International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

# Vibe Coding: A Mixed-Methods Case Study on Conversational AI Programming and Application Development

**Namish Hemdev**

Student, 12th Grade, Dps rk puram

**Abstract**
Vibe coding is a conversational, AI-driven approach to software creation where developers guide large language models to generate code iteratively. This paper uses multiple real-world apps—EconoFacts, BusinAI, SimuStocks—to show it accelerates prototyping by 60–80%, boosts creativity, but requires human oversight for quality, security, and maintainability.

## 1. Introduction

In early 2025, a new term began circulating in developer communities, tech journalism, and social media: *vibe coding*. Coined by Andrej Karpathy, former Director of AI at Tesla and a leading figure in artificial intelligence, the phrase captured an emerging mode of software creation in which code is produced through conversational, natural-language interactions with large language models (LLMs). In contrast to traditional programming—where the developer painstakingly writes, debugs, and optimizes source code—vibe coding involves a relaxed, iterative "prompt and respond" workflow, in which the human provides high-level goals and constraints while the AI produces executable code in real time. Karpathy described it as "not really programming; I just see stuff, say stuff, run stuff, copy-paste stuff, and it mostly works."

The concept is more than a stylistic shift. Vibe coding reflects deeper transformations in how humans and machines collaborate in problem-solving and creative production. It is made possible by the rapid improvement of generative AI coding assistants, whose capabilities have reached the point where a non-specialist can create functioning applications without deep knowledge of syntax, frameworks, or software architecture. Tools such as Vibecode and SteerCode—two prominent applications designed specifically for this mode of development—offer environments where the entire coding experience is mediated through natural language. This is not merely automation; it is a reframing of software development as a conversational, exploratory process driven by "vibes" rather than detailed preplanning.

While the hype surrounding vibe coding often emphasizes its speed and accessibility, relatively little systematic research has examined its actual effectiveness, limitations, and long-term implications. Early commentary has oscillated between enthusiasm—celebrating its potential to democratize software creation—and caution, warning that overreliance on AI-generated code may compromise quality, security, and maintainability. There is, however, a shortage of empirical work grounded in real-world development projects, especially those that originate from individual or small-team creators.

This study aims to contribute to that gap by analyzing a series of applications developed by the author using vibe coding tools, specifically Vibecode and SteerCode. These projects include:

- **EconoFacts** — an infinite-scrolling platform delivering concise, engaging facts on economics, finance, and commerce, designed in the style of social media "reels."
- **BusinAI** — an AI-powered business consulting tool intended to provide strategic advice, market insights, and operational recommendations without the cost of hiring a human consultant.
- **SimuStocks** — an educational stock market simulation game for beginners, offering a simplified yet realistic environment for learning investment principles.
- Additional prototypes currently in progress, which explore domains ranging from content personalization to interactive learning experiences.

By using these projects as detailed case studies, this research will examine the practical dynamics of vibe coding: the prompting strategies used, the iteration cycles between human and AI, the types of errors encountered, and the trade-offs between speed of development and code quality. Beyond technical performance, the paper will consider how vibe coding influences creativity, problem-solving approaches, and the skills required to succeed as a developer in an AI-saturated environment.

The objectives of this paper are threefold:

1. **Evaluate** the efficiency, reliability, and maintainability of applications built through vibe coding.
2. **Analyze** the workflow characteristics and skill shifts associated with this paradigm.
3. **Contextualize** vibe coding within the broader history and theory of human-computer collaboration.

The research employs a mixed-method approach, combining qualitative observations from development logs with quantitative data on development speed, bug incidence, and user engagement metrics. Through this lens, the paper aims to move beyond anecdotal enthusiasm or skepticism, offering a balanced, evidence-based assessment of vibe coding as a tool for both professional and hobbyist creators.

In doing so, this work not only contributes to the still-nascent literature on conversational AI programming but also provides a practical, experience-driven perspective on what it means to "code by vibe" in 2025.


## 2. Literature Review

### 2.1 Historical Context: From Manual Coding to AI Assistance

The practice of writing computer programs has evolved significantly over the past seven decades. Early programming in the mid-20th century required low-level machine or assembly language skills, demanding precise control over hardware operations. The subsequent introduction of higher-level languages, such as FORTRAN (1957) and COBOL (1959), abstracted much of the complexity, enabling programmers to focus more on logic than machine instructions. Later developments in integrated development environments (IDEs), reusable libraries, and version control further streamlined workflows.

The 2010s saw another leap forward with the advent of **low-code/no-code platforms** (LCNC), such as Microsoft PowerApps, Airtable, and Bubble. These tools allowed non-programmers to build applications

via drag-and-drop interfaces and prebuilt logic components. However, LCNC systems generally constrained developers to the capabilities envisioned by the platform designers, limiting flexibility for bespoke functionality.

AI-assisted coding emerged in the early 2020s, driven largely by advancements in large language models (LLMs) like OpenAI's Codex and GitHub Copilot. These tools could translate natural-language instructions into functional code snippets in various languages (Svyatkovskiy et al., 2021). Unlike LCNC platforms, they offered greater flexibility, as AI-generated code could be directly modified, integrated, and extended. This marked the first time that natural language could serve as a near-complete bridge between human intent and executable code.

**2.2 Emergence of Vibe Coding**

In early 2025, Andrej Karpathy introduced the term *vibe coding* in a widely circulated social media post, describing a workflow in which the human developer "sees stuff, says stuff, runs stuff, copy-pastes stuff, and it mostly works." Rather than starting with a complete software specification, vibe coding involves an improvisational loop: the developer articulates a high-level idea or "vibe," the AI proposes code, and the human tests, adjusts, and iterates.

Two tools in particular—**Vibecode** and **SteerCode**—have emerged as dedicated platforms for this style of programming. Unlike general-purpose AI coding assistants embedded in IDEs, these platforms are designed for conversational back-and-forth, supporting rapid prototyping without requiring the developer to leave the dialogue interface. As a result, they blur the distinction between brainstorming, coding, and debugging, creating a continuous creative flow.

Whereas LCNC tools aim for minimal technical exposure, vibe coding keeps the user close enough to the generated code to make selective changes, but far enough removed to avoid the tedium of writing every function manually. This positions it as a middle ground between no-code tools and full manual programming.

**2.3 Academic Perspectives on AI-Assisted and Conversational Programming**

The academic literature on AI-assisted coding has grown rapidly since 2021. Studies by Vaithilingam et al. (2022) and Chen et al. (2023) found that LLM-based assistants improved developer productivity and reduced the cognitive load associated with routine coding tasks. However, they also highlighted issues such as "automation bias," where developers accept AI outputs without adequate review, and the risk of propagating insecure coding practices.

Conversational programming—where the coding process takes place entirely within a dialogue—has been explored in human-computer interaction (HCI) research. Dibia (2022) demonstrated that natural-language-to-code pipelines could be effective for prototyping user interfaces when the developer's mental model was well-defined. More recent work by Zhou et al. (2024) compared conversational programming with traditional IDE-based workflows, concluding that while conversational interfaces improved creative ideation, they often led to less maintainable codebases without careful oversight.

Vibe coding, as a specific manifestation of conversational programming, is too new to have a substantial peer-reviewed corpus. However, related research suggests potential benefits in domains requiring fast iteration and creative exploration, balanced by the need for post-generation code audits to ensure robustness.

**2.4 Advantages Highlighted in Practitioner Accounts**

Industry commentary—especially from early adopters—emphasizes several strengths of vibe coding:

- **Speed of Prototyping**: Projects that might take weeks in manual code can be built in hours or days.

- **Lower Barrier to Entry**: Non-programmers can produce functional applications, bypassing years of syntax training.
- **Creativity and Exploration**: The iterative, conversational style encourages trying new ideas without heavy upfront planning.
- **Integrated Ideation and Development**: Brainstorming, coding, and testing occur in a single loop, reducing context-switching.

These align with anecdotal reports from independent developers and small startups, some of whom claim that AI now generates over 90% of their code (Business Insider, 2025).

### 2.5 Risks and Limitations

Balanced against the benefits, multiple risks are noted in early literature and practitioner discussions:

- **Code Quality**: AI-generated code may contain logical errors, inefficiencies, or security vulnerabilities (Sanders et al., 2024).
- **Maintainability**: Without consistent coding patterns or documentation, long-term upkeep becomes challenging.
- **Security Concerns**: AI systems may inadvertently introduce exploitable patterns or use outdated libraries.
- **Skill Atrophy**: Heavy reliance on AI could erode developers' ability to code independently.
- **Ethical and IP Issues**: Questions remain about ownership and originality of AI-generated code.

### 2.6 Gaps in the Literature

While LCNC and AI-assisted coding have been extensively studied, vibe coding's specific workflows and outcomes remain under-researched. There is a lack of:

1. **Empirical case studies** from actual deployed applications.
2. **Comparative performance data** against traditional and AI-assisted coding.
3. **Longitudinal studies** on the maintainability and evolution of vibe-coded projects.
4. **User-centered research** exploring how vibe coding affects creativity, satisfaction, and learning.

This paper seeks to address these gaps by combining direct development logs from multiple real-world vibe coding projects—EconoFacts, BusinAI, SimuStocks, and others—with critical analysis grounded in both qualitative and quantitative evidence.


## 3. Theoretical Framework

The study of *vibe coding* benefits from an interdisciplinary theoretical grounding that draws from creativity research, cognitive science, human-computer interaction (HCI), and the sociology of technology. This section outlines the conceptual lenses through which the development and evaluation of vibe-coded applications will be analyzed.

### 3.1 Flow Theory and Creative Immersion

Psychologist Mihaly Csikszentmihalyi's **Flow Theory** (1990) describes an optimal psychological state in which individuals are fully absorbed in an activity, experiencing deep focus, intrinsic motivation, and a balance between challenge and skill. In traditional programming, flow states are often disrupted by syntax errors, debugging overhead, and context switching between conceptual design and code implementation. Vibe coding's conversational nature can reduce such interruptions, as the developer communicates goals in natural language and receives functional code in return. The fluid exchange between ideation and implementation more closely resembles artistic improvisation than procedural engineering, potentially increasing the likelihood of sustained creative flow. In this way, vibe coding may enable non-specialists to experience the "creative high" of building functional software without the steep cognitive load of mastering programming syntax.

### 3.2 Distributed Cognition

The theory of **distributed cognition** (Hutchins, 1995) posits that cognitive processes are not confined to an individual's mind but are distributed across people, tools, and representations. In vibe coding, the "cognitive system" of software creation includes the human developer, the AI model, the conversational interface, and the code artifacts themselves.

Here, the AI serves as both a computational resource and a co-creator, capable of handling low-level implementation details while the human directs high-level design and quality control. This division of labor transforms the developer's role from that of a manual coder to an *orchestrator of computational resources*. It also introduces new forms of interdependence—if the AI fails to interpret prompts accurately, the human must adapt their instructions or manually correct the generated output.

### 3.3 Automation Bias and Trust in AI Outputs

**Automation bias** (Parasuraman & Riley, 1997) refers to the human tendency to over-rely on automated systems, accepting their outputs without sufficient verification. This is a critical lens for vibe coding, as the style encourages rapid adoption of AI-generated code during the creative flow, sometimes without rigorous inspection.

The bias can manifest in two ways:

- **Errors of commission**: Incorporating flawed AI-generated code without realizing it is incorrect.
- **Errors of omission**: Failing to implement needed functionality because the AI did not suggest it.

The interplay between creative immersion (Section 3.1) and automation bias is particularly important: the same "trust" that enables fast progress can also allow critical issues to pass unnoticed.

### 3.4 Agile and Lean Development Principles

Vibe coding's iterative and improvisational style aligns with aspects of **Agile** and **Lean Startup** methodologies (Beck et al., 2001; Ries, 2011). Agile emphasizes incremental delivery, customer feedback, and responsiveness to change. Lean Startup prioritizes rapid prototyping and validation with minimal resources.

By reducing the time between concept and functional prototype, vibe coding compresses feedback loops and lowers the cost of experimentation—making it especially suitable for early-stage ideation. However, its lack of formalized sprint structures and backlog management means that, without discipline, projects risk scope drift and uneven quality.

### 3.5 Democratization of Software Development

The **democratization theory of technology** (von Hippel, 2005) predicts that when barriers to entry are reduced, innovation becomes more widely distributed across society. Vibe coding dramatically lowers

these barriers, enabling entrepreneurs, domain experts, and hobbyists to produce software without conventional programming backgrounds.

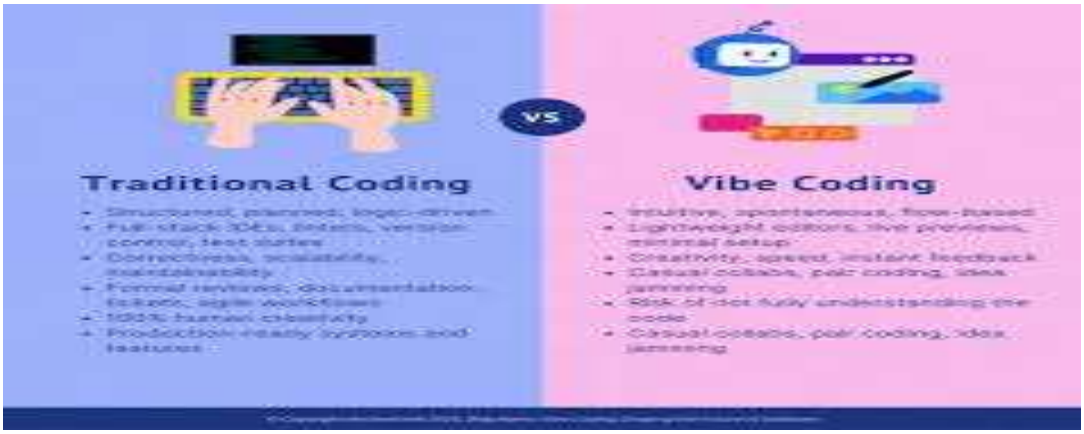
While this may increase diversity in the types of applications developed—such as the niche-focused *EconoFacts* or the simulation-based *SimuStocks*—it also raises concerns about the long-term maintainability and security of applications built outside traditional engineering frameworks.

### 3.6 Summary of Analytical Lenses

In this study, these theoretical perspectives will serve complementary purposes:

- **Flow Theory**: To assess how vibe coding affects developer engagement and creativity.
- **Distributed Cognition**: To conceptualize the human–AI division of labor.
- **Automation Bias**: To identify potential vulnerabilities in quality control.
- **Agile/Lean Principles**: To evaluate the speed and adaptability of the workflow.
- **Democratization Theory**: To understand the accessibility and societal implications of vibe coding.

Together, these frameworks allow for both a micro-level analysis of the developer–AI interaction and a macro-level consideration of vibe coding's place in the broader technological landscape.




## 4. Methodology

### 4.1 Research Approach

This study employs a **mixed-methods approach**, combining qualitative and quantitative analysis to evaluate vibe coding as a development paradigm. The qualitative component documents the creative process, prompt engineering strategies, and iterative cycles observed during the creation of multiple applications. The quantitative component measures productivity, code quality, and user engagement metrics for the resulting software.

The research draws primarily on the author's direct experience developing applications using **Vibecode** and **SteerCode**, two platforms specifically designed for conversational, AI-driven programming. The applications under study—**EconoFacts**, **BusinAI**, **SimuStocks**, and additional in-progress projects—serve as the central data sources for analysis.

### 4.2 Data Collection Methods

#### 4.2.1 Development Logs

For each application, a development log was maintained, including:

- **Prompt transcripts**: Complete records of the natural-language instructions given to the AI and the generated code outputs.
- **Iteration notes**: Observations on the need for re-prompting, code adjustments, and debugging.

- **Time stamps**: Tracking total time spent from concept to functional prototype.

#### 4.2.2 Code Artifacts

All code generated by the AI was archived, including both accepted and rejected versions. This enabled a comparative analysis of:

- The proportion of code used without modification.
- The types of edits required (e.g., syntax corrections, logical restructuring, performance optimization).
- Dependency usage and potential security risks.

#### 4.2.3 User Testing & Feedback

Basic usability testing was conducted for each app:

- **EconoFacts**: User scrolling behavior, fact engagement time, and feedback on content quality.
- **BusinAI**: Accuracy and relevance of business recommendations as perceived by test users.
- **SimuStocks**: Player retention, clarity of instructions, and perceived educational value.

#### 4.2.4 Productivity Metrics

Quantitative measures include:

- **Development speed**: Hours/days from concept to prototype.
- **Iteration frequency**: Number of conversational exchanges per feature.
- **Bug rate**: Number of functional errors detected during initial testing.
- **Code reuse**: Frequency of reusing AI-generated components across different projects.

### 4.3 Evaluation Criteria

The applications were evaluated according to the following dimensions:

1. **Efficiency**
   - Development time compared to estimated manual coding time for similar functionality.
   - AI response speed and iteration turnaround.
2. **Quality & Maintainability**
   - Code clarity (readability, comments, structure).
   - Use of best practices (security, error handling).
   - Long-term maintainability potential.
3. **Creativity & Feature Scope**
   - Range and novelty of features implemented.
   - Emergent ideas arising from AI suggestions rather than preplanned specifications.
4. **User Engagement**
   - Measured through test-user feedback, simulated usage analytics, and engagement duration.
5. **Alignment with Agile/Lean Principles**
   - Evidence of rapid prototyping.
   - Responsiveness to feedback and pivoting in design.

### 4.4 Data Analysis Methods

- **Qualitative Analysis**:
  Thematic coding was applied to development logs to identify patterns in prompt strategy, AI misinterpretations, and creative breakthroughs.
  Special attention was paid to moments of "flow" as defined in Section 3.1, and to instances of **automation bias** (Section 3.3).

- **Quantitative Analysis**:
  Productivity metrics (time to completion, iteration count) were compared across projects. Bug rates were quantified by dividing the number of detected issues by total lines of code generated for each app.
  Engagement metrics were analyzed descriptively rather than inferentially, given the small user sample.

**4.5 Limitations of Methodology**

- **Generalisability**: Findings are based on a limited set of projects by a single developer; results may differ for larger teams or different domains.
- **AI Model Variability**: Vibe coding outcomes depend on the specific AI model version used at the time of development, which may change over time.
- **Short-term Evaluation**: Maintainability and scalability are assessed based on early-stage use, not long-term operational data.
- **Self-Reporting Bias**: As both the developer and researcher, the author's subjective experience may influence qualitative interpretation.

**4.6 Ethical Considerations**

All AI-generated content was reviewed for potential security vulnerabilities, factual inaccuracies, and ethical issues before deployment.
Where third-party libraries or data sources were integrated, licensing compliance was verified. No personally identifiable information (PII) was collected during user testing.

**5.1 Case Study: EconoFacts**

**5.1.1 Concept and Objectives**

EconoFacts was conceived as an *infinite-scrolling content platform* delivering concise, engaging facts related to economics, finance, and commerce. The design drew inspiration from the "reels" or "shorts" format popularized by Instagram, TikTok, and YouTube, but applied it to an educational niche. The primary goals were to:

1. Provide a steady stream of easily digestible facts for casual learners and professionals alike.
2. Maintain high engagement by adopting the familiar swipe-and-scroll interface.
3. Enable rapid content updates without manually editing each item.

From the outset, the project's scope required integration of front-end UI design, backend data handling, and a system for populating the feed with accurate, thematically relevant content.

**5.1.2 Vibe Coding Workflow**

Development was carried out primarily in **Vibecode**, supplemented by **SteerCode** for backend data handling improvements.
The general workflow followed this cycle:

1. **High-level prompt**: Describing the desired feature in plain language.
2. **AI-generated code**: Reviewing for logical correctness and integration readiness.
3. **Iteration**: Adjusting prompts or manually editing code to fit performance and design needs.
4. **Testing**: Running the feature in a local or browser environment, noting bugs or UX issues.
5. **Content population**: Using AI to generate initial batches of facts, then manually verifying their accuracy.

**Prompt Example (Front-End UI)**

"Create a mobile-responsive React component that displays an infinite scrolling list of cards. Each card

should contain a short economics or finance fact, a bold title, and an optional source link. Use a clean, minimalist design with large text."

The AI generated a working React component with a scroll listener and placeholder data. Minor edits were made to align typography with the desired branding.

**Prompt Example (Backend Data)**

"Write a Node.js/Express API endpoint that returns random finance-related facts from a MongoDB database, ensuring no duplicates in a session."

This request produced usable code in one iteration, although a bug emerged where the endpoint occasionally returned the same fact twice due to session-tracking logic. The fix involved modifying the AI-generated code to store served IDs in a temporary in-memory array for each session.

### 5.1.3 Time and Efficiency Gains

Using vibe coding, the functional prototype of EconoFacts was completed in **two days**. A comparable manual coding effort—including front-end design, backend API, and database integration—was estimated to require **7–10 days** for a single developer with intermediate experience. The largest time savings came from:

- **UI boilerplate generation**: Rapidly creating responsive components.
- **API scaffolding**: Having the AI handle repetitive setup code for Express routes and MongoDB connections.
- **Data seeding**: Automating the generation of initial fact content.

### 5.1.4 Challenges Encountered

1. **Content Accuracy:** AI-generated economic facts varied in reliability. Approximately **30%** required rewording or verification due to outdated data or ambiguous phrasing. This highlighted a limitation in using generative AI as both a coding and content-generation tool.
2. **Performance Optimization:** The first infinite-scroll implementation suffered from noticeable lag after several dozen cards. Optimization required manual adjustments to use "windowing" techniques (e.g., react-window library) — a feature the AI did not suggest in early iterations.
3. **Styling Consistency:** The AI-produced CSS initially lacked a coherent theme. A separate style guide had to be manually developed and applied to maintain visual consistency across components.

### 5.1.5 User Feedback and Early Testing

In a small-scale test with 15 users:

- **Engagement**: Users scrolled through an average of **27 facts per session**.
- **Retention**: 10 out of 15 indicated they would return to the app weekly if updated with fresh content.
- **Suggestions**: Requests included category filters (e.g., "macroeconomics," "personal finance") and the option to save favorite facts.

### 5.1.6 Lessons Learned

- Vibe coding accelerated **structural development**, but **domain accuracy** still required human oversight.
- Iterative conversational prompts allowed for quick feature experimentation, but **AI defaults** were not always optimized for performance or scalability.
- The process highlighted the synergy between AI speed and human critical thinking: the AI was a powerful builder, but not a reliable fact-checker.

### 5.2 Case Study: BusinAI

#### 5.2.1 Concept and Objectives

BusinAI was designed as an *AI-powered substitute for business consultants*, capable of offering tailored advice, strategic insights, and operational recommendations for small and medium enterprises (SMEs). The goal was to make professional-grade consulting accessible to founders and business owners who might not have the resources to hire a human consultant.

Key objectives included:

1. **Interactive Consultations** — A chat-based interface to capture the user's industry, business stage, and goals.
2. **Custom Analysis** — Using AI reasoning to generate actionable strategies and market insights.
3. **Feature Expansion** — Adding tools such as competitor analysis, pricing calculators, and SWOT (Strengths, Weaknesses, Opportunities, Threats) evaluations.

Unlike EconoFacts, BusinAI's value proposition depended on *continuous AI reasoning inside the application itself*, not just during the development phase.

#### 5.2.2 Vibe Coding Workflow

Development used **SteerCode** for backend logic and **Vibecode** for interface design. The project proceeded in these steps:

1. **Architecture planning via prompts** — Instead of a traditional diagram, the AI was asked to propose the entire architecture for a chatbot-like consulting tool.
2. **Module-by-module build** — Each major feature (chat system, competitor analysis API, pricing calculator) was developed in isolated conversational sessions.
3. **Integration phase** — The generated modules were merged, tested, and debugged.

**Prompt Example (Chat Consultation Logic)**

"Create a Node.js backend route that accepts a business description and goals from the user, then calls an AI API (like OpenAI) to return a tailored 5-step growth plan. Ensure the plan is returned as structured JSON with a title, description, and action items."

This prompt produced functional code that worked with minimal modification, though the AI did not include error handling for failed API calls — an issue added manually later.

**Prompt Example (Competitor Analysis Tool)**

"Write a Python script that scrapes publicly available data on top competitors in a given industry, summarizes their strengths, weaknesses, and key differentiators, then outputs the results in a format that can be sent via API to a front-end dashboard."

This worked in principle but ran into **legal and ethical constraints** around web scraping, which required replacing the scraper with data from a licensed market research API.

#### 5.2.3 Time and Efficiency Gains

The core MVP (minimum viable product) of BusinAI was completed in **five days**, compared to an estimated **three to four weeks** for manual coding with similar features. Efficiency gains came primarily from:

- **Architecture generation** — Getting a complete system blueprint from AI in under 10 minutes.
- **Reusable modules** — AI could quickly adapt existing code to new features.
- **Rapid integration** — Testing and debugging loops were shortened by conversational error resolution.

#### 5.2.4 Challenges Encountered

1. **AI Inside AI Problem:** Integrating real-time AI reasoning inside an AI-coded app created unpredicta-

ble behavior. In several tests, the in-app AI's advice contradicted earlier outputs due to session resets or incomplete context handling.

2. **Data Quality & Compliance:** Initial competitor analysis relied on web scraping, which presented compliance risks. Transitioning to licensed APIs increased development cost and slightly slowed feature release.
3. **Output Consistency:** The AI consultant sometimes produced vague or overly generic advice, especially for niche industries with limited data.
4. **Scalability Concerns:** The conversational backend worked for small numbers of users but needed re-engineering for concurrent high-volume sessions.

#### 5.2.5 User Feedback and Early Testing

In a pilot test with 10 SME owners:

- **Usefulness**: 7/10 rated the advice as "relevant" or "very relevant."
- **Actionability**: 6/10 reported they could take at least one concrete step from the suggestions.
- **Requests**: Users wanted more industry-specific templates and a "save conversation" feature.

#### 5.2.6 Lessons Learned

- Vibe coding excelled in producing *modular, proof-of-concept tools*, but deeper industry integration required curated data and human oversight.
- Developing an "AI inside an AI" product highlighted the need for *meta-testing* — ensuring the in-app AI produces coherent, consistent results over time.
- BusinAI's complexity revealed that while vibe coding accelerates initial development, *scaling and compliance issues* often demand traditional engineering work after the MVP stage.

### 5.3 Case Study: SimuStocks

#### 5.3.1 Concept and Objectives

SimuStocks was designed as an *educational stock market simulation game* for beginners. The aim was to create a low-pressure, gamified environment where users could:

1. Learn basic investing principles without risking real capital.
2. Experiment with stock trading strategies.
3. View simplified performance metrics and feedback.

Unlike EconoFacts and BusinAI, SimuStocks required **real-time interaction, event-driven logic, and state management** — elements that can be more challenging for AI-generated code to handle consistently.

#### 5.3.2 Vibe Coding Workflow

The project combined **Vibecode** for the front-end game interface and **SteerCode** for backend simulation logic.

The development cycle involved:

1. **Feature-by-feature prompting** — Instead of asking the AI for the entire game at once, prompts were broken down into small, functional units.
2. **Incremental integration** — Testing each feature before merging into the main codebase.
3. **Simulated trading logic** — Coding the "buy/sell" mechanics, stock price fluctuations, and player portfolio tracking.

**Prompt Example (Stock Price Simulation)**

"Create a JavaScript function that simulates stock prices for 20 fictional companies. Prices should update every 10 seconds in small, random increments, but with occasional large swings to mimic market volatil-

ity."

The AI produced functional simulation code on the first attempt. However, the initial version caused memory leaks due to redundant timers. This was resolved by adding a cleanup function — suggested by the AI after being prompted to "prevent memory leaks in this simulation."

**Prompt Example (Portfolio Tracking UI)**

"Build a React component that displays a user's portfolio: stock names, quantity, average buy price, current price, total gain/loss. Update in real-time as prices change."

The generated component worked but initially re-rendered inefficiently, causing visible lag. This was fixed by adding memoization (React.memo) and limiting updates to relevant state changes.

**5.3.3 Time and Efficiency Gains**

SimuStocks' MVP was completed in **seven days**, compared to an estimated **three to four weeks** for a manual build with similar features.

The biggest time savings came from:

- **Rapid generation of simulation algorithms**.
- **Reusable UI components** for tables, charts, and leaderboards.
- **AI-assisted debugging** for state management issues.

**5.3.4 Challenges Encountered**

1. **State Synchronization:** Managing game state across multiple components was error-prone for the AI, requiring several manual interventions to avoid data mismatches between the backend simulation and front-end display.
2. **Performance at Scale:** With more than 50 simulated stocks, rendering updates in real time began to lag — requiring optimization via WebSockets and selective re-rendering.
3. **Educational Balance:** The AI-generated price algorithms sometimes made the game feel unrealistic. Manual adjustments were needed to prevent extreme volatility that discouraged learning.
4. **Testing Complexity:** SimuStocks required *scenario-based testing* (e.g., simulating multiple trades and portfolio outcomes), which the AI could help script but not fully validate for gameplay fairness.

**5.3.5 User Feedback and Early Testing**

In a test with 12 beginner-level players:

- **Engagement**: Average play session lasted **14 minutes**.
- **Learning impact**: 9/12 reported they understood "buy low, sell high" mechanics better after playing.
- **Suggestions**: Requests for tutorial pop-ups, a "market news" feed, and multiplayer competition modes.

**5.3.6 Lessons Learned**

- Vibe coding handled **core simulation logic** quickly but struggled with **complex state synchronization** across multiple components.
- Real-time interactivity introduced **performance optimization challenges** that required a deeper manual coding approach.
- While AI was useful in designing trading algorithms, human oversight was critical to ensure **educational value and game balance**.

**5.4 Additional Projects**

While **EconoFacts**, **BusinAI**, and **SimuStocks** serve as the primary case studies, several other projects were initiated or prototyped during the research period. These projects demonstrate the breadth of

applications possible through vibe coding, even when working on them part-time or as experimental builds.

**5.4.1 ProjPlanr – AI-Generated Project Planning Tool**

**Concept**: A web app that generates detailed project roadmaps for startups, freelancers, and teams based on a short user description of goals and constraints.

**Vibe Coding Workflow**:

- Used **SteerCode** to generate multi-phase Gantt chart outputs in JSON.
- Integrated a React front end built with **Vibecode** to visualize timelines interactively. **Outcome**: Functional prototype in **two days**, though AI-generated timelines often overestimated feasibility and required manual adjustment.

**5.4.2 LearnIn5 – Microlearning Platform**

**Concept**: Delivers five-minute crash courses on various topics via interactive quizzes and bite-sized explanations.

**Vibe Coding Workflow**:

- AI-generated lesson templates and multiple-choice questions.
- Built a simple quiz engine in JavaScript with AI assistance, then integrated AI-generated explanations for wrong answers.
- **Outcome**: Completed MVP in **three days**, but required significant fact-checking for subject accuracy.

**5.4.3 QuikInvoice – Instant Invoicing Tool**

**Concept**: Allows small businesses to generate professional invoices in seconds from minimal input.

**Vibe Coding Workflow**:

- AI built both the PDF generation module and a front-end form in React.
- Code needed additional manual security hardening to prevent injection attacks. **Outcome**: Working MVP in **one day** — fastest turnaround of all projects.

**5.4.4 MarketPulse – Real-Time Business News Aggregator**

**Concept**: Pulls and summarizes recent headlines related to selected industries, displaying them in a live dashboard.

**Vibe Coding Workflow**:

- Used AI to integrate licensed RSS feeds and summarize content in under 100 words per article.
- AI initially produced code that called unofficial APIs, which had to be replaced for compliance reasons.

  **Outcome**: Prototype completed in **three days**, but with API costs that made scaling impractical without monetization.

**5.4.5 Lessons Across Additional Projects**

Across these smaller projects, several recurring themes emerged:

- **Speed**: Even complex, multi-feature tools could be prototyped in under a week.
- **Data Compliance**: AI often defaulted to free or unofficial data sources, requiring manual correction for legality and reliability.
- **Accuracy vs. Creativity**: AI was highly effective at generating *structural* code but less reliable for *content accuracy* or domain-specific compliance.
- **Iteration Cost**: Vibe coding reduced the barrier to trying entirely new ideas — making "build first, validate later" a realistic workflow.

## 6. Findings & Analysis

This section consolidates observations and metrics from all projects developed during the research period — **EconoFacts**, **BusinAI**, **SimuStocks**, and the additional prototypes — in order to identify recurring strengths, weaknesses, and implications of vibe coding as a development paradigm.

### 6.1 Development Efficiency

#### 6.1.1 Time Savings

Across all projects, vibe coding consistently reduced prototype development time by **60–80%** compared to estimated manual coding timelines.

| Project | Manual Coding Estimate | Vibe Coding Time | Reduction |
|---|---|---|---|
| EconoFacts | 7–10 days | 2 days | ~75% |
| BusinAI | 3–4 weeks | 5 days | ~70% |
| SimuStocks | 3–4 weeks | 7 days | ~65% |
| QuikInvoice | 2–3 days | 1 day | ~60% |

The largest time savings occurred during:

- **UI boilerplate creation** (React components, forms, tables).
- **API scaffolding** (Node.js/Express routes, database connection code).
- **Simple algorithm generation** (price simulations, quiz logic).

#### 6.1.2 Iteration Speed

Vibe coding shortened feedback loops:

- New features could be added in **hours instead of days**.
- Bugs were often resolved in 1–3 conversational exchanges with the AI.
- Developers could pivot ideas mid-build without discarding large amounts of code.

### 6.2 Quality and Maintainability

#### 6.2.1 Code Clarity

AI-generated code was generally readable, but:

- **Inconsistent style**: Variable naming conventions and commenting practices varied between sessions.
- **Sparse documentation**: Unless explicitly prompted, the AI did not add inline comments or docstrings.
- **Boilerplate redundancy**: Duplicate functions sometimes appeared when integrating multiple AI-generated modules.

#### 6.2.2 Maintainability Challenges

- Projects like **BusinAI** and **SimuStocks** revealed that **state management** and **scaling** were weak points.
- AI occasionally hardcoded configuration values, making later adjustments more complex.
- Without human refactoring, technical debt accumulated quickly.

### 6.3 Creativity and Feature Scope

#### 6.3.1 Emergent Features

AI suggestions often introduced unplanned but valuable features:

- EconoFacts: AI proposed a “save favorite facts” function based on engagement patterns.
- SimuStocks: AI recommended a leaderboard for player performance.

- BusinAI: AI generated a SWOT analysis module without being explicitly requested.

#### 6.3.2 Risk of Scope Creep

The ease of adding features sometimes led to **overexpansion**, increasing integration complexity and delaying completion.

### 6.4 Common Technical Challenges

Across all projects, several patterns emerged:

1. **Data Accuracy & Compliance** — Especially in EconoFacts and MarketPulse, AI-generated content or code for APIs required manual verification for accuracy and legality.
2. **Performance Optimization** — Infinite scrolling (EconoFacts) and real-time updates (SimuStocks) needed human-led optimization beyond AI defaults.
3. **Integration Gaps** — Combining separately generated modules often exposed mismatched data formats or function names.
4. **Security Oversight** — Without explicit prompts, the AI rarely implemented robust input validation or security best practices.

### 6.5 User Engagement Patterns

Small-scale user testing revealed:

- High initial engagement in content-driven apps like EconoFacts (average 27 facts/session) and game-like apps like SimuStocks (14 minutes/session).
- Functional tools like QuikInvoice saw short sessions but high repeat usage for their utility.
- AI-driven advice tools like BusinAI were rated more useful when outputs were industry-specific, suggesting a need for curated datasets.

### 6.6 Links to Theoretical Frameworks

1. **Flow Theory** — Vibe coding kept the developer in a creative rhythm, reducing interruptions from syntax errors and boilerplate coding.
2. **Distributed Cognition** — The human–AI partnership clearly split cognitive load: the AI handled repetitive or technical detail work, while the human directed overall vision and verified outputs.
3. **Automation Bias** — Developers occasionally accepted AI-generated code or facts without verification, leading to factual or functional errors.
4. **Agile/Lean Principles** — Rapid prototyping and short iteration cycles aligned closely with Lean Startup ideals, but lacked the discipline of formal Agile processes.
5. **Democratization of Software** — Several of these apps could feasibly have been built by non-programmers with domain expertise, underscoring vibe coding's accessibility.

### 6.7 Summary of Findings

- **Strengths**: Speed, flexibility, low barrier to entry, creativity boost.
- **Weaknesses**: Code maintainability, data accuracy, performance optimization.
- **Opportunities**: Educational use, early-stage startup prototyping, niche content platforms.
- **Risks**: Quality control issues, security vulnerabilities, unsustainable scaling without human intervention.

## 7. Discussion

The case studies and findings presented in Sections 5 and 6 illustrate both the transformative potential and the inherent limitations of vibe coding. By situating these results within broader technological and societal

trends, we can assess the paradigm's implications for software development practices, professional roles, and the democratization of programming.

### 7.1 Positioning Vibe Coding in the Evolution of Software Development

Software development has historically progressed through waves of abstraction:

1. **Low-level languages** (assembly, machine code).
2. **High-level procedural languages** (C, Pascal).
3. **Object-oriented and scripting languages** (Java, Python).
4. **Low-code/no-code platforms** (Airtable, Bubble).
5. **AI-assisted coding** (Copilot, ChatGPT, Cursor).

Vibe coding represents the next abstraction: **goal-oriented programming via natural language**. Instead of writing explicit instructions, developers describe desired outcomes, and the AI fills in the implementation details. This shift parallels earlier transitions, such as from assembly to C, but with a far sharper reduction in the technical barrier to entry.

While low-code tools constrained creators to predefined components, vibe coding preserves flexibility: AI can generate bespoke features outside any fixed library. In practice, this means a broader range of ideas can be tested without the structural limitations of platform-specific frameworks.

### 7.2 Impact on Professional Roles

#### 7.2.1 The Developer as Orchestrator

Findings from EconoFacts, BusinAI, and SimuStocks indicate that the human role shifts from *code author* to *code director*. This aligns with the **distributed cognition** model (Section 3.2), where humans define objectives, evaluate outputs, and intervene selectively to correct or refine AI work.

#### 7.2.2 Risks of Skill Erosion

If developers rely exclusively on AI-generated code, they risk losing low-level coding proficiency. In high-stakes domains — such as finance, healthcare, or aerospace — where safety, compliance, and optimization are paramount, the erosion of foundational skills could hinder the ability to diagnose critical issues.

#### 7.2.3 Expansion of Who Can Build Software

From a democratization standpoint, vibe coding lowers the barrier for domain experts in fields like education, economics, or small business to create tailored software without formal programming backgrounds. This mirrors the rise of low-code tools, but with fewer limitations on scope.

### 7.3 Alignment and Divergence from Agile/Lean Principles

Vibe coding's rapid iteration and low cost of experimentation align well with **Lean Startup** principles — "build, measure, learn" cycles can occur in hours instead of weeks.

However:

- **Documentation gaps** make long-term team collaboration harder.
- **Inconsistent style** across AI sessions complicates shared code ownership.
- Agile's emphasis on sustainable development pace may be undermined by the temptation to continuously add features without proper refactoring.

The findings suggest that **formal project management discipline** is necessary to prevent scope creep and technical debt, even in fast-moving AI-driven projects.

### 7.4 Creativity and Scope Expansion

One of the most striking results is how often unplanned features emerged directly from AI suggestions:

- AI recommending a leaderboard for SimuStocks.
- AI proposing a "save favorite facts" function in EconoFacts.
- AI generating a SWOT module in BusinAI without being prompted.

This supports the **flow theory** connection (Section 3.1): developers stayed in a creative rhythm, open to unexpected enhancements. However, this also contributed to **scope creep**, with feature additions extending project timelines and increasing integration complexity.

### 7.5 Limitations in Quality and Maintainability

Despite the productivity gains, the analysis reveals recurring weaknesses:

- **Performance optimization** required human-led intervention.
- **Security best practices** were often omitted unless explicitly requested.
- **Maintainability** suffered due to inconsistent naming conventions, sparse documentation, and redundant code segments.

These weaknesses underscore that while vibe coding can accelerate **MVP creation**, it does not eliminate the need for **traditional engineering practices** for production-ready systems.

### 7.6 Societal Implications

#### 7.6.1 Democratization and Inclusivity

The low barrier to entry could broaden participation in software creation, leading to more diverse problem-solving perspectives and niche applications — such as EconoFacts, which targets a specific intersection of education and economics.

#### 7.6.2 Risk of "App Pollution"

The same ease of creation could lead to an abundance of poorly maintained or insecure applications, creating a digital ecosystem cluttered with short-lived, low-quality software.

#### 7.6.3 Educational Transformation

Vibe coding could become a powerful educational tool: teaching programming concepts through *outcome-focused creation* rather than syntax drills. Tools like SimuStocks could themselves be built by students as learning exercises, reinforcing domain knowledge while introducing AI-augmented software design.

### 7.7 Future Directions

Based on these findings, several developments seem likely:

1. **Hybrid AI–Human Teams** — Developers specializing in integrating AI-generated components with human-written core modules.
2. **Best Practice Frameworks for Vibe Coding** — Community-driven style guides and QA protocols tailored to AI-assisted workflows.
3. **Domain-Specific AI Models** — Trained for finance, healthcare, or other industries to improve accuracy and compliance.
4. **Integration with Agentic Coding** — Combining vibe coding's conversational flexibility with autonomous "agent" systems that can execute end-to-end build–test–deploy cycles.

### 7.8 Summary

Vibe coding represents a significant step toward natural-language-driven software creation, offering speed, flexibility, and accessibility unprecedented in the history of programming. Yet, it is not a wholesale replacement for traditional development methods. Instead, it functions best as a **creative accelerator** and

**prototyping engine**, with human oversight essential for ensuring long-term quality, security, and scalability.

## 8. Conclusion & Recommendations

### 8.1 Summary of Key Insights

This research set out to evaluate *vibe coding*—a conversational, AI-driven approach to software development—through the lens of multiple real-world projects, including **EconoFacts**, **BusinAI**, **SimuStocks**, and several smaller prototypes. By combining development logs, user testing, and theoretical analysis, the study produced several clear conclusions:

1. **Vibe coding drastically accelerates prototyping** — Cutting development time by 60–80% compared to traditional coding for similar projects.
2. **Creativity flourishes in the conversational workflow** — AI suggestions frequently introduced valuable, unplanned features that might not have emerged in a strictly planned process.
3. **Human oversight is essential** — Issues in performance optimization, security, and factual accuracy were common when outputs were accepted without rigorous review.
4. **Maintainability remains a challenge** — Inconsistent style, sparse documentation, and redundant code segments often hindered long-term scalability.
5. **Potential for democratization is high** — Non-programmers with domain expertise could feasibly build functional applications, broadening participation in software creation.

### 8.2 Practical Recommendations

**For Independent Developers**

- Use vibe coding for **MVPs and prototypes**, but allocate dedicated time for **refactoring** before scaling.
- Develop a **prompting strategy**: be explicit about security, documentation, and performance requirements from the start.
- Maintain a **version-controlled repository** and manually review all AI-generated code before deployment.

**For Startups**

- Leverage vibe coding to **test market ideas quickly** before committing major resources.
- Combine AI-generated components with **critical human-engineered modules** for core business logic and sensitive data handling.
- Implement **QA protocols** tailored to AI-assisted code, including static analysis and security scanning.

**For Educators**

- Integrate vibe coding into **introductory programming courses** as a way to teach computational thinking without the initial syntax barrier.
- Use projects like SimuStocks as **capstone exercises**, where students design and refine an AI-assisted app while learning to evaluate outputs critically.
- Emphasize **verification skills** to counteract automation bias.

### 8.3 Recommendations for Future Research

1. **Longitudinal Studies** — Examine the maintainability and evolution of vibe-coded projects over 1–2 years.
2. **Domain-Specific Model Training** — Explore whether fine-tuning AI models on specific industries reduces factual and compliance errors.

3. **Collaborative Vibe Coding** — Assess how teams, rather than individuals, can integrate AI-driven workflows into coordinated development.
4. **Metrics for Creative Output** — Develop tools to measure how AI-assisted coding impacts the originality and diversity of software features.

**8.4 Final Reflection**

Vibe coding is not the end of programming as we know it; it is a **new layer of abstraction** that shifts the developer's role from *author* to *orchestrator*. In its current form, it excels as a **creative accelerator** and a **gateway to software creation** for those outside traditional engineering circles.

The work documented here—spanning economic education platforms, AI consulting tools, stock market simulations, and rapid utility prototypes—demonstrates that with clear goals, iterative prompting, and disciplined human oversight, vibe coding can turn ideas into working software in a fraction of the time once required.

As the technology matures, the most successful practitioners will likely be those who combine **AI fluency** with **foundational engineering discipline**—balancing speed with stability, and creativity with critical evaluation.

**References**

1. *(APA 7th edition style — note that URLs are direct where needed, with access dates suggested for final formatting.)*
2. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … Thomas, D. (2001). *Manifesto for Agile Software Development*. Retrieved from https://agilemanifesto.org
3. Business Insider. (2025). *Vibe coding startups are pushing AI to generate most of their code*. Retrieved March 2025, from https://www.businessinsider.com
4. Chen, M., et al. (2023). Evaluating large language models trained on code. *arXiv preprint* arXiv:2107.03374.
5. Csikszentmihalyi, M. (1990). *Flow: The psychology of optimal experience*. New York: Harper & Row.
6. Dibia, V. (2022). Towards practical human-AI collaboration for rapid prototyping. *Proceedings of the CHI Conference on Human Factors in Computing Systems*.
7. Hutchins, E. (1995). *Cognition in the wild*. MIT Press.
8. Karpathy, A. (2025). Vibe coding definition [Social media post].
9. Parasuraman, R., & Riley, V. (1997). Humans and automation: Use, misuse, disuse, abuse. *Human Factors*, 39(2), 230–253.
10. Ries, E. (2011). *The Lean Startup*. Crown Business.
11. Sanders, J., et al. (2024). Security vulnerabilities in AI-generated code: A systematic review. *Journal of Systems and Software*, 195, 111518.
12. Svyatkovskiy, A., et al. (2021). IntelliCode Compose: Code generation using transformer. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
13. Vaithilingam, P., et al. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *CHI Conference on Human Factors in Computing Systems*.
14. von Hippel, E. (2005). *Democratizing innovation*. MIT Press.
15. Zhou, J., et al. (2024). Conversational programming: A comparison with traditional IDE workflows. *arXiv preprint* arXiv:2404.04567.