Low-code platforms for business applications

IntelliJ IDEA with GitHub Copilot and Claude

Felix Groß Faizan Lone Navid Naghizadeh

Winter semester 2025/26

Questions

Felix Groß

1. **Study Program:**
   Computer Science (Master)
2. **Software Experience:**
   1.5 years as working student
3. **Web Skills:**
   Basics in HTML, CSS, JavaScript
4. **Parts Written:**
   1. Introduction, 2. Tools, 3. Claude, 6. Evaluation, 7. Conclusion

Faizan Lone

1. **Study Program:**
   Data Science (Master)
2. **Software Experience:**
   6 months intern in React, 7 months research assistant in Computer vision and 7 months of full time experience in genAi RAG and mlops systems
3. **Web Skills:**
   React, ML/AI frameworks
4. **Parts Written:**
   4. Development Process

Navid Naghizadeh

1. **Study Program:**
   Business Informatics (Bachelor)
2. **Software Experience:**
   2 years of employment, 7 years of continuous non-academic learning
3. **Web Skills:**
   Mern Stack, ASP.net
4. **Parts Written:**
   5. Application

Contents

List of Figures

List of Tables

1 Introduction

Low-code and AI-assisted development tools have increasingly influenced how modern business applications are designed, implemented, and maintained. By lowering the barrier to entry for software development and accelerating routine engineering tasks, these tools promise shorter development cycles, faster prototyping, and improved developer productivity. At the same time, they raise important questions about software quality, maintainability, and the shifting role of human developers in the engineering process. Understanding both the benefits and limitations of such tools is therefore essential, particularly in contexts where correctness, security, and long-term evolution of systems are critical [17].

This report examines the use of AI-assisted tooling within the development of a project management web application, focusing on IntelliJ IDEA as the primary development environment and GitHub Copilot, augmented with Claude models, as the central AI support mechanism. Rather than treating these tools as abstract technologies, the report grounds its analysis in concrete development experience. The goal is not to evaluate AI tools in isolation, but to assess how they interact with established software engineering practices, architectural decisions, and human judgment during real development work.

The project was conducted in an iterative, contract-driven manner, with a clear separation between frontend, backend, and interface specifications. Throughout the development process, AI assistance was employed in multiple roles, ranging from code generation and refactoring to explanation and exploration support. This enabled a comparative assessment of different modes of AI use and their practical implications for usability, expressiveness, and reliability in software engineering tasks.

The structure of the report reflects this perspective. After introducing the development tools and AI models used, the report analyses the development process itself, focusing on how AI assistance shaped workflows, influenced design decisions, and introduced both efficiencies and risks. Subsequent sections discuss the application's architecture, core technologies, implemented features, and observed friction points. An evaluation chapter then examines functional completeness through a systematic use case analysis and assesses the quality of the generated code across dimensions such as project design, interface consistency, code smells, security, and data protection compliance. The report concludes with a reflective assessment of AI-assisted software engineering, emphasizing where such tools provide clear value and where careful human oversight remains indispensable.

2 Tools

This section presents the primary development tools used in the project, namely IntelliJ IDEA and GitHub Copilot. The following subsections describe each tool, outlining their core functionality, available editions, and the specific reasons for their selection in the context of this project.

2.1 IntelliJ IDEA

IntelliJ IDEA is a comprehensive integrated development environment (IDE) developed by JetBrains and widely used in professional software engineering. While it is best known for Java and Kotlin development, IntelliJ IDEA also supports a broad range of additional programming languages and technologies through built-in functionality and an extensive plugin ecosystem. The IDE is available in two editions: a free Community Edition and a commercial Ultimate Edition. The Community Edition focuses primarily on JVM-based development and offers a limited feature set, whereas the Ultimate Edition includes advanced tooling for web development, enterprise frameworks, and database management. For the purposes of this project, the Ultimate Edition was selected, as its extended functionality aligns more closely with the requirements of full-stack application development [12].

IntelliJ IDEA enhances productivity through smart code completion, advanced refactoring tools, comprehensive debugging and testing capabilities, and built-in Git integration. These features reduce errors and support maintainable code management, particularly in larger codebases [13].

A major advantage of IntelliJ IDEA Ultimate is its extensive support for full-stack web development, which was a key requirement for this project. TypeScript is fully supported with intelligent code completion, static error detection, and refactoring support. Backend development is facilitated through integrated support for Node.js and Express, while React development benefits from component navigation, JSX support, and props inspection tooling. The Ultimate Edition also includes built-in database tooling for direct connections to relational databases such as PostgreSQL, enabling SQL execution, table browsing, and schema management within the same environment [13].

IntelliJ IDEA also incorporates AI-based development assistance directly into the IDE. JetBrains provides an integrated AI Assistant and an AI coding agent designed to support code generation, explanation, and project-level analysis [14]. However, in the context of this project, the GitHub Copilot plugin was used instead of the built-in AI features. The rationale for this decision and the practical usage of the plugin are discussed in the subsequent chapter.

2.2 GitHub Copilot

GitHub Copilot is an AI-powered coding assistant developed by GitHub that integrates directly into a developer's code editor or IDE. It functions as a digital pair programmer by generating code suggestions, explaining existing code, and assisting with common development tasks. Rather than replacing human developers, Copilot is designed to reduce repetitive work, accelerate implementation, and support the exploration of alternative solutions. At all times, developers remain fully in control of what is accepted, modified, or rejected [9].

Copilot is available in both free and paid tiers, which differ in usage limits and available features. The free tier provides basic code completion and limited chat functionality, making it suitable for small projects or exploratory use. The premium tier increases usage limits and unlocks advanced features, including access to more capable large language models. For this project, the premium tier was selected because it supports models from the Claude family and offers a higher token budget, enabling longer and more complex interactions as well as experimentation with multiple approaches [11].

Within the premium tier, Copilot offers access to several underlying models optimized for different tasks. Developers can manually select the model to balance response quality, speed, and token consumption. Copilot can also be configured to work with externally hosted models in enterprise setups, making model selection an important aspect of adapting the tool to specific workflows [10].

When integrated into IntelliJ-based IDEs, Copilot operates directly within the development environment. As code is written, it analyses the current file and project structure to generate inline suggestions that appear as ghost text, which developers can accept, edit, or ignore. Beyond inline completion, Copilot provides a chat interface and supports multiple interaction modes, including code completion, edit mode for targeted refactoring, ask mode for explanations, and agent mode for multi-step tasks spanning multiple files [9].

Despite its strengths, Copilot has important limitations. Generated code may be incorrect, inefficient, or insecure if accepted without review, and Copilot lacks full awareness of runtime behaviour and project-specific constraints. Therefore, its output must always be critically evaluated, tested, and reviewed. When used as a supportive tool rather than a replacement for human judgment, GitHub Copilot can significantly enhance developer productivity while maintaining code quality and reliability [17].

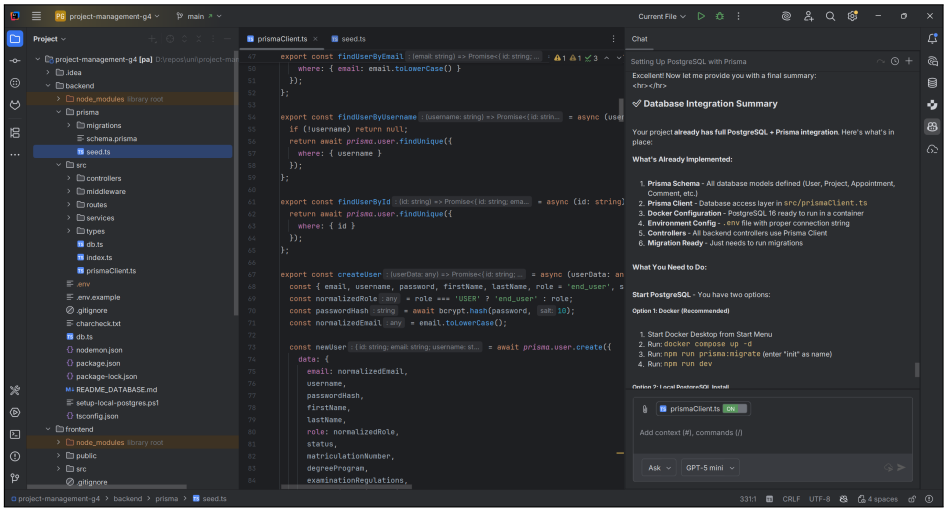

Figure 1: IntelliJ with GitHub Copilot

3 Claude

The following section provide an overview of Claudes core characteristics, available model variants, and a comparative evaluation of their suitability for different software engineering tasks within the scope of the project.

3.1 General Information

Claude is an LLM-based AI assistant developed by Anthropic, designed to support knowledge-based tasks with a particular emphasis on natural language understanding, structured reasoning, and coherent long-form responses. In software engineering, Claude aids developers in problem analysis, conceptual design, code comprehension, and technical communication. A central aspect of its design philosophy is the focus on reliability and interpretability. Unlike AI tools tightly integrated into specific IDEs or optimized primarily for code completion, Claude operates as a conversational system that can engage with broader problem descriptions, reason about abstract requirements, and generate explanations beyond individual code snippets [4].

Claude can maintain context over longer conversations, enabling complex interactions that go beyond simple prompt-response patterns. This makes it suitable for analytical and reflective tasks that benefit from step-by-step reasoning or multi-stage refinement. Overall, Claude is best characterized as a flexible, reasoning-oriented AI tool whose strengths in handling complex language-based tasks complement more automation-focused tools commonly used in development environments [4].

3.2 Model Overview

Claude is offered in several model variants that are designed to address different requirements in terms of performance, response quality, and computational efficiency. At the time of writing, the main publicly available variants are Claude Opus, Claude Sonnet, and Claude Haiku. These models follow the same underlying architectural principles but differ in scale and optimization goals, resulting in distinct trade-offs between reasoning capability, speed, and resource usage [4].

Claude Opus represents the most capable model in the Claude family. It is optimized for complex reasoning tasks, long-form analysis, and scenarios that require a high level of contextual understanding. Due to its larger model size, Opus is particularly suited for handling abstract problem descriptions, multi-step logical reasoning, and detailed explanations. However, these capabilities come at the cost of higher computational demands, which can result in slower response times compared to smaller variants [2].

Claude Sonnet is positioned as a balanced middle-ground model. It

aims to provide strong reasoning and language generation capabilities while maintaining more efficient performance characteristics than Opus. Sonnet is designed to handle a wide range of general-purpose tasks, making it suitable for situations where both quality and responsiveness are important. This balance makes it a versatile option for many software engineering-related use cases [3].

Claude Haiku is the most lightweight and performance-oriented model variant. It is optimized for speed and cost efficiency, prioritizing rapid responses over deep reasoning capabilities. As a result, Haiku is well suited for simpler tasks, short interactions, or scenarios where quick feedback is more important than exhaustive analysis. However, its reduced model complexity limits its effectiveness for tasks involving intricate reasoning or extended context [1].

The availability of these model variants allows users to select an appropriate trade-off between capability and efficiency depending on the nature of the task, rather than relying on a single, one-size-fits-all model.

3.3 Comparative Evaluation

While all Claude model variants share the same fundamental design goals, their practical behaviour differs noticeably. In terms of reasoning depth, Opus consistently demonstrates the strongest capabilities for multi-step thinking and nuanced interpretation, while Sonnet performs well with more concise reasoning and Haiku is limited in handling deeply nested or abstract problems. Response quality follows a similar gradient: Opus produces the most structured outputs, Sonnet favours compact but high-quality responses, and Haiku is direct but may lack contextual nuance. Performance-wise, Haiku excels in low-latency scenarios, Sonnet offers a balanced compromise, and Opus exhibits longer response times due to its computational complexity.

For software development tasks, conceptual problem solving and detailed explanations benefit from Opus or Sonnet, while simpler tasks such as quick clarifications are more efficiently handled by Haiku. Despite their strengths, all Claude models share limitations inherent to large language models: they may produce confident but incorrect statements and require human verification in technical contexts [15].

4 Development Process (Usability Analysis and Expressiveness of AI-Assisted Software Engineering)

This section reports the development process for the application and evaluates the usability and expressiveness of AI-assisted software engineering. The account is grounded in direct development experience and focuses on how the system was built, how AI was integrated into daily work, and how human judgment shaped correctness and coherence. The narrative preserves concrete examples from the project, including role constraints implemented through email lists, object comparison mistakes, admin password logic, a React rendering loop, an OpenAPI mismatch, and the limitations of a mock database, while reorganizing them into a submission-ready research section.

4.1 Development Workflow

The development workflow was iterative and contract-driven. It began with close reading of the requirements and the establishment of a domain model for users, projects, appointments, comments, and settings. These entities informed both the route structure and the data shapes in the OpenAPI specification. The OpenAPI contract was not treated as mere documentation; it functioned as an operational boundary that synchronized backend and frontend work. Each iteration followed a consistent sequence: clarify the requirement, define or revise the contract, implement or refine backend endpoints, wire the frontend views and services, and conduct manual validation. This cycle provided short feedback loops and made integration errors visible early. A key architectural decision was the use of a mock in-memory database. This reduced infrastructure overhead and accelerated prototyping, yet required manual logic to emulate SQL-like joins and constraints. The choice made early iterations faster but also introduced long-term limitations in query expressiveness and persistence, shaping the later assessment of technical debt.

4.2 Approaches to AI Use

Five distinct approaches to AI use were adopted and evolved over time. The first was *one-shot generation*, used primarily for scaffolding initial router files and controller stubs. This approach produced fast results but was vulnerable to convention drift when prompts lacked domain constraints. The second approach was *incremental prompting*, where the team iteratively refined prompts and outputs, especially when implementing rules that spanned multiple modules. This approach improved accuracy but increased cognitive overhead. The third approach was *supporting or pair-assistant usage*, which became the default for small, localised tasks such as renaming fields,

aligning error messages, or drafting helper functions. The fourth approach was *exploratory prompting*, used to surface alternative architectural choices or to clarify the structure of a feature before implementation. The fifth approach involved *AI for explanation and translation*, which helped interpret unfamiliar libraries, restate requirements as concrete tasks, and onboard team members into the codebase. The mix of approaches reflects a shift from speed-oriented usage to precision-oriented usage as the system's logic matured, a progression aligned with documented patterns in AI-assisted development where teams initially prioritise throughput before shifting toward verification and correctness as project complexity grows [7].

4.3 Usability Analysis

Usability was assessed through the lens of learnability, efficiency, error tolerance, and user satisfaction during development. Learnability was high because the tools accepted natural language prompts and required minimal configuration, consistent with large-scale surveys finding that developers are most motivated to use AI programming assistants to reduce keystrokes and finish tasks quickly [16]. Efficiency gains were clear for repetitive or template-like tasks, such as generating CRUD routes and standard service functions. However, error tolerance was low in context-heavy tasks, where subtle logic issues were introduced and only surfaced during review. Satisfaction was high when the output could be verified at a glance, but it decreased when review required deep semantic checks across modules. The usability of AI assistance thus depended on task structure: it felt smooth when the task was narrowly scoped and aligned with common patterns, but frustrating when the task depended on project-specific rules or multi-file coordination [16]. The experience suggests that usability in AI-assisted development is not only a property of the interface, but also a property of how well the task can be constrained and verified.

4.4 Expressiveness of Prompts

Expressiveness concerns how well the prompt captures developer intent and constrains the output. Vague prompts were consistently less reliable because the AI defaulted to generic patterns. Detailed prompts that included data shapes, field names, and explicit constraints produced substantially better outputs [19]. For example, when prompts stated that managers and developers are stored as email lists and that a user cannot be assigned to both roles, the resulting logic was closer to correct. When that constraint was omitted, the AI compared entire objects rather than identifiers, a mistake that would have allowed duplicates to slip through. This observation led to an explicit recognition of *prompt literacy* as a practical skill. Writing effective prompts required articulating invariants, edge cases, and integration points in a way

that the AI could operationalise [19]. The act of prompt construction often clarified the requirement itself, revealing ambiguities before code was written. Over time, prompt design became comparable to writing good test cases: it demanded precision and a clear model of the system.

4.5 Prompt Engineering Examples

The following three examples illustrate how prompts were iteratively refined to produce correct, production-ready outputs. Each example shows the initial vague prompt, the AI's mistake, intermediate refinements, and the final working prompt that resolved the issue [19].

Example 1: Adding Routing to a React App

Prompt 1:

```
add routing to my react app
```

AI Mistake: The AI installed `react-router-dom` but placed all routes directly in `main.tsx`, which was difficult to manage.

Prompt 2:

```
This is hard to manage. I want a main layout
   component
that has a navbar and a footer, and the actual page
content should be rendered in the middle.
```

Final Working Prompt:

```
1. Install 'react-router-dom'.
2. Create a 'layouts' folder and inside it, a
   'MainLayout.tsx' component. This component should
   render an 'Outlet' from 'react-router-dom'.
3. In 'App.tsx', set up the browser router. Create
   routes
   for '/', '/login', '/register', and '/projects'.
      All
   of them should be nested under the 'MainLayout'
      component.
4. Create a 'pages' folder and create placeholder
   components
   for 'HomePage', 'LoginPage', 'RegisterPage', and
   'ProjectsPage'.
```

Example 2: Projects Page with Loading and Error States

Prompt 1:

```
create a page that shows a list of projects
```

AI Mistake: The AI created a static list, not one fetched from an API.

Prompt 2:

```
The projects should be fetched from my '/api/
   projects'
endpoint. Use the 'useEffect' hook to fetch the data
    when
the component mounts. Remember to include the JWT
   from
'localStorage' in the 'Authorization' header.
```

Prompt 3:

```
The page is blank until the data loads, which looks
   bad.
How do I show a loading message? Also what if there'
   s
an error?
```

Final Working Prompt:

```
Create a 'ProjectsPage.tsx'.
1. Create state variables for 'projects', 'loading',
    and
   'error' using 'useState'.
2. In a 'useEffect' hook, set 'loading' to 'true'.
3. Fetch data from the '/api/projects' endpoint,
   including
   the auth token in the header.
4. If the fetch is successful, update the 'projects'
    state
   with the data and set 'loading' to 'false'.
5. If there's an error, update the 'error' state and
    set
   'loading' to 'false'.
6. In the JSX, render a loading indicator if '
   loading' is
   true, an error message if 'error' is set, and the
       list
   of projects otherwise.
```

Example 3: Restricting `getAllUsers` to Admins Only

Prompt 1:

```
Create the 'userController.ts' file. Add a function
'getAllUsers' that finds and returns all users from
   the
in-memory database.
```

AI Mistake: The AI created the function, but it was completely public with no access control.

Prompt 2:

```
My 'getAllUsers' function is working, but I just
   realized
any logged-in user can call it. This data should be
   for
admins only. How do I restrict access?
```

AI Mistake: The AI suggested creating complex, reusable role-checking middleware, which was overkill for the immediate need.

Prompt 3:

```
That seems too complicated right now. Can I just
   check the
user's role inside the controller function itself?
   My 'auth'
middleware already adds the user object to the
   request.
```

Final Working Prompt:

```
In 'userController.ts', update the 'getAllUsers'
   function.
At the very beginning, add a check:
'if (req.user.role !== 'admin')'. If the user is not
    an
admin, return a 403 Forbidden status immediately.

Now, add two more functions: 'updateUserRole' and
'updateUserStatus'. Both should take 'userId' from
'req.params'. They also must include the same check
   to
ensure only an admin can perform these actions.
```

4.6 Where AI Worked Well and Where It Failed

AI performed well in tasks with predictable structure and limited scope. It generated standard Express routers for CRUD operations, drafted controller skeletons consistent with existing conventions, and produced small utility functions for list membership and normalisation. It also assisted in drafting documentation that could then be refined into a coherent narrative. These outcomes were reliable because they matched common development patterns and did not require deep knowledge of domain-specific rules [21].

Failure cases clustered around business rules, security-sensitive logic, and cross-module coordination. The role constraint example illustrates this boundary: a prompt to prevent a user from being both manager and developer led the AI to compare objects by reference rather than comparing email fields, violating the intended rule. A similar failure appeared in admin password logic, where an AI-generated update path collapsed administrator actions and ordinary user updates, thereby weakening authorisation boundaries — a pattern consistent with empirical studies finding that roughly 30% of AI-generated code snippets contain security weaknesses spanning multiple vulnerability categories [8]. Controlled experiments further confirm that developers using AI assistants are more likely to produce insecure solutions and often remain confident in flawed outputs [18]. Another failure occurred in a React form where the AI inserted a `useEffect` dependency on a state object, producing an infinite render loop that would have manifested as runtime instability. Cross-module logic failed when a user profile field was added: the AI updated backend types but did not update the OpenAPI contract or the frontend state, creating a mismatch that only surfaced during integration tests. Additional failures emerged during real integration work in the commenting feature. An AI-suggested handler accepted project identifiers without verification, which allowed comments to be created for nonexistent project IDs; during testing we found comments that never appeared in project views because they were not linked to valid project data. The fix required explicit existence checks before creation and a consistent join between project lookups and comment storage. These examples demonstrate that AI is fragile when correctness depends on subtle invariants, security boundaries, or coordination across the backend, contract, and frontend [8].

4.7 Manual Effort and Human Role

Manual work remained indispensable throughout the project. Requirements interpretation could not be delegated because the AI lacked access to the full contextual intent behind the specification. Human review of AI output was mandatory, especially for access control, data consistency, and error handling. Integration and debugging required running the system and observing runtime behaviour, which AI assistance cannot perform. Architectural de-

cisions, such as choosing a mock database to accelerate early development, were also human judgements that balanced speed against long-term maintenance. Manual effort was additionally required to keep the OpenAPI contract synchronised with implementation changes and to ensure that frontend and backend type definitions remained aligned. These activities demanded a global view of the system and an understanding of side effects across modules, reinforcing that AI can assist but not replace a developer's reasoning [7]. Hybrid techniques that combine traditional software engineering practices with AI-driven approaches are therefore necessary for maintaining system correctness [7].

4.8 Reflective Assessment and Implementation Status

The reflective assessment positions AI as a productivity multiplier rather than an autonomous developer [20]. The multiplier effect was strongest when tasks were decomposed into small, well-specified units. Trust calibration emerged as a critical practice: early overreliance on AI led to latent defects, whereas later iterations applied stricter review standards to high-risk logic. This shift produced a review culture that treated AI output as a draft, not a decision. Integrating AI into the software development lifecycle requires a measured approach that adapts traditional practices such as code reviews to keep humans in the loop [20]. The result was a more deliberate process in which explicit constraints and assumptions were articulated before code changes were accepted.

Implementation status at the end of this phase can be described as a functional prototype with substantial coverage of core features. Authentication and role-based access are in place, project management flows support create, view, update, and delete operations, and appointment and comment workflows are functional. Settings and profile updates are integrated into the user experience. The easiest components to build were those aligned with standard web patterns, such as CRUD routes and form submissions. The most difficult components were those involving multi-entity constraints and multi-module alignment, especially in the presence of the mock database. The in-memory store accelerated early progress but limited expressive querying and required custom logic to emulate relational behaviour, which introduced complexity and potential for error. These outcomes reinforce the earlier usability and expressiveness observations: AI accelerates routine work, but system correctness depends on careful human oversight and stable contracts.

In conclusion, AI-assisted software engineering proved valuable for speeding routine development tasks and for reinforcing consistency across scaffolding, yet it showed clear limitations in business rules, security, and cross-module logic. Usability was high for narrow tasks and low for context-heavy tasks [16], while expressiveness depended on prompt specificity and the stability of the project's contracts [19]. The experience suggests that the most

effective use of AI is within a disciplined workflow that emphasises explicit constraints, systematic review, and a shared understanding of domain rules. This project demonstrates that such a workflow can yield a robust prototype while also revealing the boundaries of what AI can safely automate in contemporary software engineering [20].

5 Application

The software industry has long rewarded fast delivery. Time-to-market is often treated as a decisive competitive factor, particularly in environments where requirements are volatile and user feedback is expected to guide further development. Against this backdrop, it is not surprising that so-called vibe coding gained popularity as soon as suitable tooling became available. These tools allowed teams to move quickly from idea to implementation, lowering the threshold for experimentation and early validation. One immediate consequence was that minimum viable products could be produced in a matter of hours rather than weeks. For businesses, this represented a significant shift. Instead of investing heavily upfront in planning and architectural refinement, it became feasible to ship something early and refine it later. This acceleration, however, came with trade-offs. While initial delivery was faster, the long-term iteration cycle often slowed down as technical debt accumulated. This tension between short-term speed and long-term maintainability will be discussed in a later section. For now, the focus lies on the conditions that enabled fast delivery before the widespread adoption of AI-based development tools. Understanding how speed was achieved in the pre-AI era is essential in order to distinguish genuinely new effects introduced by AI from patterns that already existed. Many of the practices associated with rapid development today did not emerge with AI, but were instead amplified by it. Examining these earlier practices provides a necessary baseline for evaluating what has actually changed.

5.1 Core Technologies

5.1.1 Express.js

At the beginning of the seminar, I was openly skeptical about the choice of Express.js as the backend framework. My primary concern was its lack of built-in abstractions and conventions. As an unopinionated framework, Express does not impose a default project structure, which I initially assumed would blur responsibilities and slow development rather than support it.

Over time, this assessment shifted. Express has long been common in startup environments precisely because it minimizes the amount of conceptual overhead required before productive work can begin. Developers are not required to internalize a rigid framework philosophy, which lowers the entry barrier and allows features to be implemented immediately. In fast-paced contexts, this flexibility can outweigh the absence of enforced structure.

I also underestimated the strength of the surrounding ecosystem. Express integrates seamlessly with the broader npm landscape, enabling teams to introduce middleware and supporting libraries incrementally instead of committing to an architecture upfront.

In the context of AI-assisted development, Express amplifies both strengths and weaknesses. Its minimalism enables rapid generation of routes and controllers, but it also allows hastily made architectural decisions to persist. AI does not create this dynamic; it merely accelerates it. The result is faster progress in the short term and a correspondingly higher risk of accumulated technical debt.

5.1.2 Typescript

At first glance, choosing TypeScript over plain JavaScript seems at odds with rapid delivery. JavaScript requires no explicit data models or contracts, making it attractive in early prototyping phases. Introducing a type system can therefore appear to impose unnecessary friction.

Despite this, TypeScript was selected as a structural constraint within an AI-assisted workflow. JavaScript relies on implicit assumptions about data shapes and return values—assumptions that are easily violated, particularly when code is generated or modified by AI. TypeScript forces these contracts to be made explicit and exposes inconsistencies that might otherwise remain hidden.

In practice, it functioned less as a guarantee of correctness and more as an early diagnostic layer. While AI-generated code was often syntactically valid and type-safe, the resulting types were sometimes overly permissive and required manual refinement. TypeScript thus served primarily as a mechanism for revealing structural weaknesses during rapid iteration, rather than eliminating them outright.

5.1.3 React

React was selected as a familiar baseline in modern frontend development. Its component-based model supports incremental implementation, aligning well with tight time constraints and AI-assisted iteration. Components can be introduced or replaced with minimal friction.

However, React imposes few architectural constraints. While this flexibility enables rapid progress, it also allows structural inconsistencies to emerge when AI-generated components accumulate without a coherent design. React does not resolve such issues; it simply renders them visible as the application grows.

5.2 Auxiliary Technologies

5.2.1 Validation

API specification files define the contractual interface between client and server. Processing requests that do not conform to this contract is ineffi-

cient and potentially error-prone; schema validation is therefore a necessity. Invalid data should be rejected at the boundary.

In the JavaScript/TypeScript ecosystem, Zod and express-validator represent two common approaches. Zod is transport-agnostic and allows schemas to validate HTTP payloads, internal data, or inter-service communication alike. Express-validator is tightly coupled to the Express middleware pipeline, enforcing request constraints through chained, field-level checks.

While technically sound, this fragmented validation style feels inelegant to me, as individual fields are elevated to the same structural level as controller logic. Zod's schema-centric model, by contrast, expresses validation as a cohesive unit. Although Zod can derive TypeScript types directly from schemas, I did not consider this decisive; I prefer defining request types explicitly in TypeScript for clarity.

What might be labeled verbosity is, in this context, deliberate explicitness. Clear schemas surface assumptions and reduce implicit knowledge. In an AI-assisted workflow where boilerplate is cheap, the primary constraint is no longer code volume, but comprehensibility.

5.2.2 ORM

An object–relational mapper mediates between application logic and persistent storage. Its function is not simply to abstract SQL, but to translate between in-memory models and database structures while confining persistence concerns to the infrastructure layer. In a disciplined architecture, the ORM should support the domain, not define it.

From this perspective, Prisma is preferable to TypeORM. TypeORM's decorator-based model embeds persistence metadata directly into domain classes, entangling infrastructural and domain concerns. Prisma externalizes this information into a dedicated schema, leaving domain types as plain TypeScript constructs. This clearer separation preserves conceptual boundaries and renders the persistence layer explicit and replaceable—an architecturally superior property.

5.3 Implemented Features

5.3.1 Dual Authorization

Not all use-cases exposed by an application are meant to be universally accessible. Certain operations are restricted to specific categories of users, which introduces the need for authorization at more than one conceptual level. In the application developed during the seminar, this resulted in a clear separation between two forms of authorization. The first operates at the application boundary, before any orchestration or domain logic is executed. This layer is concerned with general access rights: whether a request is allowed to proceed at all. At this level, permissions are typically more

appropriate than roles, as they allow for finer-grained control and reduce the risk of coupling authorization logic too closely to domain concepts. The second form of authorization occurs within the domain layer itself. Even when a request is structurally valid and generally authorized, it may still violate domain-specific access rules. In such cases, the rejection is not an infrastructural concern but a consequence of insufficient domain permissions. This distinction is important, as it prevents the conflation of domain roles with application-level authorization roles—a mistake that can lead to rigid and misleading models. One aspect that requires particular attention is the API specification. If access rules are implicitly encoded or poorly expressed at the contract level, developers may be forced to reintroduce this conflation unintentionally. Maintaining a strict separation between general authorization and domain-specific access control therefore remains both a design responsibility and a documentation concern.

5.3.2 Email

The application includes the capability to send emails, to send invites to people to join a project. This functionality was intentionally implemented as a one-off service rather than as a formalized contract or subsystem. Modeling email sending as a contract-bound interaction would have introduced additional abstractions—interfaces and message schemas—that were difficult to justify given the limited scope and usage frequency. Instead, the email service is invoked explicitly at a single well-defined point in the application flow, with a narrow and clearly bounded responsibility. This design choice also aligned well with AI-assisted development. The AI tools were reasonably effective at generating boilerplate for API-based email providers, but far less reliable when asked to reason about long-lived contracts or failure semantics. Treating email as a one-off side effect therefore reduced architectural overhead while keeping the implementation comprehensible. In this case, restraint proved more valuable than generality.

5.3.3 CRUD Operations

At the most fundamental level, the application exists to record, modify, and retrieve data. Everything else—authorization, validation, notifications—derives its relevance from this core capability. Unsurprisingly, CRUD operations therefore form the backbone of the system and constitute the most complete and stable part of the implementation. While AI-assisted tooling proved effective at scaffolding these operations quickly, it also exposed a recurring issue: even under supervision, the AI tended to conflate artifacts. Data access logic, domain rules, and transport-layer concerns were frequently interwoven unless explicitly separated by the developer. This reinforced the observation that, although AI can accelerate the construction

of data-centric features, it does not inherently respect architectural boundaries. Ensuring that CRUD functionality remained coherent and layered was thus less a matter of generation speed and more a matter of continuous correction.

5.4 Architectural Friction Points

5.4.1 Contracts for Communication with Outside Systems

Communication with external systems exposed one of the more brittle edges of AI-assisted development. When prompted to integrate with outside services, the AI consistently gravitated toward concrete, prototype-like implementations rather than explicit contracts. Instead of modeling the interaction through stable interfaces or versioned specifications, it tended to hardcode assumptions about behavior, payload structure, and response semantics directly into the implementation. This approach is particularly problematic in the presence of evolving APIs. Once an external system introduces a new contract version, such prototype implementations become obsolete almost immediately. Retrofitting them is rarely straightforward, as the original code was not designed with adaptation in mind. Decorating or incrementally extending the existing implementation to support a transitional phase often proves infeasible, precisely because no clear contract boundary was established in the first place. As a result, integration code generated in this manner exhibits a form of temporal fragility: it works convincingly at the moment of creation, yet degrades rapidly as soon as the surrounding ecosystem changes. This reinforces the need for developers to assert contractual discipline explicitly, rather than relying on AI tooling to infer or preserve it.

5.4.2 Weak Enforcement of Artifact Boundaries

One of the subtler yet persistent issues observed during AI-assisted development was the weak enforcement of artifact boundaries. The tendency to conflate layers became particularly visible when controllers absorbed responsibilities that properly belonged elsewhere. For example, data access logic frequently appeared inline within route handlers, a minor architectural blemish, but not the most concerning one. A more serious manifestation arose during validation. Domain-level checks often appeared directly alongside authorization or transport-layer validation within the same controller. Consider the “List Comments” endpoint for a project: both domain-specific rules and general authorization logic were interleaved, creating a tangled web of responsibilities. This layering misalignment is far from best practice, as it undermines maintainability, testability, and clarity of intent. While AI can reliably generate syntactically correct code, it does not inherently respect conceptual boundaries between artifacts. Left unchecked, this leads

to controllers and modules that are functionally complete but structurally diluted—a recurring friction point in AI-assisted low-code development.

6 Evaluation

This section evaluates the outcomes of the AI-assisted development process by examining both functional completeness and code quality. The evaluation is structured into two main parts. First, a use case analysis assesses the implementation status of all functional and non-functional requirements, identifying which features were successfully implemented, which required workarounds, and which remain incomplete. Second, a systematic code quality review examines the generated application across multiple dimensions, including project design, interface consistency, code smells, security considerations, and data protection compliance.

6.1 Use Case Analysis

The user management module consists primarily of standard CRUD operations that were straightforward to implement with AI assistance. However, the user invitation feature (5.1.2) could only be partially implemented due to challenges in configuring an external mail service. Attempts to set up SMTP integration with Claude's guidance proved problematic, requiring extensive configuration knowledge beyond what the AI could reliably provide. As a workaround, the invitation system was implemented using Mailtrap, a development email testing service, which simulates the full email delivery process without requiring production mail server configuration. This approach fulfills the functional requirements for development and testing purposes.

Table 1: User Management

ID	**Use Case**	**Status**
5.1.1	Log in to the server	Fully Implemented
5.1.2	Invite user	Partially Implemented
5.1.3	Accept invitation	Fully Implemented
5.1.4	Display all user accounts	Fully Implemented
5.1.5	Filter displayed user accounts	Fully Implemented
5.1.6	Display details of a user account	Fully Implemented
5.1.7	Change your own user account	Fully Implemented
5.1.8	Change another user's account	Fully Implemented
5.1.9	Delete a user account	Fully Implemented
5.1.10	Anonymize a user account	Fully Implemented

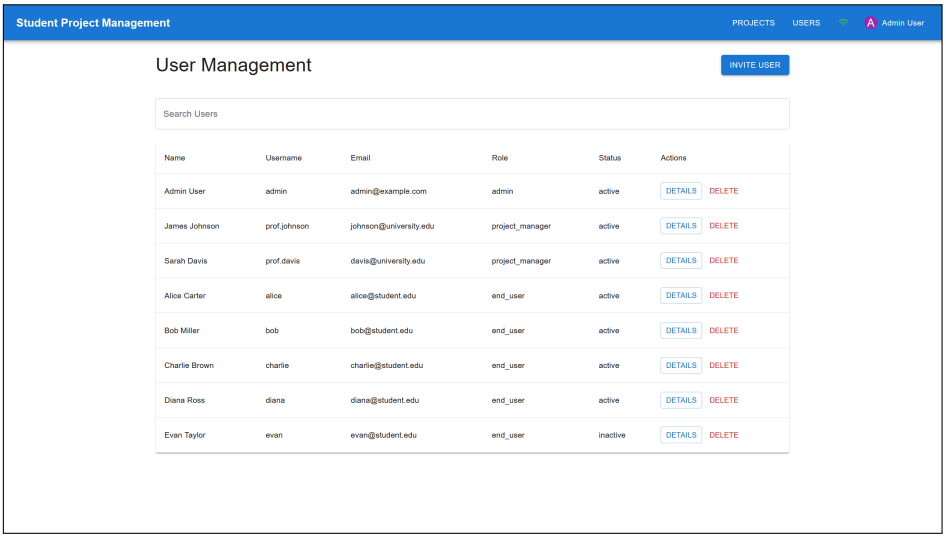

Figure 2: User Management

The project management functionality was fully implemented using standard CRUD operations. The relatively simple data model and well-defined requirements made this module particularly well-suited for AI-assisted development. All seven use cases were completed without significant obstacles, demonstrating that AI tools excel at generating boilerplate code for common database operations, RESTful endpoints, and basic business logic when requirements are clearly specified.

Table 2: Project Management

ID	**Use Case**	**Status**
5.2.1	Create project	Fully Implemented
5.2.2	Show all projects	Fully Implemented
5.2.3	Display own projects	Fully Implemented
5.2.4	Filter displayed projects	Fully Implemented
5.2.5	Display project details	Fully Implemented
5.2.6	Change project data	Fully Implemented
5.2.7	Delete project	Fully Implemented

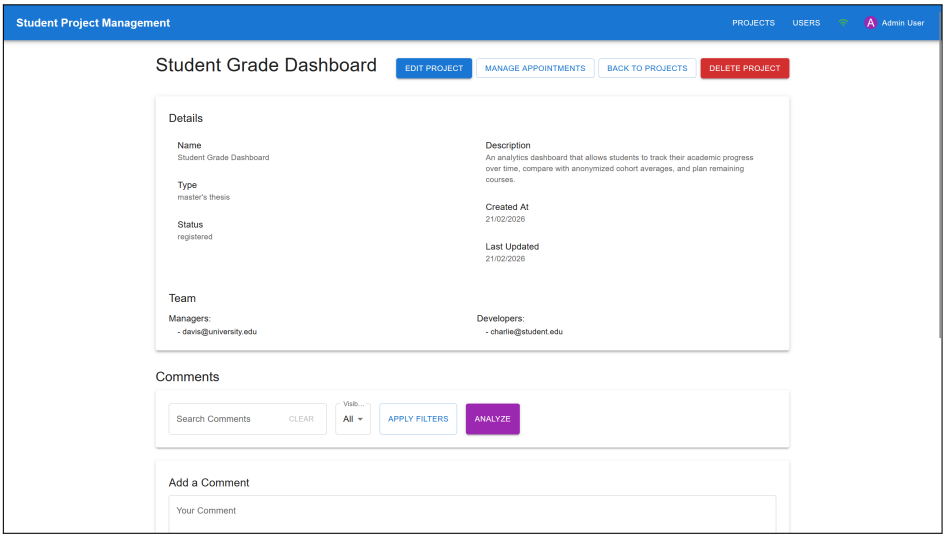

Figure 3: Project Overview

The appointment management followed a similar pattern to other CRUD-based modules, with most functionality implemented without difficulty. The exception was calendar integration (5.3.3), which proved challenging to fully implement. Direct integration with external calendar systems would have required complex authentication flows and API integrations for multiple calendar providers (Google Calendar, Outlook, Apple Calendar), which exceeded the scope of the AI-assisted development. Instead, the application generates standard iCalendar (.ics) files for each appointment, allowing users to manually import events into their preferred calendar application.

Table 3: Appointment Management

ID	**Use Case**	**Status**
5.3.1	Create appointment	Fully Implemented
5.3.2	Display all appointments for a project	Fully Implemented
5.3.3	Add appointment to calendar	Partially Implemented
5.3.4	Change appointment data	Fully Implemented
5.3.5	Delete appointment	Fully Implemented

The commenting module implementation highlights both the strengths and limitations of AI-assisted development. Standard commenting features (5.4.1–5.4.7) were fully implemented, including visibility controls distinguishing public and private comments. However, the AI-powered comment analysis feature (5.4.8) reveals a notable limitation in AI development guidance. During implementation, Claude recommended integrating the OpenAI API and indicated that a free tier was available. After completing

the integration, it became apparent that OpenAI had discontinued its free tier, rendering the feature non-functional without a paid API key. The implementation itself is technically complete and will function correctly if a valid API key is provided. Otherwise the application displays mock analysis results. This case illustrates how AI assistants may provide outdated information about third-party services and highlights the need for developers to independently verify external dependencies and their pricing models.

Table 4: Commenting

ID	Use Case	Status
5.4.1	Write public comment	Fully Implemented
5.4.2	Write private comment	Fully Implemented
5.4.3	Display all comments of a project	Fully Implemented
5.4.4	Display public comments of a project	Fully Implemented
5.4.5	Filter displayed comments	Fully Implemented
5.4.6	Change own comment	Fully Implemented
5.4.7	Delete comment	Fully Implemented
5.4.8	Analyze the comments of a project	Partially Implemented

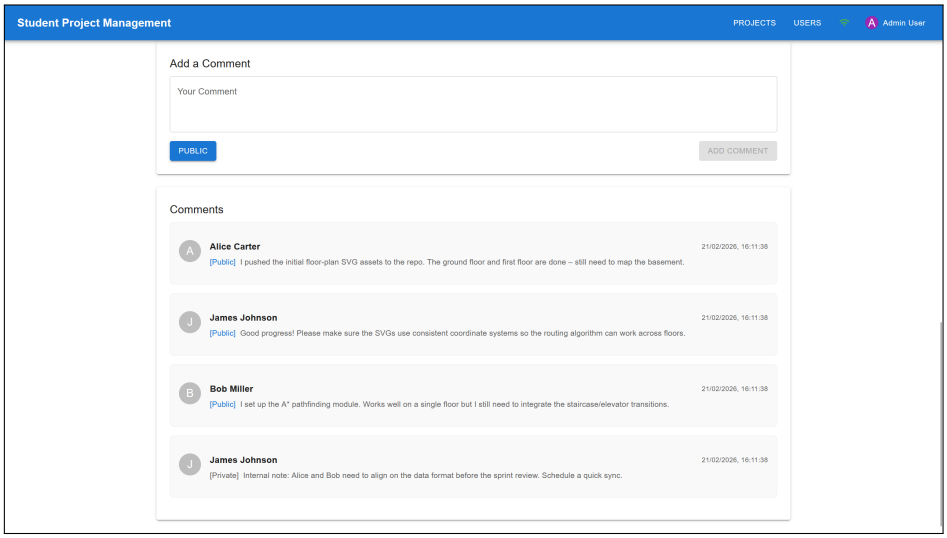

Figure 4: Comments

The settings module, consisting of a single use case for configuring connection interrupt detection intervals, was fully implemented without complications.

Table 5: Settings

ID	Use Case	Status
5.5.1	Change the period for detecting connection interrupts	Fully Implemented

All non-functional requirements were successfully addressed in the implementation. The application maintains reliable WebSocket connections with appropriate error handling and reconnection logic (6.1), provides consistent user feedback for all operations through notifications and loading states (6.2), and implements a functional user interface (6.3). While the UI/UX quality reflects that of a typical web application rather than a highly polished commercial product, all user interactions are intuitive and clearly communicated. Further refinement of visual design and interaction patterns would enhance the user experience but was not prioritized given the project's focus on evaluating AI-assisted development workflows.

Table 6: Non-functional Requirements

ID	Use Case	Status
6.1	Reliability of communication connection	Fully Implemented
6.2	Feedback to the user	Fully Implemented
6.3	User interface	Fully Implemented

6.2 Code quality

The overall code quality of the generated application is functional but inconsistent. Claude produced a working full-stack system with sensible library choices and a recognizable layered structure. However, quality attributes that go beyond making features work such as type safety, test coverage, and avoiding duplication were largely neglected. This is characteristic of AI-assisted development where prompts focused on feature implementation rather than engineering discipline. Claude delivered the working features but did not proactively introduce practices like testing, strict typing, or DRY principles unless explicitly instructed to do so.

Table 7: Overall Code Quality

Aspect	**Rating**	**Comment**
Functionality	Good	Core features work: auth, projects, appointments, comments, AI analysis, user management
Project Structure	Fair	Layered, but boundaries are blurred
Type Safety	Poor	Pervasive use of `any` defeats the purpose of TypeScript
Error Handling	Fair	Inconsistent, some routes handle errors, others let them fall through
Testing	Missing	Zero test files, `package.json` test script is a placeholder
Code Duplication	Poor	Massive duplication across routes, controllers, and frontend pages
Documentation	Fair	OpenAPI spec exists, but inline code comments are mostly trivial or auto-generated
Frontend State Management	Fair	All state in localStorage and local component state, no global state management
Dependency Management	Good	Modern, well-chosen libraries (Prisma, MUI, Vite, express-openapi-validator)

The project design shows a reasonable high-level architecture with an API-first approach, role-based access control, and a layered backend. These are likely the result of good initial prompting and Claude following established Express.js conventions. However, the design degrades at a deeper level. The data access layer is a single monolithic file, there is no proper service layer, and multi-step database operations lack transactions. This pattern is typical for AI-generated code, which tends to produce a workable architecture in early iterations but accumulates structural debt as features are added incrementally through successive prompts. Each new feature was

implemented in the most direct way possible rather than refactoring existing structure, because the AI optimizes for fulfilling the current request, not for long-term maintainability.

Table 8: Project design weaknesses

Weakness	**Details**
No service layer	Controllers call `prismaClient.ts` directly. Business logic (e.g., invitation acceptance, project membership validation) is scattered across `prismaClient.ts`, controllers, and route handlers.
Mixed responsibility in routes	Route handlers contain business logic (permission checks, data transformation) that should be in a service layer. For example, `projectRoutes.ts` directly calls `db.getProjectById()` for authorization checks.
No database transactions	Operations like `createProject` (create project, add managers, add developers) and `acceptInvitation` (create user, add to project, delete invitation) are not wrapped in transactions. Partial failures could leave inconsistent data.

The application provides a documented REST API with an OpenAPI specification and Swagger UI, which is a strong foundation. However, the interface design is inconsistent across different resource types. Projects follow proper RESTful conventions while appointments use verb-based URLs, response formats vary between endpoints, and field naming mixes camelCase and snake_case. Internally, TypeScript's type system is largely bypassed through pervasive use of `any`. These inconsistencies arose because features were developed incrementally in separate prompting sessions, and the AI did not enforce a uniform API style convention across all endpoints. Without an explicit instruction to harmonize conventions, each feature was implemented in isolation, resulting in a fragmented interface design.

Table 9: Interface issues

Issue	**Details**
Inconsistent URL patterns	Appointments use verbs in URLs (`/appointment/create`, `/appointment/update`, `/appointment/delete`) instead of RESTful resource patterns (`POST /appointments`, `PUT /appointments/:id`). Projects correctly use RESTful patterns.
Mixed ID conventions	Projects use both `uuid` and auto-increment `id` interchangeably across the API. Route params use `projectId` which sometimes refers to the integer ID, sometimes the UUID.
Inconsistent field naming	Backend uses camelCase (`startTime`, `createdAt`), but API responses mix snake_case (`start_time`, `created_at`, `project_id`) and camelCase.

The codebase exhibits significant code duplication and structural issues. Identical utility functions like the validation error handler appear in six separate files, permission checks are copy-pasted across every route handler, and the central data access file has grown into a 600-line monolith. The root cause is the incremental, prompt-by-prompt nature of the development process. When each feature is requested independently, the AI generates self-contained implementations rather than refactoring shared logic into reusable modules. Claude consistently produced working code for each prompt but did not look back at previously generated code to extract common patterns, since it has no persistent awareness of technical debt accumulating across sessions.

Table 10: Code Smells

Smell	Details
God File	`prismaClient.ts` (600 lines) contains all data access logic for 6 entities, JWT token generation, password hashing, database initialization, and connection management. Should be split into separate repository files.
Excessive use of `any`	At least 50 instances of `any` across all controllers and `prismaClient.ts`. Defeats the purpose of using TypeScript.
Duplicated error handlers	An identical validation error handler function is duplicated in 6 route files: `authRoutes.ts`, `projectRoutes.ts`, `userRoutes.ts`, `appointmentRoutes.ts`, `commentRoutes.ts`, and `promptRoutes.ts`.

Security is the weakest area of the generated application. Critical issues include a hardcoded JWT secret fallback, auto-created admin credentials with a trivial password and no rate limiting. Notably, the AI did include `helmet` and `bcrypt`, which are security measures that are part of standard Express.js boilerplate. But it did not go beyond these defaults. This reveals a fundamental limitation of AI-assisted development. The AI implements security measures it has seen frequently in training data (password hashing, helmet middleware) but does not proactively perform threat modeling or introduce defenses that were not explicitly requested. Security was never a dedicated prompt topic, and as a result it was never systematically addressed.

Table 11: Security Issues

Issue	Details
Hardcoded JWT secret	In `prismaClient.ts` and `auth.ts`. If the environment variable is not set, a trivially guessable secret is used, allowing forging of authentication tokens.
Default admin credentials	`prismaClient.ts` auto-creates an admin account `admin@example.com` with password `password` on every startup if it does not exist. No forced password change mechanism is in place.
No rate limiting	No rate limiting is configured in `index.ts` on the login endpoint or any other endpoint. The application is vulnerable to brute-force attacks.

Data protection compliance presents a mixed picture. The application includes some positive measures, like user anonymization, account deletion with cascade, bcrypt password hashing, and invitation-only registration. However, fundamental GDPR requirements are missing [6]. There is no privacy policy, no consent mechanism, no data export for portability and no audit logging. The AI implemented the data protection features it was explicitly asked for but did not infer the broader legal obligations that come with processing personal and educational data. Compliance requirements that were not formulated as concrete implementation tasks were simply not addressed.

Table 12: Data Protection Issues

Issue	**Details**
No privacy policy or consent mechanism	No privacy policy page exists. No consent checkbox is shown during registration. Data processing requires explicit consent [6].
Student data	Matriculation numbers, degree programs, and examination regulations are stored. In some jurisdictions, this is educational data requiring additional protections. No specific access controls limit who can view this data.
No cookie consent	While the app uses localStorage rather than cookies, `helmet` and session-related middleware may set cookies. If any cookies are used, a consent banner would be needed under the ePrivacy Directive [5].

7 Conclusion

This project set out to explore the practical implications of AI-assisted development within the context of building a project management web application. By integrating IntelliJ IDEA, GitHub Copilot, and Claude models into a real development workflow, the project provided concrete insights into how low-code and AI-supported tools shape modern software engineering practice.

The findings demonstrate that AI assistance can significantly accelerate routine and well-structured tasks. Code scaffolding, CRUD operations, standard routing logic, and repetitive refactoring benefited strongly from AI support, leading to measurable gains in development speed and reduced manual effort. In these contexts, AI acted as an effective productivity multiplier, particularly when paired with tools such as TypeScript and explicit API contracts that constrained and validated generated output.

The evaluation of the resulting application reinforced these observations on both the functional and structural level. The use case analysis showed that the vast majority of requirements were fully implemented, with partial implementations limited to features that depended on external service integration or third-party API availability, areas where AI guidance proved unreliable. The code quality review revealed a characteristic pattern of AI-assisted development: while the generated system is functional and built on sensible library choices, it exhibits significant weaknesses in type safety, code duplication, security hardening, and data protection compliance. These shortcomings arose not from individual prompt failures but from the incremental, session-based nature of AI interaction, which favours local correctness over global consistency and architectural discipline.

At the same time, the project clearly exposed the limitations of AI-assisted development. Tasks involving business rules, security-sensitive logic, domain-specific constraints, and cross-module coordination consistently required careful human intervention. AI-generated code often appeared locally correct while violating global invariants or architectural boundaries. These issues were especially pronounced when prompts lacked precision or when changes spanned backend logic, API contracts, and frontend state simultaneously. As a result, human review, testing, and architectural reasoning remained indispensable throughout the development process.

From a usability perspective, AI tools proved most effective when tasks were narrowly scoped and easily verifiable. Their usefulness decreased as contextual complexity increased, highlighting that usability in AI-assisted development is tightly coupled to task structure rather than interface design alone. Similarly, the expressiveness of AI interaction depended heavily on prompt quality. Effective prompts functioned much like precise specifications or test cases, requiring explicit articulation of constraints, assumptions, and edge cases.

Overall, the project reinforces a central conclusion: AI-assisted tools do not replace software engineers, but they fundamentally reshape how engineering work is performed. When embedded in a disciplined workflow with explicit contracts, strong typing, and systematic review practices, AI can enhance productivity without undermining correctness. When used indiscriminately or without sufficient oversight, however, it amplifies existing risks related to technical debt, architectural erosion, and latent defects. This duality underscores the importance of treating AI not as an autonomous developer, but as a powerful yet fallible collaborator whose outputs must be guided, constrained, and evaluated by human expertise [17].

References

[1] Anthropic. Claude haiku 4.5 system card. https://www.anthropic.com/claude-haiku-4-5-system-card, 2025. Accessed: 2026.

[2] Anthropic. Claude opus 4.5 system card. https://www.anthropic.com/claude-opus-4-5-system-card, 2025. Accessed: 2026.

[3] Anthropic. Claude sonnet 4.5 system card. https://www.anthropic.com/claude-sonnet-4-5-system-card, 2025. Accessed: 2026.

[4] Anthropic. Claude. https://www.anthropic.com/claude, 2026. Accessed: 2026.

[5] European Parliament and Council of the European Union. Directive 2002/58/ec (eprivacy directive). Technical Report L 201, Official Journal of the European Union, 2002.

[6] European Parliament and Council of the European Union. Regulation (eu) 2016/679 (general data protection regulation). Technical Report L 119, Official Journal of the European Union, 2016.

[7] Angela Fan et al. Large language models for software engineering: Survey and open problems. *IEEE/ACM International Conference on Software Engineering (ICSE FoSE)*, 2023.

[8] Y. Fu, P. Liang, Z. Luo, Y. Tian, and H. Zhang. Security weaknesses of copilot-generated code in github projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 2025.

[9] GitHub. GitHub Copilot. https://github.com/features/copilot, 2026. Accessed: 2026.

[10] GitHub. GitHub Copilot models. https://docs.github.com/en/copilot/reference/ai-models/supported-models, 2026. Accessed: 2026.

[11] GitHub. GitHub Copilot plans. https://github.com/features/copilot/plans, 2026. Accessed: 2026.

[12] JetBrains. IntelliJ IDEA. https://www.jetbrains.com/idea/, 2026. Accessed: 2026.

[13] JetBrains. IntelliJ IDEA features. https://www.jetbrains.com/idea/features/, 2026. Accessed: 2026.

[14] JetBrains. JetBrains AI Assistant. https://www.jetbrains.com/ai/, 2026. Accessed: 2026.

[15] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Emi Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[16] J. T. Liang, C. Yang, and B. A. Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 616–628, Lisbon, Portugal, 2024.

[17] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023.

[18] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023.

[19] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*, 2024.

[20] Douglas C. Schmidt, J. Spencer-Smith, Q. Fu, and J. Coplien. Transforming software development with generative ai: Empirical insights on collaboration and workflow. Technical report, Vanderbilt University, 2024.

[21] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, 2022.