, Lovable and Replit are the strongest choices due to their speed and ease of refinement. If the goal is to build a more polished or technically impressive product, Replit and Claude Code are better suited. For developing a minimum viable product (MVP), Lovable and Replit again stand out because they enable rapid prototyping. Finally, if the developer wants greater control over the codebase and plans to manage their own hosting, Claude Code and Cursor are the most appropriate options.

Limitations

This study has several limitations. First, prototypes were generated from only one Requirements Specification Document (RSD), limiting generalizability. Future research should incorporate multiple RSDs to assess whether usable prototypes can be derived from a wider range of student work. This also suggests that prototype quality may depend on prompt construction and student expertise, which should be treated as variables in future studies. Second, the feasibility study did not include student feedback, restricting insight into additional user perspectives. Third, only four GenAI tools were evaluated; while these are widely used, other tools may yield different or improved results. Finally, given the rapid evolution of GenAI tool capabilities the identification of “best” tools is likely to change, potentially enhancing future prototype generation from requirements documents.

Future research

The next step is to address the limitations stated above. Future research should have students be able to choose GenAI tools (or even multiple tools and iterations) to create prototypes in the SAD course and document their real experiences to better understand which tools they selected, their reasons for doing so, and whether the outcomes were successful. The variables of RSD quality and student skill level should be added. More objective metrics could also be added to the weighted ranking matrix such as: % of implemented requirements, functional defects, development time, quality assurance, and code maintainability. This can then be used to enhance the SAD course in the future.

Another pathway for future research is to add the Solution Design Document (includes prototypes of a report, user flow diagram, wireframes of screens, help and error messages) to the Requirements Specification Document as input for the AI in generating the prototype. Will it make the prototype better? Will it constrain the AI so much that creativity is lost?

References

- AACSB (2026). AACSB Global Standards for Business Education (accessed June 2026). AACSB. <https://www.aacsb.edu/educators/global-standards>
- ABET Computing Accreditation Commission (2026). Criteria for Accrediting Computing Programs. ABET. <https://www.abet.org/accreditation/accreditation-criteria/>
- Borg, M., Bjarnason, E., & Hedin, F. (2025). Vibe coding and the new prototyping playbook. *IEEE Software*, 42(6), 12–16. <https://doi.org/10.1109/MS.2025.3597032>
- Bjarnason, E., Lang, F., & Mjöberg, A. (2023). An empirically based model of software prototyping: A mapping study and a multi-case study. *Empirical Software Engineering*, 28(5), Article 115. <https://doi.org/10.1007/s10664-023-10331-w>
- Bureau of Labor Statistics. (2025). *Computer systems analysts*. U.S. Department of Labor, Occupational Outlook Handbook. <https://www.bls.gov/ooh/computer-and-information-technology/computer-systems-analysts.htm>, (visited March 27, 2026)
- Clear, J. (2018). *Atomic habits: An easy and proven way to build good habits and break bad ones*. Penguin Publishing Group.
- Dorey, K., May, J., Ezell, J., & Lough, S. (2026). Hands-on ERP in the Introduction to MIS Course. *Journal of Information System Education*, 37(2), 205-217.
- DeepLearning.AI. (2026). *Spec-driven development with coding agents*. <https://learn.deeplearning.ai/courses/spec-driven-development-with-coding-agents/lesson/8bhg0p/introduction>, (visited April 16, 2026)
- DeepLearning.AI. (2026). *Move beyond vibe coding with spec-driven development*. <https://info.deeplearning.ai/move-beyond-vibe-coding-with-spec-driven-development-1>, (visited April 16, 2026)
- Go-UML. (2024, January 13). *A comprehensive guide to use-case-driven modeling: From diagrams to ERDs*. <https://www.go-uml.com/a-comprehensive-guide-to-use-case-driven-modeling-from-diagrams-to-erds/>

- Grove, S. (2025, July 11). *The new code* [Conference presentation]. AI Engineer World's Fair. <https://www.youtube.com/watch?v=8rABwKRsec4>
- Halaweh, M. (2025). AI and Machine Learning for Business and Information Systems Education Using KNIME. *Journal of Information Systems Education*, 36(4), 352-366.
- IS'97. (1997). *Model curriculum and guidelines for undergraduate degree programs in information systems*. ACM. <https://doi.org/10.1145/511965.512222>, (visited April 14, 2026)
- IS 2002. (2002). *Curriculum guidelines for undergraduate degree programs in information systems*. ACM. <https://doi.org/10.1145/2593309>, (visited April 14, 2026)
- IS 2010. (2010). *Curriculum guidelines for undergraduate degree programs in information systems*. ACM. <https://www.acm.org/binaries/content/assets/education/curricula-recommendations/is-2010-acm-final.pdf>, (visited April 14, 2026)
- IS 2020. (2020). *A competency model for undergraduate programs in information systems*. ACM. <https://www.acm.org/binaries/content/assets/education/curricula-recommendations/is2020.pdf>, (visited April 14, 2026)
- Javadi, E., Gebauer, J., & Tanner, S. (2025). Experiential and Peer Learning in an IT Project Management Course: Flipped Classroom, Concept Maps, and Project Dojos. *Journal of Information Systems Education*, 36(1), 13-24.
- Karpathy, A. (2025, February 2). There's a new kind of coding I call "vibe coding" [Post on X]. <https://x.com/karpathy/status/1886192184808149383>
- Kolb, D. (1984). *Experiential Learning: Experience as the Source of Learning and Development*. Prentice-Hall. <https://www.syndetics.com/index.php?isbn=0132952610/toc.html&client=uncwh&type=nl2>
- Osmani, A. (2026). *Beyond vibe coding*. O'Reilly Media. <https://www.oreilly.com/library/view/beyond-vibe-coding/9798341634749/>
- Steghöfer, J.-P., & Borg, M. (2026). An abstraction is worth a thousand vibes. *IEEE Software*, 43(1), 13–16. <https://doi.org/10.1109/MS.2025.3621786>