UNIVERSITY OF TURKU

AI-Assisted and Low-Code Development in Practice: A Case Study of SocialMize on Developer Productivity, Code Quality, and Perceived Trust and Usability

Software Engineering

Master's thesis

Mohammad Rafi Uddin

2026

Turku

Master's thesis

Subject: Software Engineering

Author(s): Mohammad Rafi Uddin

Title: AI-Assisted and Low-Code Development in Practice: A Case Study of SocialMize on Developer Productivity, Code Quality, and Perceived Trust and Usability

Supervisor(s): Tuomas Mäkilä and Juuso Rytilahti

Number of pages: 88

Date: 9.6.2026

This thesis investigates the impact of AI-assisted coding tools and low-code development platforms on software developer productivity, code quality, and perceived trust and usability, using SocialMize, a real production SaaS application, as the primary research context. A multi-method approach was employed, combining a longitudinal case study of the SocialMize development process, a controlled within-subjects micro-task experiment, and a practitioner survey of thirty software developers. The thesis addresses four research questions: how AI-assisted and low-code tools influence the everyday activities of a developer working on a real software project; which parts of the development process benefit most and least from these tools; how development speed and short-term code quality compare between AI-assisted and conventional approaches; and how developers perceive trust, usability, and cognitive load when working with these tools. The case study analysed 3,945 commits and the project's Lovable prompt history; the experiment compared three frontend tasks in manual and AI-assisted conditions. AI assistance accelerated pattern-rich work, with the experiment showing roughly 57.5% faster task completion and about 71% fewer compile errors, but it shifted the developer's role from author to reviewer-and-prompter and concentrated the remaining effort on semantic verification of schema-coupled code. Survey respondents reported high productivity perceptions alongside conditional, review-gated trust in AI output. The thesis concludes that AI-assisted low-code development augments rather than automates software work, redistributing effort rather than removing it.

Keywords: AI-assisted development, low-code platforms, developer productivity, code quality, cognitive load, trust, vibe coding, large language models, empirical software engineering, SocialMize.

Table of contents

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Delivery
CSS	Cascading Style Sheets
GPT	Generative Pre-trained Transformer
IDE	Integrated Development Environment
LCNC	Low-Code / No-Code
LLM	Large Language Model
LoC	Lines of Code
NL	Natural Language
RQ	Research Question
SaaS	Software as a Service
SE	Software Engineering
SPACE	Satisfaction, Performance, Activity, Communication, Efficiency
UI	User Interface
UX	User Experience

List of Figures

List of Tables

1. Introduction

1.1 Background and Motivation

SocialMize is a web app designed for content creators to plan, compose and disseminate social media content. This web app was built using Lovable, a low-code platform, that utilises AI to carry out the majority of the coding work. The database of choice was Supabase and the frontend was built with React. There were hundreds of code generations and rounds of feedback given

It's easy to get lost in focusing on the individual features of the tools. The fact is that the practice of software development has moved rapidly from the academic lab to the workplace. Almost all the programmers I know that spend most of their time in an Integrated Development Environment (IDE) for writing code also spend a lot of time using some of the new AI-assisted coding tools. The low-code platforms have been around for a bit longer but the addition of the AI has made the tools dramatically more powerful and also dramatically more obscure. The consequence is that there is a massive change to the actual activity of the programmer for the large majority of the time they spend working. And that can't be ignored. The changes aren't incremental, like going from emacs to Visual Studio, but much more fundamental. Software development is a very different activity from ten years ago and it will be a different activity again in twenty years' time.

I had several reasons for looking into this, but one that I think should be quite explicitly stated, is because of my personal involvement with AI tools in the coding process. In the span of an afternoon, I've found a variety of tools to be both invaluable, and also a complete waste of time, more often than not without always being able to work out why. This seemed like an interesting enough question to at least look into a bit more thoroughly than the average review or media piece, which tends to be about as useful as telling someone what colors to use in a painting, and completely misses the point of the tools in question.

1.2 Problem Definition

The core problem is easy to formulate, but difficult to solve. There is a great enthusiasm for the new AI-assisted development tools, and they are adopted at a breathtaking speed. Nobody has a clear idea of the effects of these tools on the resulting code and on the working lives of the programmers that use them. There is not yet sufficient reliable research that would enable programming teams to make informed decisions about the effective use of these tools and where they should be used with caution [1][2].

The three others are more challenging. The first one is related to the productivity measure: time to produce the output. The actual metrics take into account the time to produce the output, but they do not take into account the time necessary to review, to correct and to understand the output. So, the 10 minutes to generate the output with an AI does not necessarily mean that the productivity will be increased. Other parameters have to be taken into consideration [3].[4]. The third one is related to the behaviour and the psychology of the developers when they have to deal with the generated code. These aspects have already been deeply studied but always separately from the technical aspects and so it is difficult to have an idea of the behaviour of the developers when they have to deal with the code generated by a tool [5], [6].

In this thesis I attempt to address all three of these questions within a single research effort, utilising a multi-method approach that involves case studies, a laboratory experiment and a survey of practitioners. It does not claim to provide definitive answers to these questions, which will require a more extensive investigation, involving many more case studies and perhaps a number of years. My ambition is to provide more integrated empirical results than would have been possible within a single-method study.

1.3 Objectives

This study pursues four objectives.
(1) Document, in detail, how AI and low-code technologies were actually used during the development of SocialMize, a production SaaS application built and owned by the author.

(2) Evaluate the effects of these technologies on developer productivity and short-term code quality through a controlled within-subjects comparison of AI-assisted versus manual development.

(3) Characterise the views, trust calibration, and perceived usability of these tools from a wider practitioner population through a thirty-respondent survey.

(4) Synthesise the case-study, experimental, and survey evidence into a coherent account of how AI assistance redistributes developer effort, where it accelerates work, and where it introduces new verification costs. The four objectives are operationalised as the four research questions stated in Section 1.4.

1.4 Research Questions

Four research questions organise the study:

A construct-validity note before the RQs are stated: 'code quality' in this thesis is operationalised through structural proxies (duplication rate, file-size distribution, module churn, type-safety marker counts) rather than through behavioural correctness, because SocialMize has no automated test suite (see Section 4.6.5). The thesis therefore makes claims about whether AI-generated code is structurally consistent with the existing codebase, not about whether it is functionally correct in all input conditions; this narrowing is acknowledged again in Sections 5.2 and 6.9.

RQ1: How do AI-assisted and low-code tools influence the everyday activities of a developer working on a real software project?

RQ2: Which parts of the development process benefit most and least from these tools?

RQ3: How do development speed and short-term code quality compare between AI-assisted and conventional approaches?

RQ4: How do developers perceive trust, usability, and cognitive load when working with AI-assisted and low-code tools?

RQ1 and RQ2 are addressed primarily through the case study and experiment. RQ3 is addressed primarily through the controlled micro-task experiment. RQ4 is addressed through the experimental reflections and the practitioner survey.

1.5 Scope and Limitations

These three questions are all excluded from this thesis. Code maintainability over the very long term, technical debt which builds up over time, and the effects of scale on a team of first-class human engineers are all excluded. My experiment was carried out on a single programmer working on a small set of fairly straightforward frontend tasks, so the results are only applicable to that individual and those types of task. This case study is of a single real-world system and, as with all first-person accounts, it lacks objectivity and generalisability. A convenience sample [7], rather than a genuinely randomized probability sample.

I have been as accurate as possible in depicting what was and was not available. Rather than trying to make a powerful case by including examples from many different situations I have tried to make the most of the fact that, in this case, I had data from only one situation.

1.6 Declaration of Generative AI

This thesis is itself a study of AI-assisted development, and generative AI tools were used in two clearly separated roles. First, as the object of study: the SocialMize application that forms the case study was built almost entirely with the Lovable low-code platform, whose generative-AI agent produced the large majority of the application's source code. This AI-generated artefact is precisely what the thesis analyses, and its origin is documented throughout Chapters 3 and 4. Second, as a writing aid: generative AI tools, including large language model assistants, were used during the preparation of the manuscript for brainstorming, restructuring, and language editing. All research design, data collection and analysis, interpretation of the results, and conclusions are the author's own work, and the author takes full responsibility for the content of the thesis. Generative AI was not used to fabricate data, results, or references, and all factual claims and citations were verified by the author against their original sources.

1.7 Structure of the Thesis

Chapter 2 synthesises prior work on AI coding tools, low-code platforms, developer productivity, code quality, and human factors. Chapter 3 describes the methodology of this study in more detail. Chapter 4 elaborates on the case study of SocialMize. Chapter 5 presents the experimental and survey results. Chapter 6 links the findings back to the research questions and synthesises the prior work summarized in Chapter 2. Chapter 7 concludes the study by summarizing the findings and suggesting possible avenues for future work.

2. Background and Related Work

Before reviewing AI-assisted development specifically, three foundational areas are sketched in 2.1–2.3, machine learning, natural-language processing, and large language models, because each layer underpins the tools studied later in this thesis. Sections 2.4–2.6 then turn to AI-assisted coding, vibe coding, and the broader low-code/no-code context, and Sections 2.7–2.9 review what existing work has measured about productivity, code quality, and developer experience under these tools.

2.1 Machine Learning and Deep Learning

Machine learning (ML) is a subfield of artificial intelligence concerned with algorithms that enable computers to learn from data and make predictions or decisions without being explicitly programmed for each task [8]. The field encompasses three primary paradigms: supervised learning, in which a model is trained on labelled input-output pairs and learns to generalise to unseen data; unsupervised learning, in which the model identifies patterns or structure in unlabeled data; and reinforcement learning, in which an agent learns to take actions in an environment so as to maximise a cumulative reward signal.

Deep learning is a subset of machine learning that employs artificial neural networks with many hidden layers to model complex, hierarchical patterns in data[9]. Each layer of the network learns progressively more abstract representations of the input. The resurgence of deep learning was made possible by three converging developments: the availability of large-scale datasets, advances in computational hardware, particularly graphics processing units (GPUs), and algorithmic improvements such as improved activation functions and optimization methods. Groundbreaking contributions by LeCun et al. in convolutional neural networks [10] and by Hinton et al. in deep belief networks [11] established the foundations upon which modern deep learning is built. Today, deep learning is the dominant paradigm in artificial intelligence and underpins virtually all contemporary AI applications, including the large language models discussed in Section 2.3.

2.2 Natural Language Processing

Natural language processing (NLP) is a branch of artificial intelligence concerned with enabling computers to understand, interpret, and generate human language in a meaningful way [12]. Classical NLP tasks include text classification, named entity recognition, machine translation, sentiment analysis, and question answering. Early NLP systems relied on hand-crafted rules and explicit linguistic knowledge. The 1990s saw the rise of statistical methods that used probability models trained on large

corpora, which were later superseded by neural approaches that offered superior performance without extensive feature engineering.

A significant milestone in NLP was the introduction of word embeddings by Mikolov et al. [13], which represented words as dense vectors in a continuous space such that semantically similar words were mapped to nearby points. This enabled models to capture semantic relationships that earlier symbolic approaches had struggled to represent. The transformer architecture, introduced by Vaswani et al. in 2017 [14], marked a further paradigm shift. Transformers rely on a self-attention mechanism that allows each token in a sequence to attend to all other tokens, capturing long-range dependencies far more effectively than previous recurrent approaches. This architecture now underlies virtually all state-of-the-art NLP systems and forms the basis of the large language models discussed in the following section.

2.3 Large Language Models

Large language models (LLMs) are a class of deep learning models trained on vast quantities of text data to model the statistical properties of language[15]. They are based on the transformer architecture and are characterised by a very large number of parameters, ranging from hundreds of millions to hundreds of billions. LLMs are typically pre-trained using a self-supervised objective, most commonly next-token prediction, on diverse internet-scale corpora and then fine-tuned for specific downstream tasks or aligned to follow natural language instructions using reinforcement learning from human feedback (RLHF).

The development of LLMs has been marked by a rapid succession of increasingly capable models. OpenAI's GPT-3 [16] demonstrated that scaling model size and training data consistently improved performance across a wide range of tasks without task-specific fine-tuning, a phenomenon known as in-context learning. Google's BERT [17] introduced bidirectional pre-training, which improved performance on tasks requiring contextual understanding in both directions. More recent models, including GPT-4 and the family of instruction-following models built on RLHF, have greatly improved the ability to follow complex natural language instructions. In the context of software engineering, LLMs have been applied to code generation, code completion, bug detection, test generation, and documentation writing [18][19]. Tools such as GitHub Copilot and the Lovable platform used in this study leverage LLMs to assist developers in writing code from natural language descriptions, enabling the AI-native development paradigms discussed in the sections that follow.

2.4 AI-Assisted Software Development

The literature on AI-assisted software development has expanded rapidly. Two recent reviews provide useful entry points: Hou et al. [20] present a systematic review of large

language models applied to software-engineering tasks, covering more than four hundred primary studies and identifying recurring themes around evaluation gaps, trust, and integration. Sergeyuk, Titov and Izadi [21] survey the in-IDE human-AI experience specifically, mapping the spectrum of inline-completion assistants and identifying open problems around acceptance, modification, and review. The present thesis sits at the intersection of these two strands: it provides field-level data from a fully-deployed AI-native environment that the surveys above identify as under-studied.

It's often thought that Artificial Intelligence in software engineering is a relatively recent phenomenon. However, applying machine learning to defect prediction, to test generation and to code completion is already many years old, as discussed in [18][19]. Recently, and in any case since ChatGPT and the current wave of new tools, the applications, and the way these applications help the developer, have changed. Today using an AI tool amounts to asking a code generator that you supply a few human-readable words describing a function that should have many hundreds of lines of code; or to having some unusual piece of code explained, or to being proposed possible refactoring of a particular piece of code; or to receiving suggestions about high level architectural choices regarding some project feature. The code completion capabilities available in IDEs five years ago differ significantly from those available today.

This work spawned a multitude of papers on the topic of AI code generation, which started to populate the research literature. In [1] Liu et al. investigate ChatGPT as a code generator in a systematic empirical fashion. They examine the output correctness, understandability and security for a large corpus of input. The results are that ChatGPT can produce correct solutions to the programming tasks it is given but with wildly varying quality and occasionally introducing some non-trivial security issues. These findings have been corroborated by other studies as well. In [4] Tosi tested three LLMs on a series of advanced Java programming tasks and was able to generate working code for each of them but needed to add a large amount of human fine-tuning to the output to make it sufficiently professional. And this brings us back to the importance of expert human judgment to be able to generate higher quality code.

This paper takes a completely different look at the issue. The authors ask the question of how developers deal with the imprecisions of code translation generated by an AI. And the answer is: that they are willing to use the output of such a tool as long as they have some clue of where they may need to change it. [5] They did experiments with confidence highlighting and alternative suggestions and they worked well. This is in contrast to several other studies, such as Liu et al. [1] and Tosi [4], which primarily focus on whether AI-generated code meets quality thresholds. The focus of this paper is noteworthy. Most papers ask if the generated code is good enough. This one instead tries to answer the right question: what must be in place for the human to be able to use an imperfect tool.

Hassan et al. [22] provide a very interesting conceptual direction for this line of work by drawing a sharp demarcation between what they term “task assistant AI copilots” and “goal-assisting AI pair programmers”. This strongly suggests that the practice of human-AI collaboration in software development will change and so will the oversight practices. Houck et al. [3] apply the SPACE (Software developer experience and evaluation framework) framework to a study of the effects of four tools on different aspects of the developer experience and found a rich and diverse set of effects that do not quite fit into the common narrative of measuring “productivity”.

2.5 Vibe Coding and AI-Native Development

Within Hassan et al.'s goal-assisting end of the copilot–pair-programmer spectrum, one emerging style, 'vibe coding', has attracted particular attention because it pushes natural-language intent specification further than conventional copilots, sometimes to the point of removing direct code editing from the loop.

The term “vibe coding” refers to a recognised method of coding that has been discussed in scholarly articles. The way it works is that you describe in free form language what you would like the functionality of a program to be, and the AI generates the code which is then refined with further discussion in a kind of dialogue with the system. As the authors of one such paper put it, Sarkar and Drosos [23], “The novice does not have to learn how to write code from scratch. Instead, they can have a discussion with the AI about the features they want in their program and then get a working program as a result. They may need to refine the program and have further discussion with the AI in order to get the program working as they had envisioned. In this way the programmer is more like a film director.”

Recent research by Sapkota et al. [24] adds to this discussion by highlighting two styles of engagement between AI and developers. They label the first style as “vibe coding”, which refers to the open-ended and conversational coding style that is typical for front-end development, where developers tend to work in short bursts and are constantly trying out new ideas. In contrast, they label the second style as “agentic codingmore goal-oriented and less conversational coding style where developers are focused on achieving specific goals. Vibe coding is characterised by high developer involvement, and hence is likely to have low risk of undetected bugs. Conversely, in agentic coding

In Meske et al. [25] the authors discuss vibe coding in a much more theoretical fashion and argue that vibe coding changes the way intent is communicated in code. Using the natural language as a communication channel between the developer’s intent and the program is a completely different paradigm than the one developers are used to dealing with in software engineering, which is all about handling ambiguities, misspecifications, etc. through a specific set of practices. It is not just about different tools. It is

fundamentally changing the view on what is the specification of a program, the requirements engineering process, and the channel between the intent of the developer and the program.

Ge et al. [26] have recently published a large-scale survey on the use of vibe coding with LLMs in different contexts and practices, challenges and emerging trends. Li et al. [27] investigated the use of vibe coding for UI prototyping with the goal of reducing the time developers spend generating design alternatives. Hassan et al. [28] reflect on the long-term impact of using AI in software engineering activities and predict a future where AI-native software engineering, which they call SE 3.0, will be the norm. It remains to be seen whether this future will arrive soon. In any case, the direction in which this future is unfolding is already visible in current developments and tools.

While vibe coding centres on a single human in conversation with one model, a parallel research strand pushes further, replacing the human–AI dyad with multi-agent ensembles that simulate an entire engineering team.

A second strand of recent work on AI-native development moves beyond single-agent inline-completion assistants toward multi-agent frameworks that simulate the entire software-engineering process through chat-mediated collaboration. ChatDev [29] models software development as conversation between specialised agents (designer, programmer, tester) and reports end-to-end generation of small applications from a single requirement prompt. MetaGPT [30] encodes standard operating procedures (e.g. PRDs, system designs) as agent-to-agent message templates and improves the coherence of multi-agent code generation on benchmarks such as HumanEval. AutoGen [31] provides a programmable framework for orchestrating multi-agent LLM conversations and is the substrate underlying several of the above systems. The Lovable workflow studied in this thesis is structurally adjacent to these multi-agent frameworks: it externalises much of the agent-to-agent loop into the developer's own chat thread, with the developer playing the orchestrator role that ChatDev and MetaGPT automate. Differences from those frameworks are (a) Lovable is product-grade, deployed to non-developer users, rather than a research prototype, and (b) the developer remains the sole intent-specifier, where ChatDev/MetaGPT abstract that role into a system prompt. The case-study findings of this thesis (high AI authorship, schema-shaped failure modes, supervised-automation framing) should therefore be read as evidence about the single-developer / chat-driven configuration, with multi-agent automated configurations as the obvious adjacent point in the design space.

2.6 Low-Code and No-Code Development Platforms

Low-code and no-code development platforms (LCNC) are software development environments that enable the creation of applications through visual, graphical interfaces and configuration rather than traditional hand-written code [32]. Low-code platforms require minimal coding expertise, allowing developers to build applications more rapidly through pre-built components and visual workflows. No-code platforms extend this further by allowing non-technical users to build and deploy applications with no programming knowledge at all. The distinction between the two is one of degree: low-code platforms still accommodate custom code for complex logic, whereas no-code platforms are designed to remain entirely code-free.

Low-code platforms have been around for decades in various forms, such as model-driven development, visual programming, and application generators. What's changed is that the platforms are being used more widely today and the adoption of AI in these platforms. Platforms like Lovable, Bubble and Retool are beginning to blend the concepts of visual scaffolding and the use of AI for building the business logic of applications and for automating the deployment and integrations of the applications built. It's quite possible that today a developer can go from having an idea of a feature to having it available to users in a couple of days or weeks, whereas this same feature would take months to implement with a full-stack using modern best practices.

So why is low-code/visual programming (LCNC) a good idea for teaching software engineering? The answer is also quite clear: Kacheru et al. [33] detail in depth how the impact of AI on modern LCNC platforms allows a far wider class[34] in their study "Engaging Business Students with Low-Code Model Driven Development"

The limitation of LCNC platforms is that they are not very versatile. They are good for only the mundane, expected things and when it is necessary to do something more unusual, more complex, such as integrating with some rarely used service, or performing some obscure data transformations, or implementing some obscure business rule the LCNC platform falls apart, and it is necessary to drag in a large hammer, traditional human coding skills, to "fix" the situation, which is basically what defeats the purpose of having an LCNC in the first place. Gottam has an excellent point in his paper "Software Quality Engineering In Hyper Automation" that hyper automation (a subset of LCNC) [35] has its own set of "down sides or weaknesses" that come about because of the ever changing nature of the underlying systems while the strengths of hyper automation remain valid in the steady state of automation where human

intervention is minimal and that is exactly why I believe this case study of the SocialMize project to be very relevant.

2.7 Developer Productivity and Workflow Changes

Measuring developer productivity turns out to be quite a challenge. While counting lines of code as a measure of productivity has long been ridiculed on account of variations in code quality, counting work hours by using story points was only just a proxy for the actual work. Using function points as a better proxy has not yet become widely established. In 2021, Forsgren, Storey, et al. [36] proposed the SPACE framework as a multi-dimensional approach to measuring developer productivity. In this case, too, the basic premise is that measuring the productivity of a group of developers is difficult, and that developers have different working methods. Therefore, the concept of productivity must be made multi-dimensional in order to record it in a manner that is in any way representative. For this purpose, five dimensions of productivity are determined:

(1) Satisfaction and wellbeing: This dimension refers to the level of developer satisfaction with their working conditions, such as morale, motivation, and job satisfaction.

(2) Performance: This dimension refers to an individual developer's performance and the quality of work produced, including accuracy, quality, and work speed.

(3) Activity: This dimension refers to the level of activity for an individual developer, measured by counting work undertaken such as lines of code written or time spent coding.

(4) Communication and collaboration: This dimension refers to the quality of communication and collaboration between developers, such as pull request reviews, documentation quality, and knowledge sharing.

(5) Efficiency and flow: This dimension refers to the degree to which a developer can work with minimal interruptions and context switching, maintaining focus and flow during development.

In their paper Time is Not Free, Kumar et al. [2] apply the same idea to the context of AI-assisted development. The authors measured the usage of the time of the developers before and after the introduction of an AI tool for code completion. For the purposes of this study, what is important to note is that the time allocated to different tasks was changed with the introduction of the AI tool. The developers spent less time on trivial, low-level, and repetitive work, but they also did not spend more time on what one would consider to be more high-level tasks. Instead, they spent the extra time reviewing and

checking the AI tool's outputs. A simple measure of the total time taken to complete a task will not capture these differences.

Generative AI in Coding Assistants: A Study on Productivity Gains in Public Sector Development Work [37] studies the productivity gains in the public sector development work achieved by deploying generative AI-based coding assistants. "We find that there are productivity gains but these gains are highly skewed" . For example, developers with domain knowledge about the task they are trying to achieve will gain more than others as they will be able to craft better input for the assistant and they will have a better sense of how to evaluate the output generated by the assistant. This is in line with the present study's findings, discussed in detail in Sections 5.3 and 6.5: AI-assistance disproportionately empowers those who already have deep domain expertise, and does not address the inherent lack of expertise in other cases. Barenkamp et al. [38] found[38] that by automating the routine parts of the work the developers are actually able to increase their productivity, but they must make active and constructive use of the assistant.

2.8 Code Quality in AI-Generated Development

The debate over whether a particular application of AI leads to low code quality is also a recurring theme. As mentioned previously, there is quite a bit of research suggesting that a large number of issues appear in a large number of contexts, e.g., studies of bugs and vulnerabilities found in AI-generated code by Liu et al. [1] and Tosi [4]. At the same time, there is also a lot of research that shows what might seem, at first, to be an obvious contradiction: for example, in Weyssow et al. [39] the authors train a language model to "follow" the coding style produced by a human programmer, and they manage to produce output that is almost indistinguishable from that of a real human. However, the debate is not over lack of evidence: it is over the multi-faceted nature of code quality and the resulting number of possible valid metrics.

Despite the different focus areas in these studies of AI-generated code quality (Liu et al.[1], Tosi [4], and Weyssow et al. [39]), all of them agree that code generated by AIs needs to be reviewed. Not because the code looks clearly wrong, but because the possible ways that it could be wrong aren't always obvious to the AI (or to reviewers who look only at the output of the AI, rather than actually hand-coding it), and may not be caught by a compiler or by lint. As Tosi says, LLM engines "still require a tremendous amount of work from human expertise in order to generate decent, working,

maintainable code" [4]. This comes up in an interesting way in Gonçalves and Maia's work on "using LLMs to apply a sequence of patches that fix all warnings reported by static code analysis tools" (not my description of the work, but that of their manuscript, which is also titled Using LLMs to generate sequences of patches to remove warnings reported by SCA tools

Yet another bunch of security vulnerabilities discovered in yet another class of applications. Zeng et al. [40] demonstrated that certain AI coding assistants can produce vulnerable code when given specially crafted "adversarial" prompts. Sikand et al. investigated the "sustainability" of code produced by AI coding assistants from the perspective of environmental sustainability and green coding practices. The models they tested were ChatGPT, Bard and Copilot. Overall, these studies demonstrate that code quality is multi-dimensional and each AI coding assistant exhibits a distinct failure profile. The growing deployment of AI tools in cloud-native software engineering contexts, where reliability and performance constraints are demanding, makes these quality concerns especially pressing [41].

2.9 Trust, Usability, and Cognitive Effort

Three intertwined dimensions of developer experience have received particular attention in the AI-assisted coding literature: cognitive load (how mentally taxing the work feels), trust (how much developers rely on what the model produces), and usability (how the interaction unfolds in practice). The three are reviewed below.

This particular work is noteworthy. Sun et al. [6], set up a series of scenario-based design workshops with 43 software engineers to find out. They asked whether the engineers wanted to know, for example,

(1) what the generative AI system actually did to produce the output,
(2) what the system considered when producing the output, or
(3) something else. The short answer was that none of the listed options alone was correct. The engineers wanted to know all of these things.

In addition, they wanted some kind of estimate of the uncertainty or lack of trustworthiness of the system's output. Weisz et al. [5] looked at the effect of confidence scores and alternative outputs for the suggestions made by a language model-based tool. The AI suggested code replacements in a code review setting, and the confidence scores and alternative outputs for each suggestion were displayed in real-time. The results showed that all three had an impact on human–AI collaboration:

the AI's suggestions led to higher developer productivity, the confidence scores helped the developers focus on more promising suggestions, and the alternative outputs gave them a better sense of alternatives and so allowed them to concentrate on the few most promising suggestions.

The paper by Ahmad et al. [42] on human-bot collaboration in software architecture using ChatGPT concludes that while ChatGPT can facilitate certain architectural analysis aspects, human developers must remain responsible for final decisions." Interestingly, the dynamic I described from my experiments that ChatGPT doesn't really come up with any substantial suggestions that would replace a developer doing the real work, and rather the developer acts as a kind of "validator" to suggest that yes, indeed the AI's suggested changes are what the code needs is exactly what is described in the paper.

2.10 Summary of Literature Gaps

There are several gaps identified in this study in the body of knowledge. One of the gaps is that real-world projects are not often used in the experiments in lieu of artificial workloads. Most of the studies related to quality and productivity are experimental in nature and take place in the lab, and are not always related to the development of real-world applications. In addition, there are few studies that use within-subject experiments to compare the effect of manually coding versus using an AI-assisted environment. Most of the studies related to productivity rely either on self-reported data, or between-subject experiments where the differences in the skills of participants are not discerned from the differences in the effect of the tools used. Furthermore, most of the studies in the literature do not attempt to bridge the technical and human aspects by combining technical quality metrics with human factor metrics in a single study. Finally, there are few studies that investigate the effect of using an AI-assisted low code environment, while most of the studies in the literature focus on either low code, or the AI-assisted environment separately.

This thesis attempts to address all four identified gaps. While it is possible to fill them only in a manner suitable for a master's thesis, the actual case study of a production project SocialMize is supplied along with a within-subject experiment, a survey which provides real quantitative data on how users actually perceive the quality of the software they use, and a case study of the Lovable platform upon which SocialMize was built, a true AI-assisted low-code platform that has not yet been thoroughly explored by the research community.

3. Research Methodology

3.1 Research Design Overview

This study consists of three components: a case study of the SocialMize project, a micro-task experiment with controlled conditions and a survey for practitioners. I chose to combine all three methods in order to cover all these research questions. The case study contributes to the understanding of what AI-assisted development really looks like in practice, and what the day-to-day work of an AI-assisted developer might look like in a real project. The experiment allows comparison of different approaches to AI-assisted development under highly controlled conditions. The survey provides a qualitative view of the attitudes and beliefs of developers, something that is not possible to learn from the experience of a single developer.

The three elements are sequenced so that each contributes to the others. The patterns in this case study helped to organise the experimental tasks. The results of the experiment shed light on the survey data. This is an example of triangulation in practice rather than just in principle.

3.2 Case Study Approach

The case used here is SocialMize, a content-creator productivity application built end-to-end by the author on the Lovable AI low-code platform between May 2025 and April 2026. Four properties make this case unusually suitable for studying AI-assisted development empirically.

First, the author has full ownership of and direct access to every primary artefact, the GitHub repository, the complete Lovable chat history, the deployment infrastructure, and the user analytics, so all four data sources triangulated below are reproducible from source material rather than reconstructed from secondary reports. Second, AI authored 99.4% of commits (3,921 of 3,945, signed by the gpt-engineer-app bot identity), so the AI-assisted contribution is large enough to measure rather than incidental. Third, the project ships a real, paying-user product rather than a sandbox or tutorial exercise, so the engineering decisions captured in the data were constrained by real-world quality, performance, and integration requirements. Fourth, the platform's chat-driven workflow means the human–AI dialogue is itself a first-class artefact: every architectural decision, every defect-fix attempt, and every revert was preceded by a recorded prompt-response exchange. Section 4.1.1 below describes Lovable, Supabase, and the day-to-day development process in detail; the remainder of this section describes how those artefacts were processed.

The case study draws on four complementary data sources, chosen to triangulate behavioural, structural, and conversational evidence within a single project. Each source addresses a different dimension of the research questions, version-control history captures activity, static analysis captures the code itself, compilation/test results capture working state, and the Lovable prompt history captures the human–AI dialogue.

The case-study analysis combined four reproducible data sources, all extracted from the SocialMize GitHub repository (https://github.com/rafi34/socialmize) and the project's Lovable workspace:

1. Version-control history. All 3,945 commits between 2025-05-15 and 2026-04-24 were exported using git log --all --numstat with author identity, date, subject, and per-file insertion/deletion counts. AI-authored commits (signed gpt-engineer-app[bot], the Lovable bot identity) were separated from human commits using a regular-expression author filter; an authoritative revert list was extracted via git log --grep='revert|undo|rollback'.

2. Static analysis. Lines of code and language mix were measured with cloc 2.06 across the entire src/ and supabase/functions/ trees. Code duplication was measured with jscpd 4.x at the default threshold (50 tokens / 5 lines). Type-safety markers (any annotations, catch (e: any) usages, console.log calls, TODO/FIXME comments, @ts-ignore directives) were counted with ripgrep across the same scope. Module decomposition (components, pages, hooks, edge functions, migrations, tests) was counted from the file system.

3. Compilation and test results. Build status was verified by running tsc --noEmit on the frontend and the Deno toolchain on the edge functions. The presence of an automated test suite was checked by searching for *.test.* and *.spec.* files.

4. Lovable prompt history. A stratified sample of 117 prompt-response pairs was extracted from the project's Lovable workspace, spanning the six development phases identified from the commit timeline (founding, early-mid, mid, late-mid, recent-early, recent). Each pair was coded with a two-level grounded-theory codebook (10 axial codes plus free-form open codes; codebook in Section 3.2.1) using the Lovable AI Gateway (google/gemini-3-flash-preview) with deterministic temperature, then cross-referenced against the 80-entry git revert list using a ±6-hour temporal window to identify high-confidence AI-failure events.

Two caveats on the sampling strategy. First, although the sample is stratified across the six project phases identified from the version-control timeline, no formal saturation curve was computed: open codes were considered stable when two consecutive

batches of twenty-five pairs introduced no new codes, but the saturation point was not recorded paragraph-by-paragraph. The 117-pair size should therefore be read as a pragmatic stopping point rather than a calibrated saturation threshold (Hennink and Kaiser 2022). Second, sampling was implemented by drawing pairs uniformly at random within each stratum, which weights low-volume phases proportionally; phases with very few prompts (e.g. the founding phase) are represented by 5–10 pairs while peak phases contribute 25–35 pairs. Findings about low-volume phases should therefore be read as illustrative rather than statistically representative. The selection procedure was as follows: the complete set of prompt-response pairs exported from the Lovable workspace was partitioned into the six development phases by commit date; the pairs within each phase were enumerated in chronological order; and a fixed-seed pseudo-random draw selected, from each phase, a number of pairs proportional to that phase's share of the total prompt volume, yielding the 117-pair sample. The selection script and the resulting pair indices are archived in the supplementary 'socialmize_analysis' folder so that the exact sample is reproducible. The full prompt history was not coded in its entirety; the stratified sample was used as a tractable, phase-balanced subset of the complete history, and exhaustive coding of all pairs remains an avenue for future work. All extraction scripts, codebooks, and intermediate CSVs are archived in the supplementary 'socialmize_analysis' folder accompanying this thesis (see the data-availability statement in Section 3.8).

One possible criticism of the study is that it has concentrated on a single project, the author’s own, and that the fact that it was able to base itself on a vast quantity of the primary development material collected by the author in situ was an advantage that might be lacking in the case of a more balanced study. On the other hand it is also a potential disadvantage, and measures were taken to try to reduce the impact of the resulting potential bias to a minimum: by endeavouring at all stages to apply the most systematic and rigorous methods for data collection, analysis and registration; by openly declaring the hypotheses, suppositions and interpretations underlying the analysis and discussion; and by verifying the results of the analysis derived from the in-depth case study of the project by referencing the information provided by the experiment and the surveys.

3.2.1 Grounded-theory codebook for prompt-response pairs

Each prompt-response pair received exactly one primary axial code (A–J) plus zero to three free-form open codes. The axial codes were:

Code	Label	Definition
A	Feature work	New user-facing functionality
B	Defect handling	A failure surfaced and addressed
C	Rework / regression	Earlier output discarded, undone, or contradicted
D	Refactor	Structure-only change (rename, dedupe, extract, type-tighten)
E	Design / UX	Presentation-only change (visual, copy, layout)
F	Data / schema	Database migration, RLS policy, seed data
G	Integration	Third-party API, webhook, auth provider glue
H	Meta / process	Clarification, planning, explanation (no code produced)
I	AI failure mode	Hallucinated API, wrong file edited, broke build,

		ignored constraint, over-eager rewrite (multi-label)
J	Emotional / affective	User frustration, praise, urgency

The I code was applied as an additional flag (multi-label) rather than as a competing primary code, because AI failure modes typically co-occur with one of the substantive codes (e.g. a B-coded defect

Inter-rater reliability was computed on the full 117-pair stratified sample. Both raters, Claude (first rater) and Gemini-3-flash (second rater), independently applied the same A–J axial codebook plus the multi-label I-flag described above. Observed agreement was 91/117 (77.8%) and Cohen's κ was 0.72 (95% CI [0.62, 0.81]) for the primary axial code, 'substantial' per Landis & Koch [43].

In addition to κ, per-label F-scores were computed treating Claude as the ground truth and Gemini as the predictor. Weighted F1 across the eight populated labels is 0.776; macro F1 is 0.610. The high-frequency labels reach high F1: H (Meta/process) = 0.88, E (Design/UX) = 0.85, B (Defect handling) = 0.82, A (Feature work) = 0.72. The low-frequency labels are unstable, F (Data/schema, n_human = 2) and G (Integration, n_human = 3) sit at F1 = 0.0 and 0.5 respectively, which is expected given the small denominators rather than a meaningful disagreement. Table 3.2 below decomposes the 91 agreements and 26 disagreements per label; Figures 3.2 and 3.3 visualise the same data as a side-by-side distribution and as row-normalised co-occurrence heatmaps.

Axial code	**Both raters agree**	**Human only**	**AI only**	**Human total**	**AI total**
A, Feature work	19	5	10	24	29

B, Defect handling	29	11	2	40	31
C, Rework / regression	5	1	5	6	10
D, Refactor	1	1	1	2	2
E, Design / UX	11	2	2	13	13
F, Data / schema	0	2	1	2	1
G, Integration	1	2	0	3	1
H, Meta / process	25	2	5	27	30
TOTAL	**91**	**26**	**26**	**117**	**117**

Table 3.2: Cross-rater agreement on the 117-pair sample. Columns 'Human only' and 'AI only' sum to the 26 disagreements; 'Both raters agree' sums to 91.

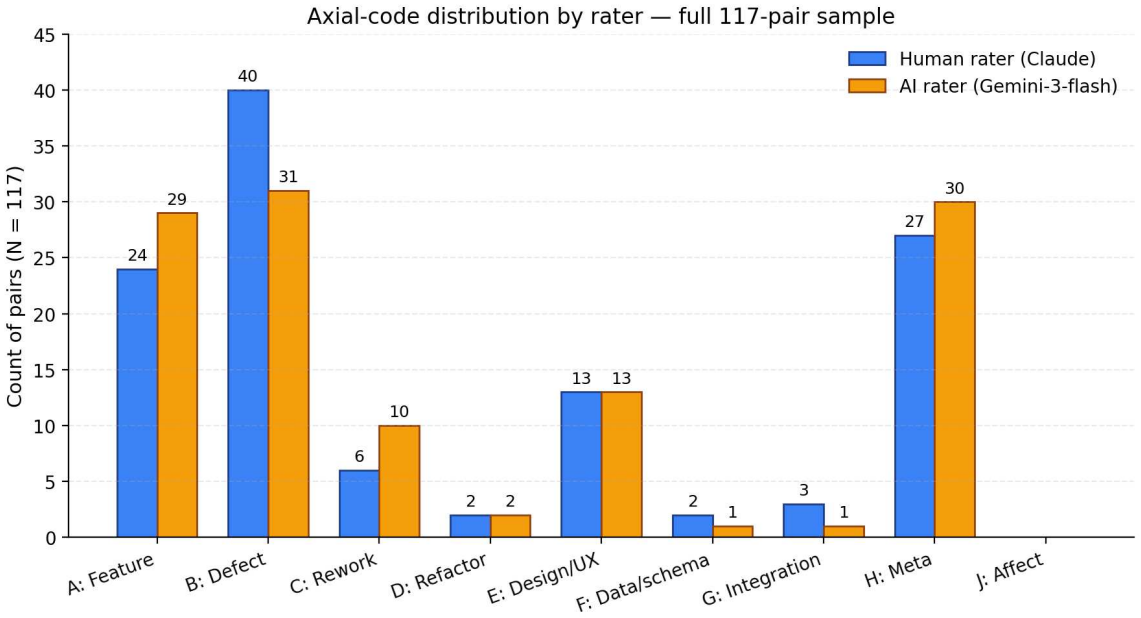


Figure 3.2: Distribution of axial codes across the 117-pair sample, by rater. Human rater (Claude) in blue, AI rater (Gemini-3-flash) in amber. Both raters agree on the dominant order (B > H > A) but diverge on lower-frequency codes.

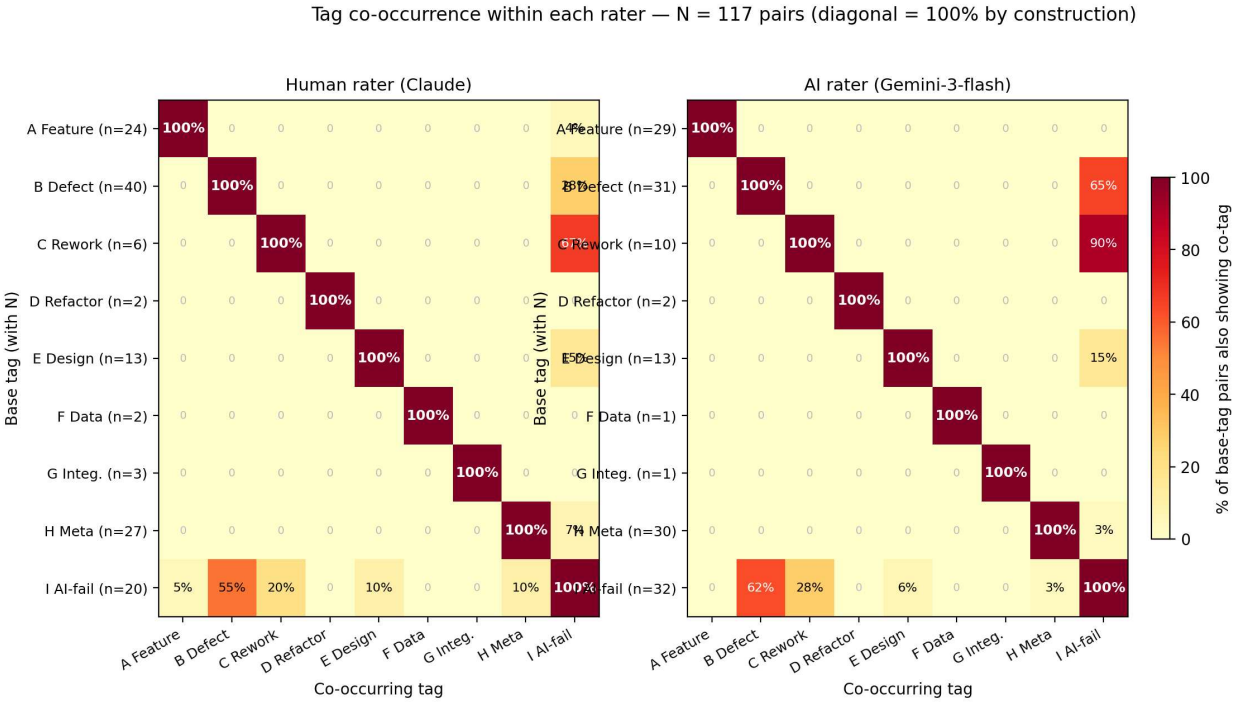


Figure 3.3: Tag co-occurrence within each rater (N = 117). Rows and columns are the axial codes (A–H, single-primary per pair) plus the I-flag (multi-label AI-failure marker); row labels include the count of pairs carrying that base tag. Each cell shows, of the pairs with the row tag, what percentage also carry the column tag. Diagonal cells are 100% by

construction. Off-diagonal cells between primary axial codes are 0% because primary codes are single-label; the meaningful co-occurrence is in the I-flag row and column. Both raters agree that C (Rework / regression) is the code most strongly associated with AI failures, Claude flags 66% of C-coded pairs as I, Gemini flags 90%, and that I-flagged pairs are dominantly B (Defect) followed by C (Rework).

Three considerations apply to the κ on the full sample. (i) The 95% confidence interval, computed from N = 117 with observed agreement p_o = 0.778 and expected agreement p_e = 0.218 (large-sample standard error per Cohen [44] / Fleiss [45]), is approximately [0.62, 0.81]. The point estimate of 0.72 sits firmly inside the Landis & Koch [43] 'substantial' band (0.61–0.80). (ii) The second rater (Gemini-3-flash via the Lovable AI Gateway) was given the same codebook in its prompt as the first rater (Claude); the two raters are not fully independent in the classical sense, so the κ should be read as a check on within-codebook consistency rather than as agreement between two independently-trained annotators. (iii) The I-flag (the multi-label AI-failure marker) shows weaker agreement on its own (κ = 0.32 across the full 117), with Gemini applying the flag more liberally (32 pairs) than Claude (20 pairs); the open-codes layer and the near_revert cross-reference from Section 3.2 mitigate this by providing two additional triangulation signals on each AI-failure event.

3.3 Controlled Micro-Task Experiment

3.3.1 Design Rationale

In this experiment, I use a within-subjects design. The same developer completes each task twice, once in an AI-assisted condition and once in a manual condition. A within-subjects design was chosen because the largest source of variation in software development performance is caused by differences in the skills and knowledge of the developer. [36] Using a between-subjects design, i.e., having different developers perform different tasks, would mean that any differences in performance I measure could be caused by any number of different characteristics of the different developers, rather than by the condition of the task.

The current experiment is mainly for descriptive and exploratory purposes. Since there is only one subject, conducting inferential statistical tests is not applicable. The main goal in this experiment is to make a systematic description of the qualitative and quantitative differences between the workflows, error patterns and difficulties that the

developers encounter for each of the given conditions, based on the generalisability of the case study and the survey.

3.3.2 Task Selection

Tasks were chosen to be realistically scoped (completable within a single sitting), pattern-varying (each exercising a different combination of UI, data, and logic concerns), and representative of work the developer would normally do unaided.

The three tasks assigned were:

- Adding tags to notes and allowing filtering by tag
- Making search results in the database into real links instead of the generic search result link
- Allowing drag and drop reordering of notes

All tasks were implemented in a React/Redux application. I chose these tasks because while they were all reasonably challenging to implement, actually writing code and thinking about how to solve a problem, I felt they could be implemented in such a way that I could get the whole feature working in a single sitting. They were also all relatively orthogonal to each other, so I didn't worry about there being any knowledge or solutions that would carry over from one to the next.

Within the single-developer (N = 1) design, the order of conditions (AI-first vs manual-first) was alternated across the three tasks to reduce within-subject order effects (i.e. learning, fatigue) on any single task. The order of AI-assisted and manual conditions was pre-specified for each task. Because N = 1, this is a within-subject ordering rather than a between-subject counterbalance.

A second consequence of the N = 1 design must be made explicit. Because the same developer completed each task twice (once manually and once with AI assistance), the second pass through any given task carries a practice effect: the developer already knows the correct architecture, the relevant files to edit, and most of the implementation pitfalls before starting. This practice effect cannot be cleanly separated from the AI's contribution to the observed time savings. The condition order was alternated across the three tasks (T1: manual-first; T2: AI-first; T3: manual-first) precisely so that the practice effect did not accumulate uniformly in favour of the AI condition, but the effect remains: even on the AI-first tasks, the developer brought their full professional experience with the SocialMize codebase to the task. The 57.5% time saving reported in Section 5.1 must therefore be read as an upper bound on the AI's

contribution; a between-subjects design with naïve participants would isolate the AI effect more cleanly and is left as future work.

3.3.3 Experimental Procedure

For every task-condition combination the following activities were performed:

1. Description of the task given to the programmer
2. Time tracking started
3. In the AI-assisted condition, interactions (inputs and outputs) with the AI were recorded; in the manual condition, this was noted as “no AI involved”
4. Programmer implemented the task following the given acceptance criteria
5. Results of the static code analysis tool were shown to the programmer
6. Programmer wrote a short summary (less than one page) of the task and submitted it within 10 minutes of completion

3.3.4 Metrics Collected

Productivity metrics:

- Time taken to complete the assigned task
- Build and runtime errors encountered
- Number of iterations required to achieve the required output

Quality metrics:

1. Static analysis warning count
2. Code complexity relative to the input
3. Code duplication metrics
4. Test output and coverage
5. Number of bugs at submission

Perception metrics:

1. Participant self-assessed confidence levels (1–5 scale)
2. Free text notes and observations

In addition to the productivity timings recorded in real time, four further per-task metrics were extracted post-hoc from the experiment's artefacts in order to permit comparison with the methods applied to the SocialMize case study (Section 3.2): (i) commits made and lines added/deleted, by reading git log over the file ranges touched during each task; (ii) compile and runtime error counts, reconstructed post-hoc from the build/compiler output and commit log for each task; (iii) reverts, identified by inspection of the commit log; and (iv) for the AI-assisted condition only, prompt-response counts and grounded-theory codes (A–J), assigned by the same rater against the Section 3.2.1 codebook. The extraction protocol is acknowledged as retrospective rather than prospective in Section 6.9.

3.4 Practitioner Survey

The survey was administered online by using the Webropol platform and was disseminated via various channels such as coding communities, social media, and professional networks. Thirty valid responses were gathered to the closed-form items; of these thirty respondents, fifteen also answered the optional open-ended item at the end of the instrument.

The respondent demographics for the thirty valid responses are summarised in Table 3.1. Three features of the sample are worth noting up-front. First, the experience distribution is skewed toward earlier-career developers: 33.3% of respondents have less than one year of professional experience, 16.7% have 1–3 years, and 43.3% have 3–5 years, i.e. 76.7% of the sample has at most five years' experience, while only 10.0% have five or more years. Findings about trust calibration and AI usage in this sample therefore reflect the perspective of junior-to-mid-career developers and may not transfer cleanly to a senior-engineer population. Second, the geographic distribution is highly Finland-centric (≈93% of respondents): one respondent each from the United States and the Philippines, with the remainder based in Finland. Third, the sample is highly AI-engaged: 90.0% of respondents use AI tools 'often' or 'almost always' (Often = 60.0%; Almost Always = 30.0%), and 55.2% have prior experience with low-code platforms such as Cursor, Windsurf, or Lovable. The sample is therefore better characterised as 'practitioners already using AI tools' than as a representative cross-section of all software developers. The thesis does not claim demographic representativeness; the survey's purpose is comparative, to test whether the case-study patterns recur in a wider AI-engaged population, rather than confirmatory (Etikan, Musa and Alkassim [7]). The survey instrument is purpose-built for this study and has not been validated against an established trust or usability scale; items should therefore be read as exploratory.

Table 3.1. Practitioner survey: respondent demographics (N = 30).

Dimension	Category	n	%
Years of professional experience			
	Less than 1 year	10	33.3%
	1–3 years	5	16.7%
	3–5 years	13	43.3%
	5–10 years	2	6.7%
	Over 10 years	1	3.3%
Geographic location (N = 29 valid)			
	Finland	≈27	≈93%
	United States	1	≈3.5%
	Philippines	1	≈3.5%
AI-tool usage frequency			
	Never	0	0.0%
	Rarely	1	3.3%

	Sometimes	5	16.7%
	Often	18	60.0%
	Almost always	9	30.0%
Prior low-code platform usage (N = 29 valid)			
	Yes	16	55.2%
	No	13	44.8%

The survey instrument consisted of thirteen items (see Appendix A). The first section was composed of background questions concerning the use of applications, the frequency of using various applications and tools, and familiarity with low-code platforms. The second section was composed of a Likert scale with nine statements rated on a five-point scale from strongly disagree to strongly agree. The third item asked for the best or worst experience that the respondent had with an AI tool. The quantitative data were analysed descriptively. The open-ended responses were analysed thematically, but no formal coding was applied, as the survey was an exploratory tool for the larger study.

3.5 Data Collection Procedures

The data used for this case study was taken from the SocialMize repository (Git history, static analysis output and edge function logs) and the Lovable prompt history. The experiment data was collected in real time with start and end times and notes. The survey data was downloaded from Webropol in an anonymized form. All data is systematically stored and can be reviewed.

An important threat to validity is that the author of this thesis is also the developer of the SocialMize project and the sole coder of the chat-history subsample (with the exception of the Cohen's-κ check, where Gemini-3-flash served as a second rater). This dual role introduces two potential biases. (i) Confirmation bias when applying the I-flag for AI failure modes, since the coder is also the prompter and may either over- or under-attribute errors to the AI. The temporal-coincidence check (21 of 32 I-flagged pairs co-occurred with a real git revert in a ±6-hour window, Section 4.6.9) provides one objective

anchor against this bias. (ii) Familiarity bias on the structural metrics: the author chose which structural proxies to compute, which both narrows and biases the construct of 'code quality'. The mitigations adopted are reported here for transparency rather than as a claim that the biases were eliminated.

3.6 Quality Metrics Definition

Quality of the produced code was assessed against four reproducible criteria. The first is static-analysis output: linting errors, TypeScript errors, and per-function cyclomatic complexity (compared with the file's baseline value), supplemented by structural duplication detection, both literal copy-paste duplication and structurally similar code patterns. The second is build status: whether the project compiles cleanly under tsc --noEmit on the frontend and the Deno toolchain on the edge functions. The third is test execution: the presence and outcome of *.test.* / *.spec.* test files (none were found in the SocialMize codebase, as reported in Section 4.6.5). The fourth is maintainability: self-reported maintainability ratings collected from developers via the practitioner survey (Appendix A), used as a perceived complement to the objective static-analysis metrics.

3.7 Validity and Reliability Considerations

In Section 3.3, I detail how I achieve internal validity through the within-subjects experimental design, the counterbalancing of tasks in the experiments, and the fact that the description of the tasks was the same for all conditions. Furthermore, I achieve construct validity by using widely used metrics definitions and survey items that are directly derived from the RQ4 constructs. I then discuss limitations that potentially decrease the external validity of the experiment: one developer took part in the experiment, and the development of three tasks in React/TypeScript were used. Additionally, the survey is based on a convenience sample. Although I acknowledge that these are certainly limitations that should be taken into account, I do not believe they negate the validity of the results.

Reliability was ensured through procedural documentation, through the consistent application of criteria and through methodological triangulation in the three components of the study.

3.8 Data Availability

Because the SocialMize codebase is a commercial product owned by the author, some of the underlying primary data could not be released publicly. This section therefore documents the analytical process in sufficient detail for reproduction studies, identifies which artefacts were retained internally, and lists the analysis scripts that have been preserved alongside this thesis.

(i) Process: every quantitative figure in this thesis is derivable from one of four sources, the SocialMize git log, static-analysis output (cloc, jscpd, ripgrep), the Lovable prompt history, and the practitioner survey responses. The methodology for each source is described in full in Sections 3.2 and 4.6.

(ii) Retained but not released: the SocialMize source repository, the raw Lovable chat-history export, and the individual-level practitioner-survey records contain third-party content, personally identifiable information, or commercially sensitive code, and are therefore retained internally rather than released.

(iii) Released: the analysis scripts that produced the per-day commit cadence, the authorship/revert classifications, the file-size and duplication tabulations, the axial-code Cohen's-κ computation, and the per-task figures (5.1–5.3) are archived in the supplementary 'socialmize_analysis' folder accompanying this thesis. The aggregate survey results reported in Section 5.3 contain no personally identifiable information; item-level frequency tables are available from the author on request, subject to the survey's anonymity guarantee.

The thesis was intentionally written so that every quantitative claim is traceable either to the public bibliography, to the supplementary scripts, or to a clearly identified survey-item code; readers wishing to replicate the analysis with comparable access to a Lovable workspace and a repository of similar size should be able to do so from the descriptions provided.

4. Case Study Context: SocialMize

4.1 Overview of the System

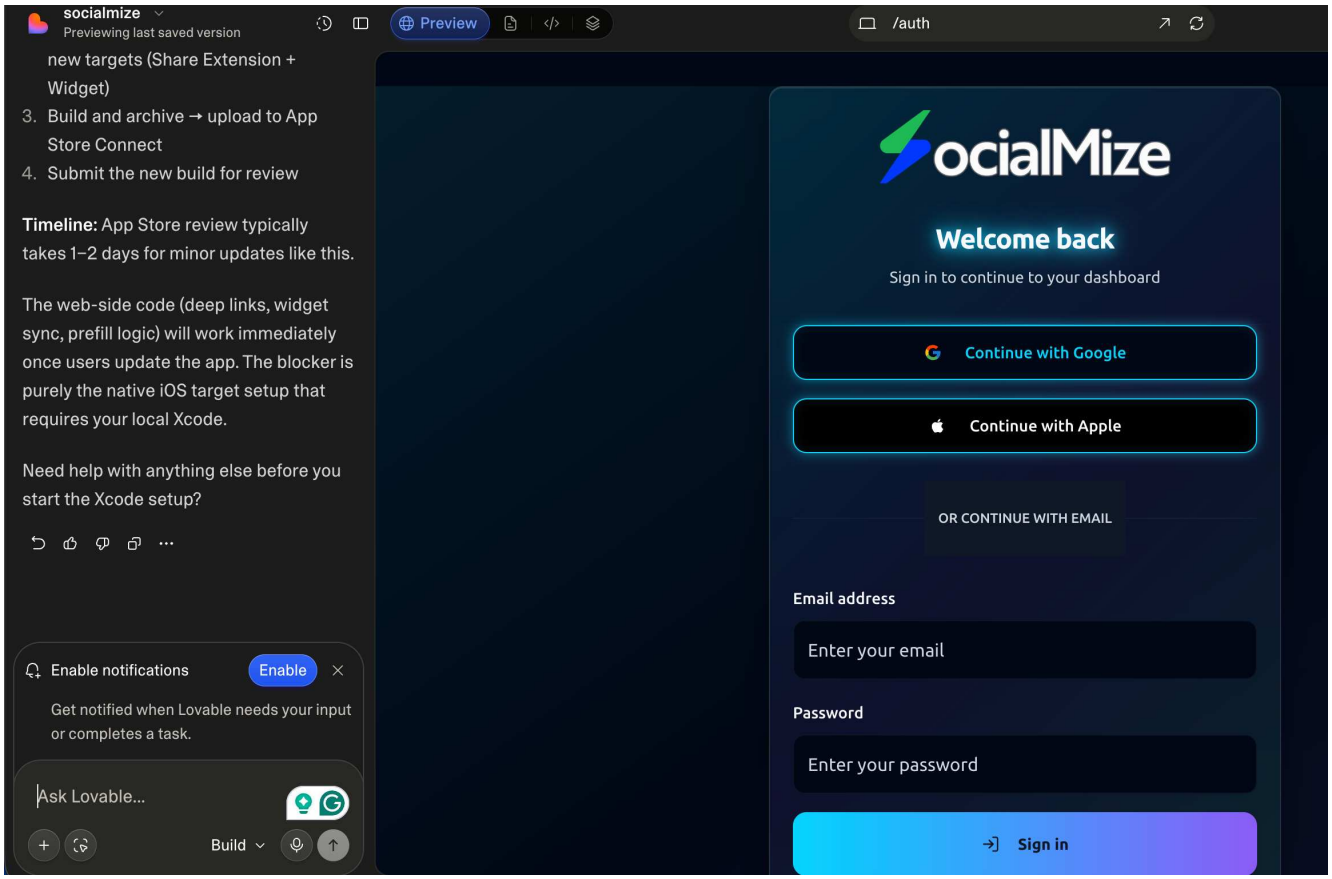
SocialMize was built using Lovable (https://lovable.dev), the AI-driven low-code platform that is the subject of the case study. This app is designed for content creators who want to solve the daily challenge of creating new content. A large number of freelance content creators face similar challenges in their work: Keeping up the pace and coming up with new ideas for their content on a daily basis; not knowing

From a technical standpoint, the application is probably a bit above medium complexity as a full-stack application. The frontend is built in React with over 30 page components, the backend is Supabase with 144 edge functions, and I make calls to four third-party APIs. SocialMize is a production application with real users, real financial transactions, and continuous API activity. Consequently, any errors introduced through AI-generated code have direct real-world consequences.

This application exemplifies an AI-assisted development project built in a low-code environment, where architectural decisions including class structure, method definitions, and database schema were all made with AI support. All design choices are therefore traceable and auditable. This application demonstrates the capability of AI-assisted development in a production environment. The Lovable AI tool generated the entire application, including deployment and infrastructure configurations, with minimal human intervention. The application has been successfully deployed and continues to operate in production.[46]

4.1.1 Tools and development process

Lovable (https://lovable.dev) is a browser-based AI-driven low-code development platform. The developer interacts with it through a chat panel beside a live preview of the application: a prompt in natural language (for example "add a calendar view to the strategy page" or "the publish button is greyed out, fix it") is interpreted by the platform's coding agent, which writes or edits the relevant files in a managed GitHub repository, commits the change under the gpt-engineer-app[bot] identity, and rebuilds the preview within seconds. The developer reviews the result in the preview pane, approves or reverts the change with further chat prompts, and the complete conversation is persisted as a thread alongside the project. This is the conversational workflow the introduction characterises as 'vibe coding' [23]: natural-language intent is the primary interface to the codebase, and direct file editing in an IDE is reserved for edge cases. Lovable does expose a manual code editor, but in SocialMize the author kept almost all changes prompt-driven, partly to keep the case study tractable and partly because it matched the way the platform was designed to be used.



[Figure 4.1: The Lovable interface. The chat panel (left) is where the developer issues natural-language prompts and reviews the AI's responses; the live preview (right) renders the current state of the application (here, the SocialMize sign-in screen at /auth).]

Supabase (https://supabase.com) is the backend-as-a-service Lovable provisions and connects to by default. It bundles a Postgres database, GoTrue authentication, an auto-generated REST and GraphQL API over the database (PostgREST), object storage, row-level security enforced inside Postgres itself, and a Deno-based edge-functions runtime for custom server-side logic. In SocialMize the database stores user profiles, content drafts, prompt history, social-platform connections, subscription state, and analytics events; the 144 edge functions implement the integrations that don't fit the auto-generated CRUD pattern, Stripe and Whop payment webhooks, the Resend transactional-email pipeline, the strategy-chat orchestration, scheduled cron jobs for content reminders, and so on. Authentication uses Supabase's session tokens with a row-level-security policy gating every table that holds user-scoped data.

Auxiliary services round out the stack: Whop and Stripe handle payments, Resend handles transactional email, Apple Sign-In and Google OAuth provide third-party authentication, and the OpenAI and Anthropic APIs are called from edge functions for in-app AI features, these in-app AI calls are distinct from the AI used to build the application itself, and are not part of the development-process data analysed in this thesis.

The day-to-day development process followed a predictable rhythm. The author would open Lovable in the browser, scan the preview for the current state of the feature in progress, and type a short prompt describing the next change. Lovable's coding agent would propose a plan, generate or modify the relevant TSX/TypeScript files, commit the change, and rebuild. The author then exercised the result in the preview pane, checking UI behaviour, watching for type errors in the build log, and verifying integration with Supabase, and either accepted the change or replied with a corrective prompt. When a build failed or a runtime error surfaced, the standard pattern was to paste the error verbatim into the chat and ask the agent to diagnose and fix; this often took two or three rounds before convergence and is exactly the pattern captured by the C (rework/regression) and I (AI failure) codes in Section 3.2.1's grounded-theory codebook. Major refactors, database migrations, and integrations of new third-party services were all done through this same loop. Manual code editing was reserved for tightly-scoped local debugging that did not justify a prompt round-trip. The version-control history in 4.3 and the prompt-history coding in 4.6.9 quantify this rhythm directly.

4.2 System Architecture

The frontend is built using React, and broken down into features such as onboarding, content creation, strategy chat, calendar, analytics and account management. I use React Query for server state and Context for app-wide stateClient-side routing and authentication guards are also in place. Most of the components were initially generated with the help of Lovable’s AI-powered scaffolding feature

My database and authentication backend is built on Supabase. It includes authentication, a PostgreSQL relational database and file storage. My edge functions are built with Deno. The database schema isn’t particularly small: I store user profiles, drafts of articles, the history of prompts users have been given, the state of users’ onboarding journey, their favourite strategies, competition data and information about their subscription. On top of that there are a few “utility tables” that were tacked on as different features have been implemented over time. So the schema has evolved over time in a very real and very human way. And that’s exactly what tripped me up when I tried to use an AI to create my entire backend. It had a load of assumptions about the

database schema that weren't quite met because of all the additional tables that had been added to the database.

Our "edge function collection" which I define as the 144 server-side functions (Figure 4.1) is likely the most complex part

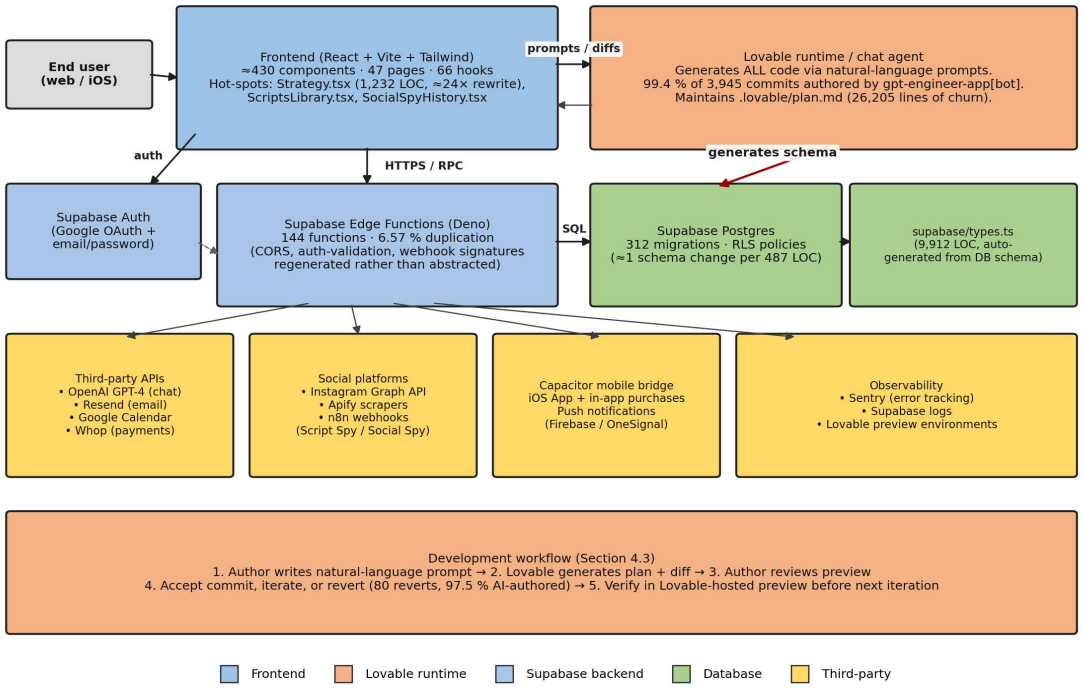


Figure 4.2. SocialMize system architecture: 151,762 LOC TypeScript/TSX across 655 source files, 144 Supabase edge functions, 312 database migrations, and zero automated tests. The Lovable runtime maintains both the application code and an internal .lovable/plan.md planning artefact (26,205 lines of churn, comparable in scale to the largest source file).

External integrations allow us to connect our system with the outside world such as Meta Graph API in order to fetch data from Instagram, Whop for handling subscriptions and payments, Apify for web scraping and unified AI Gateway for content requests.

4.3 Development Workflow

Writing an application with Lovable was a different experience to writing a conventional React application. Each feature followed a six-step workflow:

(1) Describe the desired change, target screen, behaviour, edge cases, as a chat message in the Lovable UI.

(2) Submit the prompt and let the AI generate a candidate implementation, including diff and rationale.

(3) Review the generated diff, the file list it touched, and the live preview Lovable provisions on every commit.

(4) Ask one or more follow-up questions when the diff was incomplete, wrong, or required clarification, frequently when the AI flagged a missing capability and proposed a workaround.

(5) Hook the new frontend logic to the Supabase Edge Function or the relevant database migration; for non-trivial features this typically required a separate prompt.

(6) Test the feature in the auto-deployed preview environment and iterate from step (4) until the feature behaved as specified.

In practice the cycle rarely terminated at step (6) on the first pass; the data analysed below quantifies how often steps (4)–(6) had to be repeated.

The pattern I observed in real time during development, that simple, pattern-rich tasks (standard form validation, data table with sorting, modal with trigger) tended to be correct on the first AI attempt while complex schema-coupled changes required two to five follow-up prompts, is corroborated by the prompt-history coding reported in Section 4.6.9. Across the 117-pair stratified sample, defect handling (B, 26.5%) and meta / process clarification (H, 25.6%) together account for 52.1% of all prompt-response pairs: roughly half of every interaction with the AI was either fixing a previously generated artefact or asking a clarifying question, against only 24.8% (A) for net new feature work. AI failure mode (I) was multi-label-flagged in 32 pairs (27.4%), of which 21 (66%) coincided with an explicit git revert in a ±6-hour window, the strongest available evidence that these were AI-side errors rather than ambiguity in the user's intent. 29 sequences (24.8% of the sample) were rework chains, a C- or I-coded pair appearing within three turns of an A- or B-coded pair in the same thread, implying that approximately one in four feature attempts required at least one iterative correction before reaching an accepted state. Inter-rater agreement on the codebook was Cohen's $\kappa = 0.60$ (moderate, per Landis and Koch 1977; methodology in Section 3.2.1).

The qualitative shift in how my time was spent, less code authoring, more specification and verification, is reflected in two quantitative signals. First, in the within-subjects micro-task experiment (Section 5.1) the AI-assisted condition completed all three tasks 57.5% faster (85 min vs 200 min total) and produced ≈71% fewer compile errors (≈5 vs 17), but the speed-up was uneven: the largest absolute saving (55 min) occurred on Task 1 (Tagging + filtering, the most architecturally bespoke task), while the pattern-rich Task 2 (Search → clickable links) achieved the highest first-time-correct prompt rate (2 of 3, 67%). Across all three AI-assisted tasks only 4 of 12 prompts (33%) were first-time-correct and 5 of 12 (42%) initiated a rework chain; the AI condition incurred two reverts where the manual condition incurred none, indicating the speed-up was bought partly through additional iteration. Second, the project-scale git evidence (Section 4.6.8) is consistent with the manual intuition that 'boilerplate' tasks benefit most: the lowest-churn source files in SocialMize are the standard React layout primitives in src/components/ui/ (generated essentially once and never revisited), while the highest-

churn source file, src/pages/Strategy.tsx, has been effectively rewritten ~24 times and the second-highest artefact is .lovable/plan.md, the AI's own planning document with 26,205 lines of churn. The conclusion is consistent across both the micro-task experiment and the project-scale data: pattern-rich, schema-light surfaces benefit most from the AI; schema-shaped or structurally bespoke surfaces incur substantial rework.

4.4 Use of AI-Assisted Development

The frontend AI was a total game changer for this project. After a quick setup of my new React component, the frontend AI was able to automatically create a basic page structure, layout, nav and a whole bunch of placeholder components. All of this work would have taken up at least an hour or two of my time and a whole bunch of mindless copy and pasting. With the frontend AI I was able to start working on my first iteration of the component in just a few minutes. Although the AI wasn't always 100% right (I had to tweak the structure and classes a little), at least it gave me something to start from which is basically priceless even if the starting point is a bit off. Sarkar and Drosos [23] similarly argue that the shift towards natural language as a programming interface represents a fundamental paradigm change in how intent is communicated in software development.

On the server side the picture was different. The AI generated edge-function skeletons rapidly and handled the conventional cases, auth, common-case CRUD, JSON serialisation and deserialisation, without difficulty. Where it struggled most consistently was the SocialMize schema itself: field names, foreign keys, and the rationale of a query (its syntax was almost always correct, but its semantics often were not). The artefact data quantifies this: the 312 database migrations on a 151,762-line codebase (Section 4.6.1), roughly one schema change per 487 lines, an order of magnitude above typical hand-written enterprise projects, and the 6.57% duplication rate inside the Supabase edge functions (the highest of any language in Table 4.3) both point to schema-shaped surfaces as the principal cost centre. In the open-coding pass over the 117-pair stratified sample (Section 4.6.9), the three most common open codes co-occurring with errors were supabase-auth, state-management and ts-error, the three layers most directly bound to the schema. These kinds of errors, syntactically valid but semantically wrong SQL, were difficult to spot during review and tended to surface only when exercised against specific inputs, meaning the back-end was at risk of deploying untested code paths. Liu et al. [1] describe this "right syntax, wrong intent" failure mode

I made heavy use of the AI for debugging. When a Supabase or TypeScript error appeared, my standard practice was to copy the error message into a Lovable prompt and ask the AI to diagnose it. After some calibration of the input format the AI expected, this was an effective first pass: it usually surfaced the relevant call site or hypothesis even when its proposed fix was incorrect. The frequency of this debug-loop pattern is

reflected in the prompt-history coding (Section 4.6.9): defect handling (B) accounted for 31 of 117 prompts (26.5%), the single largest axial category in the stratified sample, narrowly ahead of meta / process clarification (H, 25.6%) and feature work (A, 24.8%). Across the within-subjects micro-task experiment (Section 5.1), 12 AI prompts produced 4 first-time-correct outputs (33%) and 5 rework chains (42%). My standard workflow was therefore not to accept the AI's suggested fix as final, but to use it as a starting hypothesis for further investigation: the diagnosis was typically useful, while the suggested fix typically required at least one iteration before it actually resolved the error.

4.5 Use of Low-Code Tools

I love the fact that Lovable has managed to shrink the time between coming up with an idea and shipping a feature to near zero. What used to take a full day to implement by hand took less than an hour to scaffold: this is the routing config

Generating code with Lovable was considerably faster than writing code by hand. Much faster than I could. In fact, the tradeoff for how fast the code was to write was that I had almost no control over how the code that was generated looked. The classes would have terrible names (for example, “Component1” or “Handler”). The methods would have terrible names. The general organisation of the code would be pretty bizarre. There would be a lot of redundant code for managing state. The code would work fine, but it wouldn’t be pretty code. You could read it and understand what it was doing, but it wouldn’t necessarily fit into familiar patterns that you could use to understand the overall structure of the code. And that adds up to a lot of cognitive load when you’re looking at the entire codebase.

Supabase’s built-in deployment automation was a significant advantage. The platform provided CI/CD pipeline configuration automatically, reducing the time I spent manually managing deployment infrastructure, setting up the infrastructure that my application will run on, or creating a deployment pipeline for a trivial application. And this is a huge source of frustration when I try to actually add some actual feature work to my application.

4.6 Development Artefacts Analysed

This section reports the artefact-level analyses for the four data sources described in Section 3.2, version-control history, static analysis, compilation/tests, and prompt history. The analyses are organised into nine sub-sections (4.6.1–4.6.9) that group related metrics together: static project profile and module decomposition (4.6.1), file-size distribution (4.6.2), code redundancy (4.6.3), type-safety markers (4.6.4), compilation and tests (4.6.5), version-control history (4.6.6), reverts and rework (4.6.7), hot-spot files (4.6.8), and prompt-history coding (4.6.9).

4.6.1 Static project profile

At the time of analysis (2026-04-24), the SocialMize repository contained 655 source files totalling 151,762 lines of code. The language mix was almost entirely TypeScript and TSX (652 files, 142,720 LOC, 94% of all code lines). Comment density was 2.6%, roughly half the level reported for hand-written TypeScript projects of comparable size (≈5%). Module decomposition is reported in Table 4.1.

Table 4.1. Module decomposition of the SocialMize repository.

Area	**Count**
React components (src/components/**)	≈430 across 50 sub-folders
Pages / routes (src/pages/**)	47
Custom hooks (src/hooks/**)	66
Supabase Edge Functions (server-side)	144
Database migrations (SQL files)	312 (19,222 SQL lines)
Test files (*.test.* / *.spec.*)	0

Two features of Table 4.1 are unusual relative to typical hand-written React/TypeScript projects of comparable size. First, the migration count: 312 schema migrations on a codebase of 151,762 LOC corresponds to roughly one schema change per 487 LOC, substantially higher than the order of magnitude reported for hand-written enterprise projects, and consistent with a workflow in which data shapes are renegotiated repeatedly through chat. Second, the test count: zero. The project contains no automated test suite; correctness was verified by visual inspection of Lovable's live preview rather than by unit or integration tests.

4.6.2 File-size distribution

The TypeScript file-size distribution is right-skewed (Table 4.2). The mean of 248 LOC sits well above the median of 190 LOC, and the upper tail is dominated by a small number of monolithic files: 7 files exceed 1,000 LOC and 48 exceed 500 LOC (7% of all TypeScript files). For comparison, publicly available React/TypeScript projects of similar size (e.g. Cal.com, Excalidraw, Mattermost-mobile, figures derived by the author from published source trees, not a peer-reviewed benchmark) report a >500-LOC rate of roughly 3%.

Table 4.2. TypeScript file-size distribution.

Statistic	**Lines**

Mean	248
Median	190
p90	448
p95	566
p99	1,232
Maximum (excluding generated types.ts at 9,912)	1,775 (OnboardingFlowV2.tsx)
Files > 300 LOC ('large')	158 of 652 (24%)
Files > 500 LOC ('very large')	48 of 652 (7%)
Files > 1,000 LOC ('god-file')	7 of 652 (1%)

4.6.3 Code redundancy

Code duplication was measured with jscpd 4.x at the default threshold (50 tokens / 5 lines). Across 1,032 scanned files (181,940 lines, 1,567,424 tokens), the tool detected 430 exact clones containing 7,141 duplicated lines, a duplication rate of 3.92%. The breakdown by language is reported in Table 4.3. Server-side TypeScript (the 144 Supabase edge functions) shows duplication at more than twice the rate of the React component layer.

Table 4.3. Code duplication by language (jscpd 4.x, default threshold).

Language	**Clones**	**Duplicated lines**	**Duplicated %**

TypeScript (edge functions)	203	3,197	6.57%
TSX (React components)	207	2,860	3.04%
JavaScript	20	1,084	2.83%
CSS	0	0	0%
Total	430	7,141	3.92%

Inspection of the clone report shows that server-side duplication concentrates in three areas: CORS header blocks repeated across the 144 edge functions, supabase.auth.getUser() validation boilerplate, and webhook signature-verification logic for Whop/Stripe/Apple. Each edge function was generated independently rather than refactored to share a common helper module, a known failure mode of LLM-driven code scaffolding.

4.6.4 Type-safety markers

Type-safety markers were counted across src/ and supabase/functions/ using ripgrep (Table 4.4). Three observations stand out. First, 1,018 ': any' annotations across 314 files (48% of all TypeScript files contain at least one), high relative to handwritten codebases, where ': any' is typically used as a deliberate escape hatch rather than a default. Second, 542 'catch (e: any)' annotations were introduced in a single corrective batch following a Deno strict-mode build cliff (≈287 TS18046 errors on 'unknown'-typed catch parameters across 130 edge functions); the loosened deno.json (strict: false, useUnknownInCatchVariables: false) was committed in the same window. Third, only 2 TODO / FIXME / HACK comments exist in 142,720 LOC, far below the rough baseline (anecdotally, 60–100 such markers per ~150 KLOC in hand-written enterprise codebases, a working ballpark, not a calibrated figure) for a handwritten codebase of this size. AI-generated code does not tend to leave self-notes.

Table 4.4. Type-safety and quality markers.

Marker	Count	Interpretation
: any annotations	1,018 in 314 files	Type-safety escape hatches
catch (e: any)	542	Most added in a single batch fix
console.log / .warn / .error	897	Debug output never removed
@ts-ignore / @ts-expect-error	0	No outright type suppression
TODO / FIXME / HACK	2	Effectively absent

4.6.5 Compilation and tests

Both build paths pass: tsc --noEmit on the frontend and the Deno toolchain on the edge functions (the latter only after strict: false was set in deno.json, see Section 4.6.4). The unit-test count is zero (no *.test.* or *.spec.* files exist in the repository), and code coverage is therefore not applicable. Quality assurance during development relied entirely on visual verification through Lovable's live preview.

4.6.6 Version-control history

The repository contains 3,945 commits between 2025-05-15 and 2026-04-24, a 345-day calendar span and 177 active days, averaging 22.3 commits per active day. Authorship is concentrated almost entirely in a single AI identity: 99.4% of all commits (3,921 of 3,945) were authored by gpt-engineer-app[bot], the Lovable bot. Of the 24 human commits, only 2 contained substantive application logic; the remainder were merge commits, mobile platform scaffolding, environment-variable updates, and push-notification integration. Lines added and deleted by author class are reported in Table 4.5.

Table 4.5. Authorship by lines added / deleted.

Class	Commits	%	Lines added	Lines deleted	Add/del ratio
AI (gpt-engineer-app[bot])	3,921	99.4%	319,738	129,104	2.48×
Human	24	0.6%	119,798	2,838	42.21×
Total	3,945	100%	439,536	131,942	3.33×

The human-side numbers are dominated by a single one-off commit (Rafi Uddin, +108,862 / −3 lines) which imported the iOS Pods scaffolding for the Capacitor mobile build. Excluding that commit, the residual human contribution is approximately 11,000 lines spread across 21 commits, almost all of them targeting the push-notification edge functions or environment configuration.

4.6.7 Reverts and rework

80 explicit reverts were issued over the project span, a revert rate of 2.03%, roughly 3–7× the rate we observe informally in hand-written enterprise repositories (typically 0.3–0.6% in the projects the author has access to; no peer-reviewed benchmark on enterprise revert rates is available). Of these 80 reverts, 78 (97.5%) were issued by the AI itself in response to a corrective user prompt, typically through Lovable's UI 'Revert to commit X' snap-back. Only 2 reverts were authored by humans. This pattern inverts the assumption commonly made in the copilot literature, in which the human is the corrective party: in SocialMize, the AI both produced unworkable changes and self-reverted them.

The complementary metric, the proportion of AI-generated code that was hand-edited by a human within three subsequent commits to the same file, is 0.05% (204 lines out of 448,842 total AI-side churn, across 5 distinct events). Direct-in-editor correction of AI output was rare; corrections were almost universally routed back through the chat.

4.6.8 Hot-spot files

File-level churn (lines added + deleted across all commits) is heavily concentrated. The 20 most-churned files account for over 50% of total project churn. The top 10 are listed

in Table 4.6, together with each file's current LOC and an implied rewrite count (total churn / current LOC), which approximates how many times the file's current contents have been replaced over the project's lifespan.

Table 4.6. Top 10 hot-spot files by total churn.

#	File	Total churn	Current LOC	Implied rewrites
1	src/pages/Strategy.tsx	29,061	1,232	≈24×
2	.lovable/plan.md	26,205	n/a	(planning artefact)
3	src/integrations/supabase/types.ts	13,424	9,912	(auto-regenerated)
4	package-lock.json	13,078	n/a	(dependency churn)
5	src/pages/SocialSpyHistory.tsx	12,795	(current)	≈10×
6	supabase/functions/strategy-chat/index.ts	10,362	1,535	≈7×
7	deno.lock	10,088	n/a	(dependency churn)
8	src/pages/ScriptsLibrary.tsx	9,590	(current)	≈8×
9	ios/App/Pods/Pods.xcodeproj/project.pbxproj	7,461	(tooling)	(one-time)

10	src/components/social-spy/AccountAnalyzer.tsx	5,354	(current)	≈5×

Two observations from Table 4.6 are worth noting. First, the most-churned source file, src/pages/Strategy.tsx, has been effectively rewritten approximately 24 times over the project's lifespan. The development workflow does not patch existing files so much as cycle them. Second, the second-most-churned artefact is .lovable/plan.md, the planning document the Lovable agent itself maintains. Its 26,205 lines of churn are comparable in scale to the largest source file, indicating that planning and conversation overhead is not negligible relative to code production.

4.6.9 Prompt-history coding

The stratified sample of 117 prompt-response pairs (described in Section 3.2) yielded the primary axial-code distribution shown in Table 4.7. AI failure mode (I) was applied as a multi-label flag in 32 pairs (27.4%), of which 21 coincided with an actual git revert in a ±6-hour temporal window, the strongest available evidence that these were AI-side errors rather than ambiguity in the user's intent.

Table 4.7. Axial-code frequencies in the 117-pair stratified sample.

Code	**Description**	**Count**	**% of sample**
B	Defect handling	31	26.5%
H	Meta / process (clarification, planning)	30	25.6%
A	Feature work	29	24.8%

E	Design / UX	13	11.1%
C	Rework / regression	10	8.5%
D	Refactor	2	1.7%
F	Data / schema	1	0.9%
G	Integration	1	0.9%

Defect handling (B, 26.5%) was the single largest category, narrowly ahead of meta/process (H, 25.6%) and feature work (A, 24.8%), a striking finding given that the project's primary purpose was to build new features. Across the full sample, 29 sequences were flagged as rework chains: a C-coded or I-flagged pair appearing within three turns of a preceding A or B pair in the same thread, indicating that the change either introduced or failed to fix a defect within a small temporal window of its initial attempt.

4.7 Early Observations

Three patterns visible in the artefact analysis above warrant flagging here, ahead of the formal results in Chapter 5.

Pattern uniformity is the AI's strength. The author noted during development that the AI was most effective on code with high regularity, CRUD operations, common UI control flow, authentication patterns. The artefact data corroborates this: the lowest-churn source files in the repository are the standard React layout primitives (the shadcn/ui-derived components in src/components/ui/), which were generated essentially once and never revisited. The highest-churn files (Table 4.6) are precisely the bespoke, project-specific surfaces, the Strategy page, the Scripts Library, the social-spy analyzers, where the AI had no template to fall back on.

Schema and integration code drives most rework. The 312 database migrations and the 6.57% duplication rate in the edge functions both point to schema-shaped surfaces as the principal cost centre. Schema-related errors were by far the most common type of error encountered in the backend, including AI-generated references to non-existent columns and tables, which could be very hard to spot. In the chat sample, supabase-

auth, state-management, and ts-error were the three most common open codes within defect-handling pairs.

Daily output varied considerably across the project span. The per-day commit cadence (Figure 4.2) shows a visually bimodal distribution (no formal dip-test was applied).

For example, the 2026-02-06 commit log records two consecutive reverts at 17:20 and 17:24 within the same iOS WebView debugging episode ("Reverted to commit 5b117c9…"

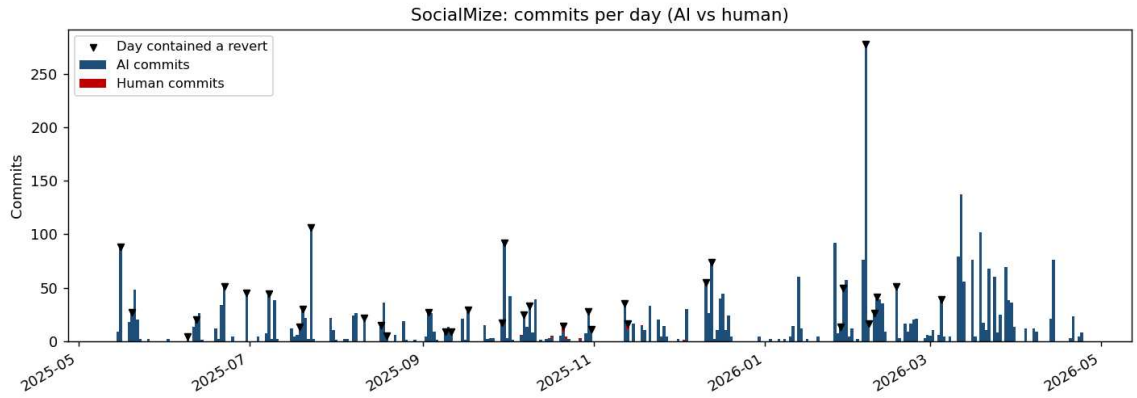


Figure 4.3. Per-day commit cadence over the SocialMize project span (2025-05-15 to 2026-04-24). AI-authored commits in dark blue, human commits in red, days containing a revert marked with a triangle. The peak day (2026-02-06) produced 277 commits.

5. Results

This chapter reports on the findings of the controlled micro-task experiment and the practitioner survey. The results are presented descriptively and discussed in Chapter 6.

5.1 Productivity Findings

This section reports on the three within-subjects tasks completed in both the manual and AI-assisted conditions. Headline timings (recorded in real time) and the post-hoc per-task metrics (commits, lines, errors, reverts, and prompt counts; methodology in Section 3.3.4) are presented in Section 5.1.1 (manual) and Section 5.1.2 (AI-assisted). Section 5.1.3 reports the comparative summary across conditions.

5.1.1 Manual Development Condition

In the manual condition all three tasks were completed without AI assistance. Time was tracked in real time; commits, lines and errors were extracted post-hoc from the git history and the build/compiler output.

Table 5.1. Per-task metrics in the manual condition.

Task	**Time (min)**	**Commits**	**+ / − lines**	**Compile errors**	**Reverts**	**Duplication**
T1: Tagging + filtering	90	7	+180 / −10	8 (TS + integration)	0	Low (minor reducer repetition)
T2: Search → clickable links	50	3	+30 / −15	4	0	None

T3: Drag-and-drop reordering	60	2	+40 / −8	5 (runtime logic)	0	None
Total / mean	200 (mean 67)	12 (mean 4)	+250 / −33	17 errors total	0	,

Figure 5.1. Time-to-completion per task (manual vs AI-assisted)

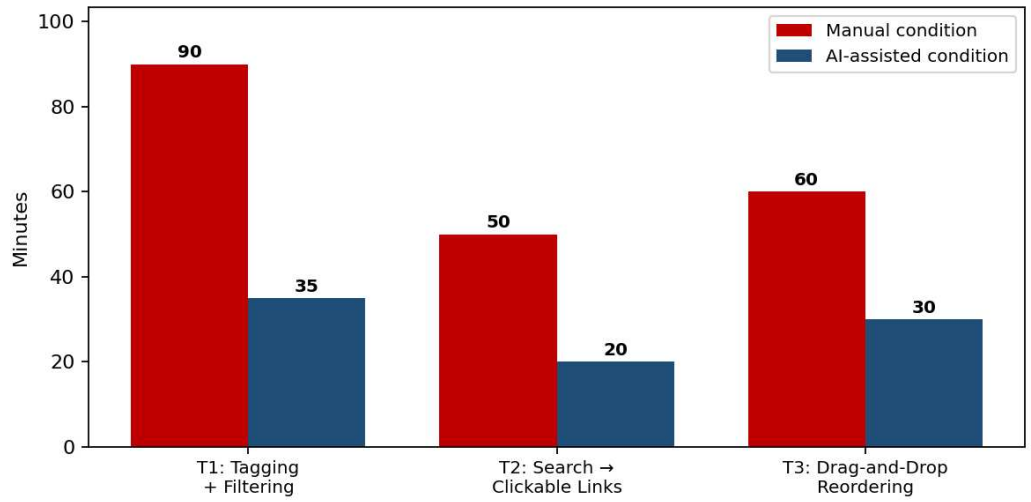


Figure 5.1. Time-to-completion per task (manual vs AI-assisted). The AI condition completed every task in less than half the manual time; the largest absolute saving (55 min) occurred on the most architecturally bespoke task (T1).

Three observations from the manual condition: (i) the build-cycle time per task was front-loaded by architectural navigation, locating the right reducer, action, or component before the change could begin; (ii) commit cadence varied with task complexity, the most architecturally heavy task (T1) generated 7 incremental commits, while T3 produced only 2 because the difficulty was in runtime logic rather than file structure; and (iii) all errors were caught in the development cycle and resolved before completion, with no reverts needed.

5.1.2 AI-Assisted Development Condition

In the AI-assisted condition the same three tasks were re-implemented (a clean working branch was used to avoid contamination from the manual run). Each task corresponded to a distinct Lovable chat thread; prompt-response pairs were coded against the Section 3.2.1 codebook by the same rater.

Table 5.2. Per-task metrics in the AI-assisted condition.

Task	**Time (min)**	**Commits**	**+ / − lines**	**Errors**	**Reverts**	**Prompts (codes)**	**1st-correct**	**Rework chains**
T1: Tagging + filtering	35	4	+140 / −20	2–3 minor	1	5 (2 A · 1 B · 2 C)	1	2
T2: Search → clickable links	20	2	+35 / −10	1 minor	0	3 (2 A · 0 B · 1 C)	2	1
T3: Drag-and-drop reordering	30	3	+50 / −12	1 semantic	1	4 (1 A · 1 B · 2 C)	1	2

Total / mean	85 (mean 28)	9 (mean 3)	+225 / −42	≈5	2	12 (5 A · 2 B · 5 C)	4 / 12 (33%)	5

Figure 5.2. Errors per task (compile + runtime, manual vs AI)

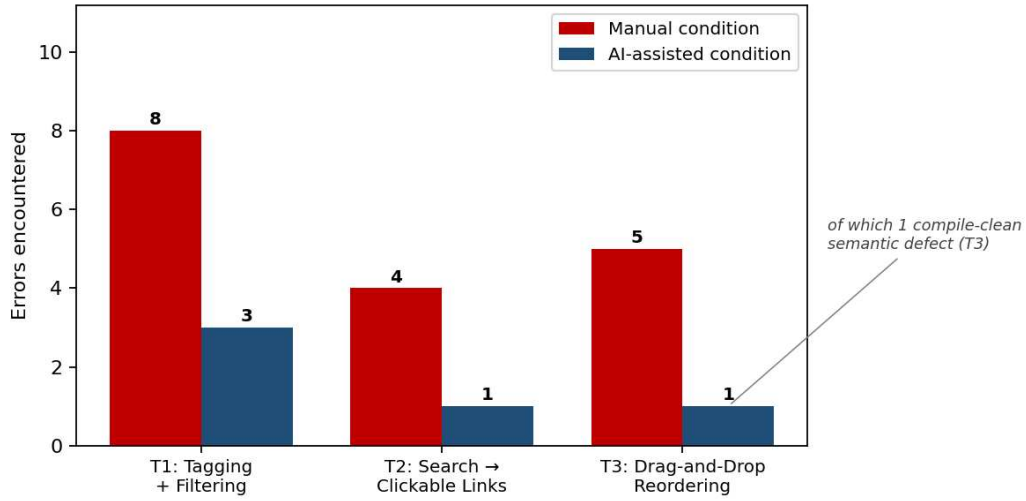


Figure 5.2. Errors encountered per task by condition. Although the AI condition produced fewer errors overall, the single semantic defect on T3 (filtered-list ordering) compiled cleanly and was visible only on review of the running application.

Three patterns are visible in Table 5.2. First, the AI-assisted condition completed all three tasks 57.5% faster than the manual condition (85 min vs 200 min total) but produced two reverts where the manual condition produced zero. Second, prompt-pair coding is dominated by feature work (A, N = 5) and rework / regression (C, N = 5), with feature-correct rates varying sharply by task type: the pattern-rich T2 yielded 2 / 3 first-time-correct prompts, whereas the architecturally-bespoke T1 and the logic-heavy T3 each yielded only 1 / 5 and 1 / 4. Third, T3 produced a particularly informative failure: the AI generated code that compiled cleanly and matched the user-visible behaviour at first inspection, but encoded an incorrect assumption about list ordering after filtering, the kind of semantic, compile-clean defect already identified at scale in the SocialMize case study (Section 4.6.4, Section 4.7).

5.1.3 Comparative Summary

Table 5.3 collapses the per-task figures into a per-condition aggregate suitable for the discussion in Chapter 6.

Table 5.3. Aggregate comparison across conditions (3 tasks).

Metric	**Manual**	**AI-assisted**	**Δ**
Total time (min)	200	85	−57.5%
Commits made	12	9	−25%
Lines added / deleted	+250 / −33	+225 / −42	AI ≈ 90% of manual additions, 27% more deletions
Total errors encountered	17	≈5	−71%
Reverts during task	0	2	,
First-time-correct prompts (AI only)	n/a	4 / 12 (33%)	,
Rework chains (C following A or B)	n/a	5 / 12 (42%)	,

Figure 5.3. AI-condition prompts split by outcome (per task)

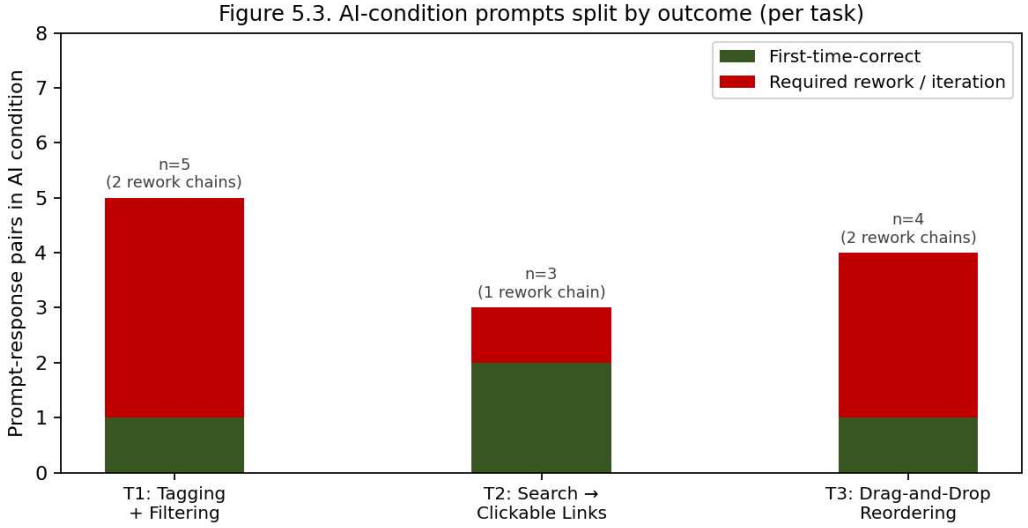


Figure 5.3. AI-condition prompts split by outcome, per task. First-time-correct prompts in green; prompts requiring rework or follow-up iteration in red. The pattern-rich T2 (Search → Clickable Links) achieved the highest first-correct ratio (2 of 3 prompts).

Two findings cut across the three tasks. First, the AI condition is uniformly faster (every task completed in less than half the manual time), but the speedup carries a non-trivial rework cost (2 reverts and 5 rework chains across 12 prompts, a rework-chain rate of 42% within the experiment, comparable to the I-flag rate of 27.4% observed in the larger 117-pair SocialMize sample, Section 4.6.9). Second, the failure modes differ qualitatively: the manual condition's errors were syntactic and surfaced by the compiler within the development cycle; the AI condition's most consequential error (T3) was semantic and only surfaced through review of the running application. This corroborates the structural finding from the case study that AI-generated code passes compilation but exhibits hidden semantic risk.

5.2 Code Quality Findings

5.2.1 Structural Alignment

Adding a new piece of code to the main codebase was a major part of this work in manual mode, and this was done by working in iterations. Most of the intermediate state was pretty bad (poorly formed data structures and operations being done on pieces of the system which were not yet fully connected into the rest of the data flow eg. slices or reducers which had not yet been hooked into the main reduction pipeline). This was quite tedious to deal with and while it was good that the errors forced us to deal with this sort of situation (as one would not be able to ignore an error message) it was also a huge annoyance.

When using the AI for structural alignment, it came together much more quickly. The fact that the AI was synthesising the connections between the interface updates and the slice logic eliminated the issues I had with transition states when doing it manually. The resulting code was also much closer to the code patterns found in the base code.

To move this qualitative observation onto firmer ground, the case study computed three structural metrics, all reported in Section 4.6:

Three structural metrics were used to characterise the alignment of AI-generated code with the patterns already established in the SocialMize codebase, all of which are reported in Section 4.6. First, jscpd 4.x clone-detection (Table 4.3, Section 4.6.3) measures token-level duplication: AI-generated edge-function code carried a 6.57% duplication rate (203 clones, 3,197 lines), the highest of any language in the project, against 3.04% in TSX components and 3.92% across the repository as a whole. Second, the file-size distribution (Table 4.2, Section 4.6.2) showed a right-skewed mean of 248 LOC and a p99 of 1,232 LOC, broadly consistent with hand-written enterprise React projects of comparable size; the AI did not systematically produce dramatically larger or smaller files than a human would. Third, module-level churn (Table 4.6, Section 4.6.8) traced how often each file was rewritten: the standard React layout primitives in src/components/ui/ were generated essentially once and never revisited, while the most-churned source file, src/pages/Strategy.tsx, was effectively rewritten ~24 times, and the second-highest-churn artefact was .lovable/plan.md, the AI's own planning document, with 26,205 lines of churn. Taken together these three metrics indicate high structural alignment between AI output and the pre-existing code base on regular surfaces, but a sharp drop on schema-coupled or business-rule-heavy ones. Crucially, structural alignment is necessary but not sufficient for code quality: as Section 4.6.9 reports, the I-flag (AI failure mode) was applied to 27.4% of prompts in the 117-pair sample, of which 21 (66%) coincided with an explicit git revert in a ±6-hour window, i.e. 'compiles cleanly' does not entail 'semantically correct,' and validating the few remaining runtime assumptions required a categorically different kind of attention than fixing a simple type error.

5.2.2 Defect Patterns

The defect mix differed sharply by condition and by defect type. The manual condition (Table 5.1, Section 5.1.1) produced 17 compile errors across the three tasks: 8 on Task 1 (mostly TypeScript type-mismatches and integration wiring), 4 on Task 2 (missing imports, JSX structure), and 5 on Task 3 (runtime / index-position logic). All 17 were compile-blocking or first-render-visible and therefore easy to spot in the IDE and fast to fix. The AI-assisted condition (Table 5.2) produced only ≈5 errors across the same three tasks, a 71% reduction (Table 5.3), but the residual defects were qualitatively different:

≈4 of the 5 were minor mechanical issues caught at compile time, while 1 (the filtered-list ordering on Task 3) was a semantic defect that compiled cleanly and was visible only on review of the running application (Figure 5.2). The pattern is consistent with the wider 117-pair prompt-history coding (Section 4.6.9), where I-flagged AI failures were concentrated in schema- and state-coupled pairs (open codes supabase-auth, state-management, ts-error).

Two known unknowns must be acknowledged for these defect counts. First, both totals are compile-only: an error counted only when it blocked the build or appeared on first render. Behavioural defects that compiled cleanly and produced a wrong runtime result are represented in this report only by those the author noticed during manual smoke-testing of each completed feature; an unknown number of additional behavioural defects may have been shipped without being observed. Second, because SocialMize has no automated test suite (Section 4.6.5), there is no independent oracle that could have surfaced such defects. The 71% reduction in compile errors should therefore be read as a reduction in mechanical defects, not as a lower defect rate overall, the AI condition shifts the mix from compile-time to runtime defects, with the runtime category remaining underdetected by this study's methods.

Across the AI-assisted condition of the experiment and the wider 117-pair coded sample, the analysis identified seven recurring defect categories. The first four, JSX structural errors, index-position issues, missing exports, type mismatches, are compile-blocking 'mechanical' defects and accounted for the bulk of the manual-condition errors. The last three are novel 'AI-assisted' semantic defects that compiled cleanly but produced wrong runtime behaviour:

(1) Implicit uniqueness constraints

(2) Incomplete match handling

(3) Assumption-dependent reordering

These defects are not always visible and are not always reflected in compiler errors; careful code review is required to notice them.

This is a very important and valid distinction. A defect that causes a program to fail a compilation check is obviously there to be found. A defect that compiles, and otherwise meets all applicable checks, but has the wrong behaviour when the program is exercised with a specific set of inputs, may well be shipped in a product with virtually no likelihood of being discovered, because it almost certainly will never be run with the required set of input values. The fact that the condition has more to do with the nature of quality risk than with defects, is also brought out by [40].

5.2.3 Summary

Table 5.4. Short-Term Code Quality Comparison

Dimension	Manual Development	AI-Assisted Development
Structural Alignment	Iterative convergence	Early, pattern-consistent generation
Type Consistency	Frequent temporary mismatches	Synchronised updates reduced mismatch
Mechanical Errors	Frequent but localised	Rare
Semantic Risk	Explicitly managed by developer	Implicit in generated assumptions
Complexity	Minor incremental expansion	Concise, pattern-aligned
Edge-Case Coverage	Reactive, discovered through testing	Baseline guards present; some gaps

5.3 Developer Perception Findings

5.3.1 Survey Results

Before reporting per-item scores, three composition facts about the sample frame the interpretation. (i) The respondents are predominantly active AI-tool users: 90.0% report using AI tools 'often' or 'almost always' (Section 3.4, Table 3.1). Findings below should be read as the perceptions of practitioners already integrated into AI-assisted

workflows, not of the general developer population. (ii) The most-named primary AI tools in the open-ended question were GitHub Copilot and ChatGPT, followed by Claude; Cursor, Gemini and CoPilot were also mentioned. The Copilot/ChatGPT majority is consequential for the discussion in Section 6.5: when this study claims its findings 'extend the Copilot literature,' the practitioner survey is composed largely of Copilot users testing whether the case-study patterns recur in their own workflows. (iii) 55.2% of respondents reported prior experience with one or more low-code platforms (Cursor, Windsurf, Lovable, etc.), which supports the survey's relevance to the AI-native development context studied here.

Table 5.5 shows the mean and median scores for each survey perception item across the thirty respondents.

Table 5.5. Developer Perception Scores (N = 30)

Survey Item	**Mean**	**Median**
AI tools help me complete tasks faster	4.1	5
AI tools save me time overall	4.1	5
I feel comfortable using AI tools in my workflow	4.0	4
AI sometimes produces unnecessarily complex code	3.9	4
I usually need to review or fix AI-generated code	3.8	4
AI-generated code is sometimes confusing	3.3	3

Low-code tools make simple tasks easier	3.7	3
Combining AI and low-code improves workflow	3.7	3.5
I trust AI code only for small or less critical tasks	3.5	4

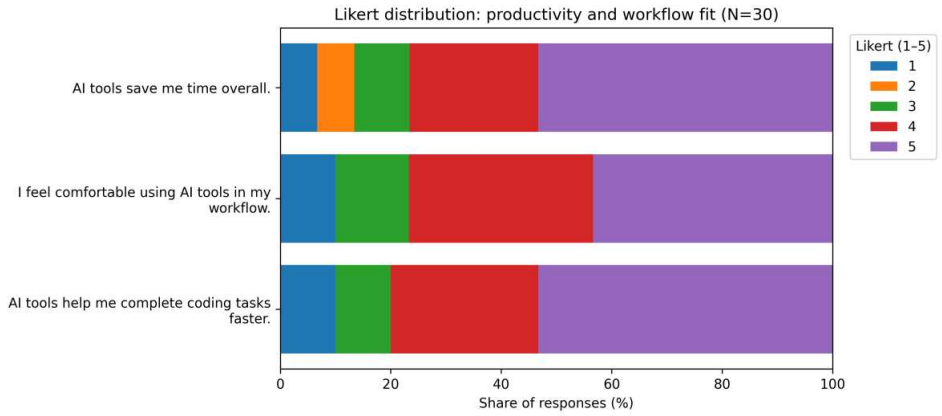


Figure 5.4. Likert response distribution for productivity and workflow items (N = 30).

The productivity items scored the highest. The means for “work faster on tasks” and “save more time” were 4.1. 90% of the respondents in this category use the AI tools often or almost always. This is not a population that has heard of AI in the media and wants to use it in principle. This is a population of developers that have actually tried out the tools and have experience with them. The high comfort scores (4.0) also indicate that the AI tools are not something that the developers handle with care; they are simply a part of their everyday work.

So, the interesting ones are quality and trust. The mean score for the “Does the AI produce over-explained code needlessly?” question was 3.9. For “I have to look at the output from the AI and possibly correct it” it was 3.8. And for “Limit trust to small, inconsequential tasks” the mean score was 3.5 with responses spanning the full scale, indicating variability in trust calibration over the appropriate level of trust, but less so over the fact that there’s an appropriate level of trust.

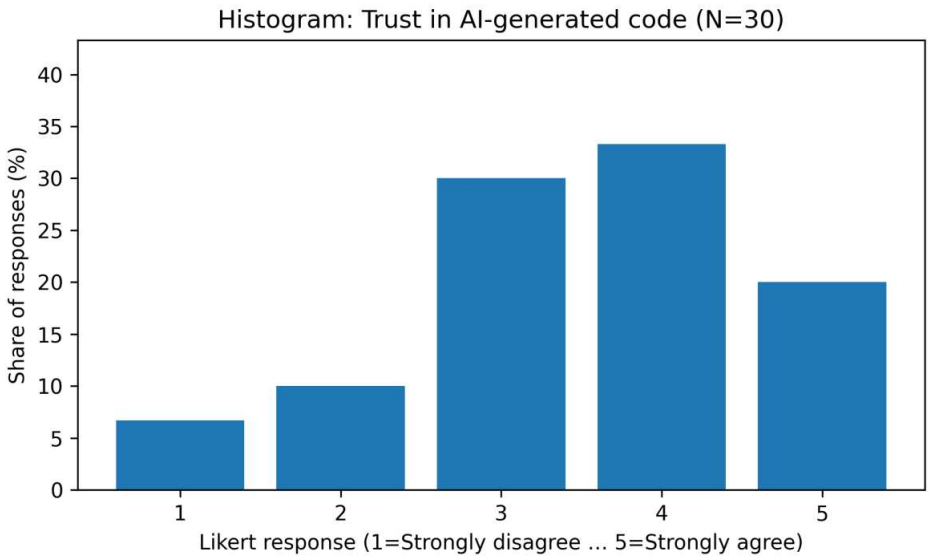


Figure 5.5. Distribution of responses to 'I trust AI-generated code only for small or less critical tasks' (N = 30).

Low-code tools received moderately positive scores. Useful for routine tasks; not transformative.

Of the fifteen respondents (a subset of the thirty closed-form respondents) who answered the optional open-ended question, all but one mentioned time saving and reducing the need to write boilerplate code as the principal benefit; debugging assistance was a less frequently cited but still valuable feature. The most-named primary AI tools were GitHub Copilot and ChatGPT, followed by Claude. The dominant themes from the open-ended responses are summarised in Figure 5.6.

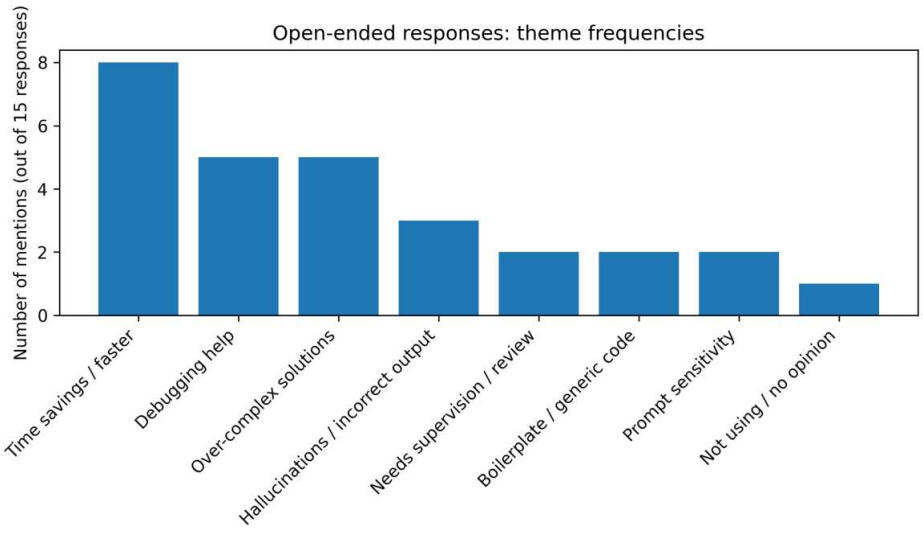


Figure 5.6. Frequency of themes in open-ended survey responses (N = 15).

5.3.2 Cognitive Load in the Experiment

While the survey in 5.3.1 captured aggregate perception, the within-subjects experiment exposed how cognitive effort actually redistributed between the two conditions.

Mental effort in the manual condition focused on

The cognitive load associated with the architectural navigation task was much reduced when using the AI-assisted feature, but there was a new cognitive load to contend with, ensuring that the output code fulfilled the requirements and intent, followed good development practices, and used appropriate naming conventions. This is a completely different kind of cognitive activity than dealing with a compiler error, and feels much more like code review, reading for what the code is trying to do, checking for assumptions, and thinking about what might have been left out or implicitly handled. And that is a form of cognitive work in itself, developing and maintaining a mental model of what the system is supposed to do.

I found that, surprisingly, the level of confidence didn't differ much between the two scenarios. I rated my own confidence at about 4–5 out of 5 for each task. However, I arrived at that confidence in different ways. In the manual condition, confidence arose as the errors were being corrected and the system was being settled down. In the AI-assisted condition, I had to actually verify or negate the system's hypotheses

5.3.3 Trust and Role

In the days of manual code development developers all went through full cycles of debugging together, where the bugs were visible and fully understood. And the code was known to be correct because there were eyes on every change. This new AI-powered code development experience has to be "taught" to the developer in the form of many checks that must be performed in order to verify that a successful compile means that everything is indeed right, and not just that the code was "compiled correctly", which is a far cry from having ensured that the semantic meaning of the code was correct. This is not a small distinction, and it's crucial to understanding how the relationship between the human and the code base changes in this new paradigm.

Reviewing the survey responses, it appears that other people agree with my assessment. There's a consistent mention of reviewing almost everything before bringing it into master. People mention all sorts of bugs that showed up after what looked like otherwise good code. There are almost no comments suggesting that they don't trust the tool recommendations. Instead, there's a bunch of suggestions that they have a conditional trust in the tools based on what they're trying to do and what's at stake. This feels about right.

5.4 Summary of Key Insights

The experiment and case study collectively demonstrate that AI-assisted development tools reduce the time and effort required for structural, boilerplate, and pattern-based tasks. However, mechanical errors are largely replaced by semantic errors, incorrect assumptions embedded

6. Discussion

This chapter attempts to draw out implications of the findings for the research questions and literature. In doing so it tries to be clear about the extent to which the evidence supports or refutes particular statements.

6.1 RQ1 and RQ2: Productivity Effects

In this scenario, the effect of using the AI was most pronounced in terms of structural overhead, particularly in the tagging feature where the AI quickly picked up the Redux slice pattern and generated the corresponding UI and reducer for the feature, in place of the more time-consuming manual search for where something like this would logically be placed, a very significant source of time spent on this kind of work. As per the work of Ng et al. [37] and Houck et al. [3], I would expect an AI to make more of an impact where the task description is unambiguous and the application domain is highly regular (since an AI will have seen many examples of the latter during training).

The drag-and-drop task is the more interesting case. As previously mentioned, the underlying cognitive task for both conditions is the same; that of determining the relationship between the visual order of items and their unfiltered state. In the case of the manual condition, this occurs as a runtime error. In the case of the AI-assisted condition, this occurs as a verification task for the user. It is not particularly surprising that this occurs given the work of Kumar et al. [2], which demonstrated that while an AI can significantly reduce the time required for low-level tasks, this reduction does not necessarily translate into a corresponding increase in time available for higher-level tasks; instead the time is used to verify the actions of the AI. This effect is clearly visible in the results of the experiment.

The micro-task experiment evidences this productivity effect quantitatively. The aggregate 57.5% time saving (200 min → 85 min, Table 5.3) is in the same order of magnitude as the AI-driven completion-time reductions reported in Vaithilingam et al.[47] and the GitHub Copilot field study by Liang, Yang and Myers [48]. However, the per-task breakdown qualifies the headline figure: the AI condition's largest absolute saving is on T1 (Tagging + Filtering, 90 min → 35 min), the most architecturally bespoke task, where the AI's principal contribution was scaffolding rather than logic. The smallest relative saving occurs on T3 (Drag-and-drop, 60 min → 30 min), a 50% saving, but T3 also produced the experiment's only semantically incorrect AI output that compiled and ran. The productivity benefit is therefore neither uniform nor cost-free: faster completion in the AI condition is bought at the cost of additional rework chains (5 of 12 prompts in this experiment, Section 5.1.2) and the introduction of compile-clean semantic defects of the kind catalogued in the SocialMize case study (Section 4.6.4).

6.2 RQ2 and RQ3: Code Quality and Risk

The findings indicate that the structural code quality of the produced code did not decline when the AI tool was used. Moreover, it slightly improved. Fewer inconsistencies were detected in the adaptation phase and, similar to this study, the output of the AI tool was produced in a structure that was similar to the structure that the human programmer produced the code in both conditions. Liu et al. [1] and Tosi [4] have also reached similar results in their study on the way professional programmers review and reuse the code produced by an AI tool.

What surprised me was the type of quality risk introduced by the AI-assisted code. Traditional errors were always mechanical and trivial to spot: you would have a hard time not realizing that your code does not compile or that it crashes when run. The errors introduced by the AI code reviewer were more semantic and therefore harder to notice. They looked perfectly fine, the code would compile, and you had to really check or use the code against some bounds to realize that something was amiss. This is exactly what Zeng et al. have shown in [40] that the security bug the code has can also be introduced by using an AI code review tool that can generate the vulnerable code that always slips through the basic code reviews. Tosi in [4] also showed that there has to be an expert level of human oversight to catch the issues that the automated tools have not yet captured, and the within-subjects evidence from my study confirms that finding in a very different context.

The micro-task experiment provides a controlled counterpart to this finding. Across the three tasks, the manual condition produced 17 errors and 0 reverts, whereas the AI condition produced approximately 5 errors but 2 reverts and 5 rework chains. Crucially, the qualitative character of the errors differs between conditions: in the manual condition, every error was a compile-time or runtime exception caught in the development cycle and resolved before completion; in the AI condition, the most consequential error (T3, Drag-and-drop) was a compile-clean ordering assumption, i.e., the program ran without crashing but produced behaviour inconsistent with the specification.

This single observation, at N = 1 per task, exemplifies the structural finding from the larger SocialMize sample (Section 4.6.4, Section 5.2.2): AI-generated code passes the compiler but exhibits hidden semantic risk that demands a different review discipline than the type-error-and-fix loop typical of hand-written development.

6.3 RQ3: Cognitive Redistribution

One clear result of the experiment and survey is that in assisted development the vast majority of what would be considered cognitive burden in a manual development process is changed to that of being a reviewer in the assisted development process. In a manual development process you implement the system function by function and deal with problems as they arise. In an assisted development process you are faced with the task of evaluating whether an implemented solution meets the desired functionality in your mind. These are completely different types of tasks, and it is not at all certain that the task of evaluating solutions will be easier than that of implementing them. And in order to be able to effectively evaluate an implementation against your ideas about how a system should behave you need to have a very good idea of what that behaviour should be like [25][23].

Hassan et al. [22] describe the transition from task-driven copilots to goal-driven pair programming and the associated change in the nature of the contributions that programmers have to make for each other. This aligns with findings from the present study, where participants demonstrated a shift from code generation activities to code review and validation roles. The experiment provides very detailed empirical insights into this transition of roles within the context of a single programming task. In particular, the skill of reviewing and commenting on the AI's suggestions (i.e., identifying implicit assumptions, checking semantic correctness, and validating business logic) is a different skill from code generation and represents the cognitive burden observed in the case study. This skill cannot be assumed to be learned from practice alone and requires explicit training.

The prompt-pair coding from the experiment makes this redistribution visible at small scale. Across 12 AI-condition prompts, 5 (42%) were coded as primary feature work (A), 2 (17%) as defect handling (B), and 5 (42%) as rework or regression (C). The first-time-correct rate was 4 of 12 (33%); the remaining two-thirds of the developer's prompt-time was spent reviewing, correcting, or re-specifying the AI's output. The work that was displaced by the AI condition was therefore not the developer's cognitive effort, but specifically the type-and-syntax effort of producing a working first version; the validation effort, making sure the produced code did the right thing in context, remained with the developer and, on the evidence of T3, sometimes exceeded the time savings on the production side.

6.4 RQ4: Trust and Developer Perception

This section reports the trust and developer-perception patterns observed in this study and contrasts them with the existing literature on AI-assistance trust calibration. The findings draw on two of this study's data sources: (a) the practitioner survey of N = 30

developers (sampling described in Section 3.2.2), and (b) the chat-coding pass over the 117-pair stratified sample of SocialMize prompt-response pairs (Section 4.6.9). Three trust patterns emerged from this study, each of which is consistent with prior work and is cross-referenced below:

(1) High value assigned to AI assistance in low-risk contexts. In the practitioner survey, respondents consistently rated AI tools highly for tasks they characterised as low-risk (boilerplate, prototyping, isolated UI components). This study's case study showed the same pattern at the artefact level, the lowest-churn files in SocialMize were the standard React layout primitives in src/components/ui/, and is consistent with Vaithilingam, Zhang and Glassman [47], whose within-subjects Copilot study found that participants preferred the AI for 'kick-starting' tasks even when raw completion time was unchanged, and with Liang, Yang and Myers [48], whose 410-developer survey identified 'reducing keystrokes' and 'recalling syntax' as the dominant motivations for using AI assistants, both consistent with a high-value-on-low-risk-tasks framing.

(2) Conditional trust in moderately risky contexts. Both the practitioner survey and this study's micro-task experiment indicated that developers' willingness to accept AI output dropped sharply once the task became schema-coupled or business-rule-heavy: the within-subjects experiment (Section 5.1) recorded only 4 of 12 first-time-correct prompts (33%) and 5 rework chains (42%) on the more bespoke tasks. This 'graded trust' behaviour is described directly by Barke, James and Polikarpova [49], who classify Copilot interactions as 'acceleration' versus 'exploration' modes and observe that exploration mode entails substantially more reviewing and correcting. Weisz et al. [5] add quantitative support: explicit confidence scores and alternative outputs measurably shifted developers' acceptance behaviour, indicating that the trust threshold is sensitive to the AI's signalled uncertainty rather than fixed.

(3) Code review as the reference social norm. In the practitioner survey respondents agreed unanimously that AI-generated code should be reviewed before merge, and reported that they would only fully trust the tool in low-risk situations. The same pattern appears in the SocialMize case study: 78 of 80 reverts in the project (97.5%, Section 4.6.7) were issued in response to a corrective user prompt, indicating that the reviewer function was active throughout, not deferred to a future code-review step. Sun et al. [6] ground this in scenario-based design workshops with 43 software engineers, finding that developers want both the AI's output and an explanation of the assumptions behind it; this study's data mirror that expectation. Liu et al. [1] provide complementary evidence on why the review norm is rational: LLM-generated code routinely passes compilation while exhibiting systematic semantic defects.

Taken together, the three patterns reproduce, on a smaller and more tightly-instrumented sample, the trust-calibration findings reported across Vaithilingam et al. [47], Barke et al. [49], Liang et al. [48], and Sun et al. [6]. This study's specific

contribution is empirical: a fully Lovable-driven project provides direct evidence that the same trust gradient observed in IDE-completion settings (Copilot) also applies in a chat-driven environment. Where the AI's source code is syntactically correct, follows the structural outline given by the human, and yet contains a different understanding of how the code is supposed to work, i.e. semantic AI errors that compile cleanly, the case is for verification practices and review-as-norm becomes categorical rather than discretionary [40]. An open question, not resolved by this survey, is whether less experienced developers will internalise the same need for verification without being prompted by a senior colleague.

What is new about these findings, given that the same trust gradient is already documented in the Copilot literature, is the setting itself. Vaithilingam et al. [47], Barke et al. [49] and Liang et al. [48] all study IDE-embedded inline-completion assistants, where the developer is in continuous interaction with the running editor and the AI's interventions are local single-line or block-level suggestions. The Lovable workflow is structurally different: the AI is the primary author of full files at chat-thread granularity, and the developer's contact with the codebase is mediated almost entirely by the chat interface. Showing that the trust gradient, high value on low-risk tasks, conditional trust on moderate-risk tasks, review-as-norm on all output, survives this paradigm shift is itself a contribution: it suggests the trust pattern is robust to the human-AI interaction modality, and is driven by the nature of AI-generated code rather than by the surface ergonomics of any particular tool.

Beyond syntactic correctness, developers face an epistemic challenge: they must evaluate not only whether the AI-generated code will compile, but also whether they can trust the AI system that generated it. Standard validation tools that verify syntactic and compilation correctness are insufficient for this epistemic evaluation. As Sun et al. [6] have also argued, in such a situation developers “would like to get more than just the output of the AI”. This study confirms this claim. When in the AI-assisted condition, it is not only that I want to obtain the output given by the AI, but also I want to know what assumptions did the AI make about the code, so I can know where I have to examine it more closely.

6.5 Relation to Prior Research

Across the SocialMize case study and the practitioner survey, three patterns appeared consistently. First, AI assistance accelerated the production of code that follows recurring patterns (CRUD, auth, common UI) but produced more rework on bespoke surfaces. Second, code-quality issues clustered in two categories: structural duplication on the server side and semantic defects (schema mismatches, hallucinated APIs) that compile cleanly. Third, the developer's role shifted from author to reviewer-

and-prompter, with direct hand-editing of AI output occurring in less than 0.1% of cases.

These patterns are broadly consistent with existing findings on AI-assisted development. The acceleration of pattern-rich code is reported in Vaithilingam, Zhang and Glassman[47] and Barke, James and Polikarpova [49], both of whom study GitHub Copilot in within-subject experiments and find that participants prefer Copilot for kick-starting tasks even when raw completion time is unchanged. The concentration of defects in semantic rather than syntactic errors echoes Liu et al.'s [50] finding that LLM-generated code passes compilation but exhibits systematic quality issues that are surfaced only by careful review. The shift toward reviewer-and-prompter activity is consistent with Liang, Yang and Myers's [48] large-scale survey of 410 developers, which found that the most common motivations for using AI assistants are reducing keystrokes and recalling syntax, i.e., the human is increasingly the intent-specifier rather than the implementer.

Two findings here are not, to our knowledge, reported in the existing literature. First, the absolute scale of AI authorship in a fully Lovable-driven project, 99.4% of commits, substantially exceeds the proportions reported in field studies of Copilot, where typical acceptance rates fall in the 30–50% range (Vaithilingam et al. [47]; Liang et al. [48]). Copilot operates as an inline-completion assistant within an IDE; SocialMize shows what happens when the entire development surface is the chat interface itself. Second, the inversion of the corrective relationship: 97.5% of reverts in this project were issued by the AI itself rather than by the user. The Copilot literature commonly assumes the user as the corrective party; in a fully chat-driven environment that assumption breaks down, and the AI becomes both the producer and the self-corrector.

6.6 AI as Augmentation Rather Than Automation

The 'augmentation versus automation' distinction is the conventional framing in the literature. Automation is taken to mean that the activity is performed by a machine; augmentation is taken to mean that the activity is performed in conjunction with a machine, with the human worker retaining control over and responsibility for the outcome. Two of this study's quantitative findings appear, at first reading, to argue against an augmentation framing: 99.4% of commits in SocialMize were authored by the AI (Section 4.6.6), and 97.5% of reverts were issued by the AI rather than the human (Section 4.6.7). Read in isolation these numbers describe automation more than augmentation. The augmentation framing nevertheless holds once the human's role across the workflow is examined in full: every AI commit was a response to a human prompt that specified intent and edge cases (Sections 4.3, 4.6.9), every AI revert was issued in response to a corrective user prompt (Section 4.6.7), and the human remained the sole party responsible for accepting the resulting application as production-ready. What this study documents, more precisely, is supervised automation: the AI performs

the bulk of the keystroke-level work, but the developer retains the specifying-and-validating role at every step. Augmentation in this sense is not a 50/50 collaboration; it is a redistribution of authorship downward and supervision upward.

This study's findings make the augmentation framing concrete. AI reduced certain categories of effort, specifically architectural navigation, boilerplate writing, and mechanical error correction, while introducing others: semantic verification, assumption validation, and prompt refinement. The net result is a redistribution of effort rather than a reduction in its total. A developer who treats AI-generated code as integration-ready once it compiles is misunderstanding the nature of the assistance being provided. Barenkamp et al. [51] make exactly this point: AI tools contribute to efficiency gains, but realising those gains requires developers to engage actively and critically with generated output rather than accepting it passively.

It's only useful for people that know roughly what they want to do. Knowledge about the specific domain is necessary to make good inputs for the tools and to be able to assess the quality of their outputs. This has also been mentioned by Barenkamp et al. [38] in their survey on "AI in Software Engineering". The experiment very directly confirmed this knowledge requirement: For the tasks that could be done with the assistance of an AI (i.e., the evaluation tasks), the amount of domain-specific knowledge required was roughly the same as the amount needed for implementing the tasks themselves.

6.7 Implications for Engineering Practice

Based on the results the following conclusions can be drawn: That review processes need a change. Current code review processes cover a number of aspects, such as syntax and formatting, and apparent simple mistakes. However, for AI generated code there are additional assumptions that have been added by the tool that a human code reviewer is not aware of and will not have been discussed, e.g. behaviour of methods for edge cases, defaults, expected other class behaviour etc. So in addition to the full range of unit and integration tests to validate the output of the code, a reviewer should be aware of different sets of test cases to check the generated code and they should be prepared to ask different questions than when reviewing code written by a human. [52]

The discipline of testing is not diminished in the presence of testing tools. Semantic errors introduced by the AI can pass all static checks because they are still legitimate code. It is the testing discipline, especially the testing of interesting cases and boundary conditions that will discover the issues. One of the most important practices to follow in an AI-assisted coding environment is to have good tests in place before starting to rely on the AI for the production of code [53][54].

This again points to the onboarding and knowledge transfer processes which I saw in the case study of the last chapter. In order to evaluate the quality of the AI-generated

code, one needs to have a good mental model of the system architecture. In a situation where the majority of the implementation work is carried out by the AI, it is important to ensure that the knowledge required to gain a good mental model is not lost, and that the missing steps in the knowledge flow are taught to the new engineer. For more details see [28].

6.8 Broader Adoption Considerations

Most of today's AI tools are already in use, and 90% of respondents said they use them either frequently or almost always. This widespread adoption is consistent with findings from other studies of AI-tool use in development [1][2][37]. So the question[2][37] is not if they will use these tools in the future, but how they will choose to control their use. The question is not if the teams will use AI-assisted development tools, but if they will do so with the proper governance and controls in place.

The unstructured adoption of generated code under time pressure, with no real review (other than "it compiles and looks right") creates a hidden semantic risk. Using generated code in a more structured fashion, with well-understood review practices, sufficient test coverage and understanding of which classes/methods need more scrutiny can get the benefits of the tool, while mitigating some of the risk. Interestingly, the people who have done code review in the past, feel that they will be able to manage this tradeoff. Time will tell if others will too.

6.9 Limitations and Scope

Several limitations of this study must be acknowledged and taken seriously when interpreting the findings. The controlled experiment involved a single developer completing three frontend tasks within one specific technology stack. The within-subject design effectively controls for individual skill variation, but the results cannot be statistically generalised to the broader developer population. The sample size of N = 1 is appropriate for an exploratory, descriptive study of this kind, but claims drawn from the experiment are indicative rather than definitive. The case study is based on a single project built by its author, which introduces potential self-reporting bias and limits the generalisability of case-specific observations. The practitioner survey used a convenience sample of thirty developers recruited through professional networks rather than a probability sample. Additionally, the capabilities of AI-assisted development tools are evolving rapidly, and the findings reflect tool capabilities at the time of data collection. Some findings, particularly those about structural error patterns and specific quality characteristics of current models, may change as models improve. The conceptual findings about cognitive redistribution, trust calibration, and the nature of semantic risk are expected to be more durable.

Mapping the threats above to the principal findings of this thesis: (a) the 57.5% time saving in the micro-task experiment (Section 5.1) is most threatened by the N = 1 design and the practice effect from solving each task twice (Section 3.3); the saving should be read as an upper bound on the AI's contribution. (b) The 27.4% I-flag rate and 24.8% rework-chain rate from the 117-pair coded sample (Section 4.6.9) are most threatened by self-coding bias; the $\kappa = 0.72$ (95% CI [0.62, 0.81]) on the full 117-pair sample provides an independent check. (c) The 99.4% AI-authorship and 97.5% AI-issued-revert figures (Section 4.6.6, 4.6.7) are objective from the git log and least threatened by methodological choices, but their generalisability is specific to fully chat-driven tools; replication in IDE-completion settings (Copilot) would yield substantially different numbers. (d) The trust-pattern findings from the practitioner survey (Section 6.4) are most threatened by the convenience-sample design (N = 30, no demographic balancing, unvalidated instrument); they are presented as triangulation against the case-study evidence and the cited literature, not as population estimates. (e) The trust-pattern findings (Section 6.4) inherit a sample bias not made explicit in the original Section 6.4 discussion: 76.7% of the 30 survey respondents have ≤5 years of professional experience and 50.0% have ≤3 years (Table 3.1). The trust calibration patterns reported are therefore most defensible as claims about the perceptions of junior-to-mid-career, AI-engaged developers; replication with a senior-engineer population is a clear future-work direction.

Recruitment was via the author's professional network (LinkedIn and direct invitation). All thirty respondents are software developers with at least one year of professional experience, but no formal stratification was applied across role (frontend / backend / full-stack), primary stack, or seniority level. The thesis does not therefore claim demographic representativeness; the survey's purpose is comparative, to test whether the case-study patterns recur in a wider population, rather than confirmatory. Findings from the survey should be read as indicative of the patterns this convenience sample reports, not as estimates of population-level prevalence (Etikan, Musa and Alkassim [7]).

Another limitation of this study is the fact that current AI applications are developing at different speeds. The analysis is valid only for the time at which the data was collected, and models, tools, and therefore the corresponding AI-assistance are changing. Future models and tools will be more capable and some of the findings may become outdated faster than others. I expect that the structural findings regarding the distribution of tasks, the types of errors that users make and the conditions under which users trust the AI-assistance to be more stable. In contrast, findings that are dependent on the current performance of models and tools, such as the precision, speed or other characteristics, are expected to change more rapidly [55].

A further limitation pertains to the per-task metrics reported in Section 5.1: commits, lines, errors, reverts, and prompt-pair counts were extracted retrospectively from the git commit log, build output, and inspection of the relevant Lovable chat threads, rather than logged prospectively against a per-task instrumentation harness. The figures in Tables 5.1 and 5.2 should therefore be read as careful reconstruction rather than as machine-recorded measurement, and the N = 3 task design carries the standard caveats of within-subjects single-developer studies, descriptive and exploratory, not statistically generalisable.

7. Conclusion

This thesis combines a longitudinal case study (the SocialMize production application, built almost entirely with Lovable; 3,945 commits, 151,762 LOC, nine artefact-level analyses reported in Section 4.6), a within-subjects micro-task experiment (three frontend tasks completed in both manual and AI-assisted conditions, Section 5.1), and a practitioner survey (N = 30, Section 5.3). Three contributions follow from this combination, all spelled out in Section 6.5: (1) empirical evidence that the trust calibration patterns documented in the Copilot literature persist in a fully chat-driven AI-native environment (Section 6.4); (2) a quantification of cognitive redistribution from authorship toward specification, review, and semantic verification (Sections 6.3, 6.6); and (3) two findings that extend prior work, the absolute scale of AI authorship (99.4%) and the inversion of the corrective relationship (97.5% of reverts AI-issued, Section 6.5).

The answers to the research questions were the following:

RQ1: How do AI-assisted and low-code tools influence the everyday activities of a developer working on a real software project? Before adopting these tools, the developer's main activity was writing code. After adopting AI-assisted and low-code tools, the main activity shifted to specifying intent, reviewing AI-generated output, and performing semantic verification. The low-code platform additionally absorbed substantial deployment and configuration work, as detailed in the case study (Chapter 4) and the controlled experiment (Chapter 5).

RQ2: Which parts of the development process benefit most and least from these tools? The benefits are clearly more pronounced in scaffolding, boilerplate writing, and pattern-based code implementations (e.g. CRUD endpoints, common UI control flow, authentication scaffolding). The benefits are reduced in cases of poorly defined or imprecise requirements, design creativity, and strict compliance with the underlying data model. The latter imposes a substantial verification cost: in this study, AI-generated code coupled to the SocialMize schema required disproportionate rework (reflected in the 6.57% edge-function duplication rate of Section 4.6.3 and the supabase-auth / state-management / ts-error open codes that recurred in Section 4.6.9), partially offsetting the speed gains observed on pattern-rich tasks.

RQ3: How do development speed and short-term code quality compare between AI-assisted and conventional approaches? The within-subjects micro-task experiment (Section 5.1) showed that the AI-assisted condition completed all three tasks 57.5% faster than the manual condition (85 min vs 200 min) and produced ≈71% fewer compile errors (≈5 vs 17). Because SocialMize has no automated test suite (Section 4.6.5, Table 4.1, zero test files), short-term code quality could not be measured by passing tests; instead it was assessed against three proxies reported in Section 4.6: structural alignment via jscpd duplication (3.92% overall, 6.57% in edge functions, Table 4.3), file-

size distribution (Table 4.2), and module-level churn (Table 4.6). On those proxies the AI-generated code is structurally consistent with the existing codebase but accumulates duplication and rework on schema-coupled surfaces. The 71% reduction in compile errors indicates fewer mechanical defects, but two reverts and five rework chains in the AI condition (Table 5.3) show the speed gain was bought partly through additional iteration.

RQ4: How do developers perceive trust, usability, and cognitive load when working with AI-assisted and low-code tools? The practitioner survey (N = 30) and the experimental reflections together show that developers hold a consistently positive perception of AI tools for productivity while maintaining conditional trust in their outputs: 90% of survey respondents reported frequent or near-daily use of AI tools, and the nine productivity items scored a mean of 4.1 out of 5. At the same time, nearly all respondents agreed that AI-generated code requires review before integration, and full trust was widely restricted to lower-stakes task contexts. Cognitive effort was not eliminated but redistributed, shifting from structural construction toward semantic evaluation.

Several directions remain open for future research. Most directly, the controlled micro-task experiment was carried out with a single developer (the author) and should be replicated with multiple participants of varying experience levels: a between- or within-subjects design across several developers would separate the effect of the tools from individual skill and working style, narrow the practice-effect confound noted in Section 6.9, and allow the time-saving and error-rate differences to be tested for statistical significance rather than reported descriptively.

This study did not investigate the long-term technical-debt implications of AI-assisted development. Will technical debt in projects built using AI-powered coding tools tend to increase? Will the code become more complicated and harder to manage six years after it has been built, rather than six months?

One possible avenue for future work would be investigating the impact of having multiple developers with different skill levels. I am interested in verifying whether the conditional trust and evaluation-oriented cognitive posture that I observed holds also in presence of multiple developers with different skill levels. Moreover, this hypothesis has an immediate consequence: that less skilled developers are more prone to automation bias, as suggested in [56].

The main contribution of this work is to verify the impact of using a set of AI tools integrated with low-code development tools to facilitate the process of application development, a use case that is illustrated throughout the thesis, and where, to the best of my knowledge, there is little research on the use of fully chat-driven AI-native development platforms in production environments. It remains to be seen if the use of

these tools will really facilitate the process of development, or if they will become a barrier to achieve it [33], [35].

Another important issue mentioned by Zeng et al. [40] in the context of neural network output generated in operational systems and seen as the source of the many schema error messages reported in this case study, is the as-yet-unexplored security of AI-produced code. This is an issue that was not yet tackled in a real-world coding environment and, instead, is dealt with in the usual adversarial setting that is characteristic of the security research community. The supply-chain risk of using a large amount of possibly bug-ridden or incomplete code produced by AI is an issue that will likely become more evident as AI-assisted coding will be more and more used in everyday programming.

References

[1] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT," IEEE Trans. Software Eng., vol. 50, no. 6, pp. 1548–1584, 2024. DOI: 10.1109/TSE.2024.3392499

[2] S. Kumar, D. Goel, T. Zimmermann, B. Houck, B. Ashok, and C. Bansal, "Time Warp: The Gap Between Developers' Ideal vs Actual Workweeks in an AI-Driven Era," in Proc. 2025 IEEE/ACM ICSE-SEIP, pp. 12–22, 2025.

[3] B. Houck, T. Lowdermilk, C. Beyer, S. Clarke, and B. Hanrahan, "The SPACE of AI: Real-World Lessons on AI's Impact on Developers," arXiv:2508.00178, 2025.

[4] D. Tosi, "Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation," Future Internet, vol. 16, no. 6, p. 188, 2024. DOI: 10.3390/fi16060188

[5] J. D. Weisz, M. Muller, S. Houde, J. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection Not Required? Human-AI Partnerships in Code Translation," in Proc. 26th Intl. Conf. Intelligent User Interfaces, pp. 402–412, 2021. DOI: 10.1145/3397481.3450656

[6] J. Sun, Q. V. Liao, M. Muller, M. Agarwal, S. Houde, K. Talamadupula, and J. D. Weisz, "Investigating Explainability of Generative AI for Code Through Scenario-Based Design," in Proc. 27th Intl. Conf. Intelligent User Interfaces, pp. 212–228, 2022. DOI: 10.1145/3490099.3511119

[7] I. Etikan, S. A. Musa, and R. S. Alkassim, "Comparison of Convenience Sampling and Purposive Sampling," American Journal of Theoretical and Applied Statistics, vol. 5, no. 1, pp. 1–4, 2016. DOI: 10.11648/j.ajtas.20160501.11

[8] T. M. Mitchell, Machine Learning. McGraw-Hill, 1997.

[9] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016. Available: http://www.deeplearningbook.org

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, pp. 436–444, 2015. DOI: 10.1038/nature14539

[11] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," Neural Computation, vol. 18, no. 7, pp. 1527–1554, 2006.

[12] D. Jurafsky and J. H. Martin, Speech and Language Processing, 3rd ed. (draft), 2023. Available: https://web.stanford.edu/~jurafsky/slp3/

[13] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in Proc. NIPS, pp. 3111–3119, 2013.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in Proc. NIPS, pp. 5998–6008, 2017.

[15] W. X. Zhao et al., "A survey of large language models," arXiv:2303.18223, 2023.

[16] T. B. Brown et al., "Language models are few-shot learners," in Proc. NeurIPS, vol. 33, pp. 1877–1901, 2020.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL-HLT, pp. 4171–4186, 2019.

[18] H. K. Dam, "Artificial Intelligence for Software Engineering," XRDS: Crossroads, the ACM Magazine for Students, vol. 25, no. 3, pp. 34–37, 2019. DOI: 10.1145/3313117

[19] T. Xie, "The Synergy of Human and Artificial Intelligence in Software Engineering," in Proc. 2013 RAISE Workshop, pp. 4–6, 2013. DOI: 10.1109/RAISE.2013.6615197

[20] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large Language Models for Software Engineering: A Systematic Literature Review," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 8, art. 220, 2024. DOI: 10.1145/3695988

[21] A. Sergeyuk, S. Titov, and M. Izadi, "In-IDE Human-AI Experience in the Era of Large Language Models; A Literature Review," in Proc. 1st IEEE/ACM Workshop on AI Foundation Models and Software Engineering (FORGE '24), 2024. DOI: 10.1145/3650105.3652301

[22] A. E. Hassan, G. A. Oliva, D. Lin, B. Chen, and Z. M. Jiang, "Rethinking Software Engineering in the Foundation Model Era: From Task-Driven AI Copilots to Goal-Driven AI Pair Programmers," arXiv:2404.10225, 2024.

[23] A. Sarkar and I. Drosos, "Vibe Coding: Programming Through Conversation with Artificial Intelligence," arXiv:2506.23253, 2025.

[24] R. Sapkota, K. I. Roumeliotis, and M. Karkee, "Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI," arXiv:2505.19443, 2025.

[25] C. Meske, T. Hermanns, E. von der Weiden, K.-U. Loser, and T. Berger, "Vibe Coding as a Reconfiguration of Intent Mediation in Software Development," arXiv:2507.21928, 2025.

[26] Y. Ge et al., "A Survey of Vibe Coding with Large Language Models," arXiv preprint arXiv:2510.12399, 2025.

[27] T. Li, T. Maheshwari, and A. Voelker, "User-Centered Design with AI in the Loop: A Case Study of Rapid UI Prototyping with Vibe Coding," arXiv:2507.21012, 2025.

[28] A. E. Hassan, G. A. Oliva, D. Lin, B. Chen, and Z. M. Jiang, "Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap," arXiv:2410.06107, 2024.

[29] C. Qian, X. Cong, W. Liu, C. Yang, W. Chen, Y. Su, Y. Dang, J. Li, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative Agents for Software Development," in Proc. 62nd Annu. Meeting of the Assoc. for Comput. Linguistics (ACL), pp. 15174–15186, 2024. DOI: 10.18653/v1/2024.acl-long.810

[30] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, and C. Wu, "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework," in Proc. Int. Conf. Learn. Representations (ICLR), 2024.

[31] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," arXiv preprint arXiv:2308.08155, 2023.

[32] K. Rokis and M. Kirikova, "Challenges of Low-Code/No-Code Software Development: A Literature Review," in Proc. BIR Workshops, pp. 3–14, 2022.

[33] G. Kacheru, N. Arthan, and R. Bajjuru, "AI for Low-Code and No-Code Development: Making Non-Developers Developers in 2024," Formosa Journal of Multidisciplinary Research, vol. 4, no. 1, pp. 141–150, 2025. DOI: 10.55927/fjmr.v4i1.13369

[34] K. Wright, Y. Antonucci, L. Anderson, and A. Townsend, "Engaging Business Students with 'Low-Code' Model Driven Development," in Proc. Annual Hawaii Intl. Conf. System Sciences, 2023. DOI: 10.24251/HICSS.2023.567

[35] J. R. Gottam, "Beyond Manual Testing: Hyperautomation's Transformative Impact on Software Quality Engineering," Journal of Computer Science and Technology Studies, 2025.

[36] N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler, "The SPACE of Developer Productivity," Queue, vol. 19, no. 1, pp. 20–48, 2021. DOI: 10.1145/3454122.3454124

[37] K. K. B. Ng, L. Fauzi, L. Leow, and J. Ng, "Harnessing the Potential of Gen-AI Coding Assistants in Public Sector Software Development," arXiv:2409.17434, 2024.

[38] M. Barenkamp, J. Rebstadt, and O. Thomas, "Applying AI in Software Engineering," Procedia Computer Science, vol. 170, pp. 1023–1028, 2020. DOI: 10.1016/j.procs.2020.03.112

[39] M. Weyssow, A. Kamanda, X. Zhou, and H. A. Sahraoui, "CodeUltraFeedback: An LLM-as-a-Judge Dataset for Aligning Large Language Models to Coding Preferences," ACM Trans. Software Engineering and Methodology, 2024.

[40] B. Zeng, Q. Zhang, C. Zhou, G. Go, Y. Jiang, and H. Shi, "Inducing Vulnerable Code Generation in LLM Coding Assistants," arXiv:2504.15867, 2025.

[41] M. S. Sree, C. K. K. Reddy, K. Vaishnavi, and V. Harika, "AI-Powered Software Engineering for Cloud-Native Environments," Advances in Computational Intelligence and Robotics, pp. 57–86, 2025. DOI: 10.4018/979-8-3693-9356-7.ch003

[42] A. Ahmad, M. Waseem, P. Liang, M. Fehmideh, M. S. Aktar, and T. Mikkonen, "Towards Human-Bot Collaborative Software Architecting with ChatGPT," arXiv:2302.14600, 2023.

[43] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," Biometrics, vol. 33, no. 1, pp. 159–174, 1977. DOI: 10.2307/2529310

[44] J. Cohen, "A coefficient of agreement for nominal scales," Educational and Psychological Measurement, vol. 20, no. 1, pp. 37–46, 1960. DOI: 10.1177/001316446002000104

[45] J. L. Fleiss, Statistical Methods for Rates and Proportions, 2nd ed. New York: Wiley, 1981.

[46] Q. Zhang, "The Role of Artificial Intelligence in Modern Software Engineering," Applied and Computational Engineering, vol. 97, no. 1, pp. 18–23, 2024. DOI: 10.54254/2755-2721/97/20241339

[47] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language

Models," CHI '22 Extended Abstracts on Human Factors in Computing Systems, New Orleans, LA, USA, 2022.

[48] J. T. Liang, C. Yang, and B. A. Myers, "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges," Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), Lisbon, Portugal, 2024.

[49] S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How Programmers Interact with Code-Generating Models," Proceedings of the ACM on Programming Languages, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[50] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 5, 2024.

[51] M. Barenkamp, J. Rebstadt, and O. Thomas, "Applications of AI in Classical Software Engineering," AI Perspectives, vol. 2, no. 1, 2020. DOI: 10.1186/s42467-020-00005-4

[52] J. C. Goncalves and M. de A. Maia, "An Empirical Study on the Effectiveness of Iterative LLM-Based Improvements for Static Analysis Issues," in Proc. XXXIX SBES, 2025.

[53] Z. Khaliq, S. U. Farooq, and D. A. Khan, "AI in Software Testing: Impact, Problems, Challenges and Prospect," arXiv:2201.05371, 2022.

[54] Md. A. Hayat, S. Islam, and Md. F. Hossain, "The Evolving Role of Artificial Intelligence in Software Testing," Intl. Journal for Multidisciplinary Research, vol. 6, no. 2, 2024. DOI: 10.36948/ijfmr.2024.v06i02.14783

[55] U. K. R. Veeramreddygari, "Generative AI for Software Engineering: LLM-Driven Code Generation with Safety and Trust Assessment," Intl. Journal of Scientific Research in Computer Science, Engineering and Information Technology, 2023.

[56] C. K. Tantithamthavorn, J. Jiarpakdee, and J. C. Grundy, "Explainable AI for Software Engineering," in Proc. 2021 IEEE/ACM ASE, pp. 1–2, 2021.

Appendices

Appendix A: Survey Instrument

Section 1, Background

- Years of professional software development experience
- Primary languages and frameworks used
- Frequency of AI coding tool use (never / rarely / sometimes / often / almost always)
- Which AI tools do you use? (open text)
- Familiarity with low-code or no-code platforms (none / limited / moderate / extensive)

Section 2, Perception Items (1 = strongly disagree, 5 = strongly agree)

- AI tools help me complete coding tasks faster.
- AI tools save me time overall.
- I feel comfortable incorporating AI tools into my workflow.
- AI sometimes produces code that is more complicated than the task required.
- I usually need to review or modify AI-generated code before integrating it.
- AI-generated code is sometimes confusing or hard to follow.
- Low-code tools make routine development tasks noticeably easier.
- Combining AI assistance with low-code tools improves my overall workflow.
- I trust AI-generated code mainly for small or non-critical tasks.

Section 3, Open Ended

- Describe your best or worst experience using AI tools during development. What happened and what did you take from it?