Agile-AI: Agentic Framework for AI-Assisted Software Development Using Scrum Principles

Technical Report v1.0

Prof. Dr. Andreas Haja

Laboratory for Innovations in Engineering
University of Applied Sciences Emden/Leer, Germany
andreas.haja@hs-emden-leer.de

***Abstract*—This paper presents Agile-AI, a framework that transfers established Agile project management principles to AI-assisted software development—a novel approach that brings structured methodology to an otherwise ad-hoc practice. The framework employs three specialized LLM agents (Visionary, Architect, and Sprinter) that guide users through structured phases of requirements engineering, technical planning, and iterative implementation. Unlike unstructured approaches where developers interact with AI assistants in often chaotic dialogues, Agile-AI positions users as product owners who orchestrate AI agents through defined workflows. The framework contributes to advancing engineering practice into the AI era by enabling engineers to leverage AI capabilities systematically while maintaining professional rigor. Process discipline emerges through guided, structured dialogue: agents ask clarifying questions, make suggestions, and provide critical feedback, leading engineers to describe, structure, and detail their intentions comprehensively. Additionally, architectural constraints ensure that agents refuse vague specifications, preserve user requirements across phases, and maintain strict scope boundaries during implementation. The framework was evaluated in a pilot study with approximately 30 Master's students at the University of Applied Sciences Emden/Leer, where participants successfully developed substantial software applications. Results demonstrate that the structured approach enables engineers to guide AI agents effectively, validating the framework's contribution to bringing engineering workflows into the age of AI. This paper presents the conceptual framework, technical architecture, implementation details, and evaluation results.**

***Index Terms*—Agile, Scrum, LLM Agents, AI-Assisted Development, Software Engineering, Process Methodology**

1. Introduction

The emergence of Large Language Model (LLM) coding assistants has fundamentally transformed software development practice. Tools like GitHub Copilot, Cursor, and Claude have demonstrated that AI systems can generate functional code from natural language descriptions, with empirical studies showing measurable productivity gains in collaborative development contexts [15]. However, this transformation has largely occurred without structured methodologies—developers interact with AI assistants ad-hoc, leading to inconsistent results, requirement drift, and difficulty maintaining project coherence across multiple sessions.

This paper presents Agile-AI, a framework that addresses these challenges through a key innovation: transferring established Agile project management principles to AI-assisted development. Rather than inventing new processes, Agile-AI adapts proven Scrum methodology—with its emphasis on clear roles, defined artifacts, and structured iterations—to the unique characteristics of human-AI collaboration. The framework organizes the entire development process around three specialized agents—Visionary, Architect, and Sprinter—each responsible for distinct phases of the software development lifecycle.

1.1. The Problem with Vibe Coding

Current AI-assisted development typically follows an unstructured pattern: developers describe what they want, receive generated code, iterate through corrections, and eventually arrive at working solutions. This approach has been termed *vibe coding*—a practice where developers rely on intuition and conversational flow rather than systematic methodology when working with AI assistants [14]. Vibe coding has gained significant traction due to its low barrier to entry and ability to produce rapid prototypes. However, while it excels at generating initial implementations quickly, vibe coding exhibits significant problems when systematic, reliable delivery of specific requirements is needed:

- **Requirement Drift**: Without explicit documentation, initial requirements become lost or distorted through multiple interaction cycles
- **Scope Creep**: The ease of requesting "one more feature" leads to uncontrolled expansion
- **Inconsistent Quality**: Ad-hoc interactions produce varying code quality depending on prompt phrasing
- **Context Compaction**: When context windows exceed their limits, AI systems compress conversation

history, continuing with significantly reduced knowledge—a change that can be easily overlooked if users miss the notification amid ongoing output
- **Chaotic Dialogue**: The conversation between developer and AI may meander without clear structure, mixing requirements, implementation details, and debugging in unpredictable sequences

A common pattern in *vibe coding* is optimizing for speed of initial output rather than systematic coverage of requirements. A developer might quickly generate a working login form but discover weeks later that error handling, accessibility, or edge cases were never specified—and the AI, lacking explicit instructions, simply omitted them. Agile-AI addresses these limitations not by abandoning AI-assisted development, but by applying Agile project management discipline to structure the human-AI collaboration.

These problems mirror challenges that Scrum methodology was designed to address in traditional software development with purely human teams [1]. The iterative nature of Scrum, with its emphasis on clear roles, defined artifacts, and structured ceremonies, provides a proven framework for managing complexity—one that can be adapted for AI-mediated development.

1.2. Engineering Practice in the AI Era

Beyond addressing the problems described above, Agile-AI contributes to a broader transformation: bringing engineering development processes into the age of Artificial Intelligence. As AI systems assume increasing responsibility for technical implementation tasks, the question arises: how should engineers structure their interaction with these tools? Current practice often defaults to unstructured dialogue—the *vibe coding* pattern described in Section 1.1. Agile-AI proposes an alternative: applying the same project management discipline that has proven effective for human teams to the coordination of AI agents.

This transformation reflects a fundamental shift in engineering practice that extends well beyond software development. Engineers are transitioning from hands-on technical executors to project leaders who guide and supervise AI capabilities—defining what should be built rather than building it directly. AI systems increasingly handle implementation details that once required years of specialized training: code generation, structural analysis, signal processing, and system integration. Recent research demonstrates this breadth: Regenwetter et al. [21] provide a comprehensive review of generative models in engineering design, while Wang et al. [22] present multi-agent LLM frameworks for mechatronics design automation. The engineer's role evolves from technical execution to strategic guidance: specifying objectives clearly, validating AI outputs systematically, and ensuring that results meet professional standards.

Traditional engineering education emphasizes deep technical skills: programming, mathematical modeling, system design. While these foundations remain valuable for understanding and validating AI outputs, the day-to-day practice of engineering increasingly involves specifying requirements precisely, orchestrating AI tools systematically, and maintaining project coherence across multiple AI interactions—skills that current education and tooling inadequately address.

Agile-AI explicitly supports this evolution by providing structured processes designed for AI-assisted development. The framework treats engineers as product owners who define requirements and validate results, while AI agents handle implementation details. This role distribution anticipates a future where engineers increasingly guide AI systems toward correct solutions rather than implementing solutions themselves.

1.3. The Agile-AI Approach

Agile-AI reimagines the relationship between human developers and AI assistants by introducing three specialized agents that embody different aspects of the Scrum process:

1) **Visionary** (Phase 1 - Vision): Conducts structured requirements interviews, extracting and documenting user needs without inventing features or making technical decisions
2) **Architect** (Phase 2 - Structure): Validates technical feasibility, refines requirements with implementation details, and creates execution plans while preserving original user intent
3) **Sprinter** (Phase 3 - Work): Implements code autonomously within defined scope boundaries, refusing new features during active sprints and maintaining strict quality standards

This separation of concerns ensures that each phase receives appropriate attention and that the constraints of one phase cannot be bypassed by jumping to another. The Visionary cannot make technical decisions; the Architect cannot write production code; the Sprinter cannot accept scope changes. These architectural constraints enforce the process discipline that ad-hoc approaches lack.

1.4. Contributions

This paper makes the following contributions to AI-assisted software engineering:

- **Three-Agent Architecture**: A framework that separates requirements, architecture, and implementation into distinct phases with specialized AI agents, preventing the role confusion that plagues unstructured AI development
- **Artifact Preservation**: Rules and tagging conventions that ensure user requirements survive technical refinement, addressing the *requirement drift* problem inherent in iterative AI interactions
- **Scope Guard Mechanism**: Architectural enforcement of sprint boundaries that prevents *feature creep* during implementation, making scope discipline a technical constraint rather than a matter of willpower
- **IDE-Integrated Implementation**: A portable, Markdown-based agent definition system that works across multiple AI coding assistants without requiring custom infrastructure

- **Open-Source Tool Landscape**: The framework builds on open-source tools (VS Code and OpenCode) rather than proprietary solutions like Claude Code or OpenAI Codex, avoiding provider lock-in and enabling users to switch between different LLM providers—even during active workflows—without losing project continuity
- **Command-Based Workflow**: Slash commands that create clear phase transitions, enabling progress tracking and preventing premature advancement
- **Empirical Validation**: Results from a pilot study with approximately 30 Master's students, demonstrating that engineers can successfully develop software using the framework

The remainder of this paper is organized as follows: Section 2 reviews related work in agile methodologies and AI-assisted development. Section 3 presents the three-phase workflow in detail. Section 4 describes the agent architecture and implementation. Section 5 presents evaluation results from the pilot study. Section 6 discusses limitations and future work.

2. Background and Related Work

2.1. Scrum Methodology

Scrum provides a framework for managing complex projects through iterative development cycles called sprints [1], [9]. Originally developed in the 1990s, Scrum has been successfully applied for decades not only in software development but also in engineering-intensive fields such as product development, manufacturing, and systems engineering. The methodology defines three core roles: the Product Owner who represents stakeholder interests and maintains the product vision; the Scrum Master who facilitates the process and removes impediments; and the Development Team who transforms requirements into working software.

Key Scrum artifacts include the Product Backlog (a prioritized list of requirements), the Sprint Backlog (work selected for the current iteration), and the Increment (the working product delivered each sprint). Ceremonies such as Sprint Planning, Daily Standups, Sprint Reviews, and Retrospectives create regular opportunities for inspection and adaptation.

The effectiveness of Scrum emerges from how these elements work together as a system [4]. Sprint boundaries prevent uncontrolled scope changes. The Product Owner's prioritization directs effort toward high-value deliverables. Regular reviews keep stakeholders informed and involved. Retrospectives identify concrete process improvements. These mechanisms have proven effective across diverse project types and team configurations.

However, a common challenge in educational and professional settings is maintaining process discipline under deadline pressure. Based on the author's experience in large-scale industrial development environments and a decade of conducting an advanced project management simulation—in which student teams design and build autonomous robots—a consistent pattern emerges: teams initially follow established methodologies but progressively abandon structured practices as deadlines approach, reverting to ad-hoc problem-solving to "make things work." This pattern reflects a fundamental tension: engineers must simultaneously manage technical implementation details and project management methodology, yet tend to immerse themselves in technical problem-solving at the expense of process discipline. Agile-AI addresses this challenge by delegating technical implementation to AI agents—with the degree of delegation adjustable from full autonomy to assisted co-development with a human—freeing team members to focus on requirements validation and systematic progress tracking without the cognitive burden of holding both domains in mind simultaneously.

2.2. AI-Assisted Development and Engineering

The release of GitHub Copilot in 2021 marked a paradigm shift in software development, demonstrating that LLMs could generate functional code from natural language and code context [13]. This capability builds on *foundation models*—large language models such as GPT-4 [11] and Claude [10]—which provide the underlying intelligence for code generation. On top of these foundation models, a growing ecosystem of development tools has emerged: IDE integrations like Cursor [12] that embed AI assistance into traditional development workflows, and increasingly autonomous coding agents such as OpenAI Codex, Claude Code, and Gemini CLI that can navigate large codebases and execute multi-step tasks with minimal human intervention.

Empirical research has begun to quantify the effects of AI coding assistants. Song et al. [15] analyzed GitHub Copilot usage data across open-source projects and found that AI assistance increased project-level code contributions by 5.9%, driven by both higher individual productivity and increased developer participation. However, their study also revealed a tradeoff: coordination time for code integration increased by 8%, suggesting that AI-generated contributions require additional review effort. Cui et al. [16] conducted a controlled field experiment and found significant productivity gains, though effects varied by task complexity and developer experience.

Research on educational contexts has examined both opportunities and challenges. Denny et al. [6] found that students using Copilot developed different problem-solving strategies, focusing more on problem decomposition than syntax details. Becker et al. [8] identified challenges including over-reliance and reduced learning of fundamental concepts—concerns that highlight the importance of structured approaches to AI-assisted development.

Beyond code generation, research has explored AI assistance for requirements engineering—a domain directly relevant to Agile-AI's Visionary phase. Cheng et al. [23] provide a systematic literature review of generative AI applications in requirements engineering, documenting approaches for requirements elicitation, analysis, and validation. This growing body of work suggests that structured AI assistance for early development phases—not just implementation—represents a significant research direction.

Recent developments have moved beyond simple au-

tocomplete toward semi-autonomous development agents capable of navigating large codebases, understanding technical documentation, and solving complex tasks independently. However, experience with these systems reveals that structured project management and deliberate human guidance significantly improve outcomes. Without such guidance, autonomous agents may solve the wrong problem, introduce architectural inconsistencies, or lose sight of overarching project goals. The developer—or, in Agile-AI's framing, the project leader—remains responsible for maintaining coherence across sessions, tracking requirements, and ensuring that AI-generated outputs align with intended objectives.

Recent research has also explored *multi-agent* LLM systems that coordinate multiple AI agents for software development. He et al. [17] survey over 100 such systems, with MetaGPT [18] representing seminal work that introduced role-based agents collaborating through standardized outputs. Several frameworks have adopted agile terminology—AgileCoder [19] assigns Scrum roles to LLM agents, and Cinkusz et al. [20] explored cognitive agents for backlog refinement. However, these approaches focus on *automating* development: AI agents collaborate with each other while the human provides initial specifications and receives generated code. Agile-AI takes a fundamentally different approach, focusing on *orchestrating* the human-AI interaction through structured project management. Rather than replacing the developer, it positions the human as project leader who defines requirements, validates outputs, and maintains project coherence—while AI agents handle implementation details.

3. The Three-Phase Workflow

When developers work with AI assistants without structure, conversations tend to jump between "what do I want?", "how should it work?", and "just build it"—often in the same prompt. The AI obliges, but requirements get lost, technical decisions happen implicitly, and nobody remembers what was agreed upon three sessions ago.

Agile-AI enforces a simple separation: first figure out what to build (Phase 1), then figure out how to build it (Phase 2), then build it (Phase 3). Each phase has its own agent that refuses to do the other phases' work. The Visionary will not discuss databases. The Architect will not write production code. The Sprinter will not accept new features mid-sprint. These restrictions are not bureaucracy—they prevent the mode-switching that causes requirements to drift and scope to creep.

Figure 1 illustrates this three-phase workflow and shows how artifacts flow from one phase to the next. The following subsections describe each phase in detail: what the agent does, what it refuses to do, and what artifacts it produces.

3.1. Phase 1: Create Vision (Visionary Agent)

Most users start with a rough idea: "I want an app that does X." The Visionary's job is to turn that rough idea into a clear project description through conversation. This means asking questions, making suggestions, and sometimes proposing features the user hadn't considered—not to invent requirements, but to help the user discover what they actually want.

The phase produces two main outputs: a *project vision* (a short document describing what the software should achieve and for whom) and a *feature list* (concrete functionalities grouped by topic, each with clear success criteria). In Scrum terminology, these are called the Product Vision and Product Backlog, but the underlying idea is simple: before building anything, write down what you're building and how you'll know when it's done.

The Interview Process

The Visionary guides the user through a structured conversation in five steps:

1) **Explore**: The user describes their idea freely. The agent listens, asks clarifying questions, and identifies gaps—but does not impose structure yet.
2) **Define**: Together, they formulate who the software is for, what problem it solves, and what is explicitly out of scope. This prevents the common problem of building something nobody asked for.
3) **Structure**: The agent proposes major feature groups—called *Epics* in Scrum—and breaks them down into smaller, implementable units called *User Stories*. Think of building an autonomous robot: Epics are major subsystems like "Navigation" or "Power Management", while User Stories are the concrete tasks within them ("robot can detect obstacles", "battery level is displayed"). In Phase 3, these User Stories will be grouped into *Sprints*—focused implementation cycles that, once completed, result in a fully realized Epic. The user confirms, adjusts, or adds to these groups.
4) **Detail**: For each feature group, they define specific functionalities with verifiable success criteria—not "make it user-friendly" but "users can complete checkout in under 3 clicks."
5) **Validate**: Before finishing, the agent checks that all functionalities are specific enough for the next phase to plan the technical approach. Vague items get refined until they are concrete.

Each step ends with explicit user confirmation. If the user wants to jump straight to coding, the Visionary will suggest completing the vision first—experience shows that skipping this step leads to wasted effort when requirements turn out to be unclear.

The Visionary also maintains a written protocol of the conversation. AI assistants typically lose context between sessions; this protocol serves as memory that survives session boundaries, allowing work to resume where it left off.

With the vision and feature list complete, Phase 2 takes over to determine *how* these features should be built.

3.2. Phase 2: Structure Work (Architect Agent)

Phase 1 answered the question "What should we build?"—capturing the user's vision in concrete terms. Phase 2 builds on these results and answers the follow-up question: "How do we actually build this?" The Architect reads the Phase 1

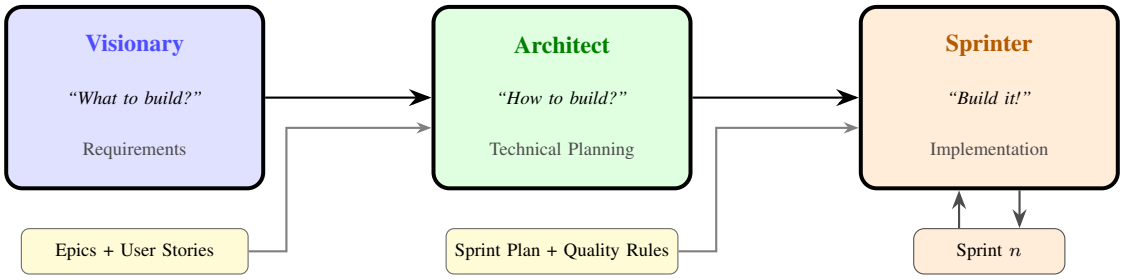

Figure 1. The Agile-AI workflow. The Visionary clarifies *what* to build (Epics and User Stories), the Architect plans *how* to build it (technologies, sprint plan, quality criteria), and the Sprinter executes iterative sprints until all features are implemented.

artifacts—the project vision and product backlog—assesses technical feasibility, selects suitable technologies, and creates an execution plan—all while explaining decisions in terms the user can understand.

A key challenge is that users arrive with varying technical backgrounds. A computer science student needs different explanations than a business major. The Architect addresses this through an explicit *competence check* at the start of the phase: "How technical are you? Are you a developer yourself, or more on the business side?" Based on the answer, the agent adjusts its communication style—using analogies and avoiding jargon for beginners, or speaking directly in technical terms for experienced developers.

The Five-Step Architecture Process

Like the Visionary, the Architect agent follows a structured workflow to produce its outputs:

1) **Analyze**: The agent reads the Phase 1 artifacts, assesses technical feasibility, and asks clarifying questions. Is this a new project or does existing code need to be integrated?
2) **Decide**: The agent proposes suitable technologies (e.g. programming languages, frameworks, databases) and explains why each choice fits the project. The user confirms or requests alternatives.
3) **Refine**: The agent walks through each feature group and adds technical tasks required for implementation. These additions are clearly tagged, so they remain distinguishable from original user requirements.
4) **Define**: Together, the agent and user establish quality standards: What does "done" mean for this project? The agent proposes criteria appropriate to the project's scope—a quick prototype needs different standards than production software.
5) **Plan**: The agent organizes the User Stories from Phase 1 into a logical Sprint sequence. What must come first? Which features depend on others? The result is a concrete build sequence with clear goals and rough effort estimates.

The Mentor Role

Unlike the Visionary, who mainly asks questions to understand user needs, the Architect also teaches. Not all users might be familiar with the technologies involved. In such cases, the Architect adopts a mentor role, proactively explaining concepts rather than waiting for the user to ask.

This educational approach serves two purposes. First, it helps users make informed decisions about their own project. Second, it builds understanding that will be useful during Phase 3, when the user needs to validate whether the implementation matches their expectations.

Phase 2 produces three major outputs: an updated project vision that now includes technical architecture decisions, a refined product backlog with technical enabler tasks added, and a Definition of Done—the quality contract that will govern Phase 3. Together, these artifacts answer the question "how will we build this?" and provide the blueprint for the actual implementation in Phase 3.

3.3. Phase 3: Do the Work (Sprinter Agent)

Phase 2 answered the question "How should we build this?"—producing a sprint plan, technology decisions, and quality criteria. Phase 3 takes these plans and turns them into working software. The Sprinter agent implements the product through iterative development cycles, each ending with a testable increment that the user can evaluate. Unlike the previous phases, which focus on dialogue and documentation, Phase 3 involves actual code generation. The Sprinter writes to a dedicated output directory, keeping generated code separate from the planning artifacts.

Throughout implementation, the Sprinter communicates in terms users can act upon. Technical errors—dependency conflicts, permission issues, port collisions—are translated into plain-language statements describing what went wrong and what the agent will try next, shielding users from implementation details they cannot influence.

A key challenge in this phase is maintaining focus. In the age of generative AI, the temptation to request "just one more feature" has grown immensely. What once required days of manual coding can now be conjured in minutes, making it dangerously easy to drift from the original vision. Users watch AI agents implement their wishes almost instantaneously and naturally think: "While we're at it, could you also add...?" Without discipline, this pattern leads to scope creep—the sprint goal shifts, the original plan becomes meaningless, and the project loses coherence. The Sprinter addresses this by refusing to accept new features during an active sprint. When users propose additions, the agent acknowledges the idea, logs it to a backlog refinement document for future

consideration, and redirects focus to completing the current sprint goal. This makes scope discipline a technical constraint rather than a matter of willpower.

The Sprint Cycle

Each sprint follows a three-stage cycle: planning with the user, implementation, and review with the user.

Sprint Planning. The agent reads the prioritized backlog from Phase 2, selects stories for the upcoming sprint (starting from the top), and proposes a scope to the user. This proposal includes specific stories and—critically—a testable outcome: "At the end of this sprint, you will be able to..." followed by concrete actions the user can perform to verify success. The user must explicitly agree before implementation begins. This agreement creates a contract: both parties know what will be delivered and how success will be measured.

Implementation. Once the user agrees, the Sprinter works through each task following an implement-test-fix cycle. For every feature, the agent writes the code, tests it (by running the server, executing commands, or verifying output), and fixes any issues before moving to the next task. While the goal is autonomous execution, user involvement may still be required—for example, when the agent encounters environmental issues it cannot resolve, needs clarification on ambiguous requirements, or requires feedback on visual elements it cannot evaluate. When automated testing is not feasible, the agent documents manual test steps for the user to follow later. For interactive applications, browser automation tools such as Playwright—which can programmatically control web browsers, click buttons, fill forms, and capture screenshots—allow the agent to verify behavior even for programs that would normally require human operation. Progress is tracked through a task list that mirrors the sprint backlog, with each task marked as pending, in progress, or completed.

Sprint Review. After completing all tasks, the Sprinter provides test instructions tailored to the user's operating system and technical background. The user tests the delivered increment—what Scrum calls a *potentially shippable product increment*—and provides feedback. Here the agent faces a critical distinction: is the feedback about a defect or missing element that was part of this sprint's agreed scope, or is it a new idea triggered by seeing the software in action? The former requires immediate attention; the latter belongs in the backlog for future sprints. This boundary must be maintained even when users are excited about possibilities they've just discovered. Deferred ideas are captured in a refinement document and become candidates for discussion during the next sprint planning session—ensuring that nothing is lost while keeping the current sprint focused.

Only after explicit user approval does the agent verify that all quality criteria from the Definition of Done have been met, generate a sprint report, and save a versioned snapshot of the project state using Git. This is the only point in the entire workflow where version control changes occur—ensuring that the repository contains only tested, approved increments.

4. Agent Architecture

The three agents described in Section 3 share a common technical architecture that enables their specialized behaviors while maintaining portability across different AI coding assistants.

4.1. Agent Definition Structure

Each agent is defined through a Markdown file with YAML frontmatter—a format readable by both humans and machines. The frontmatter specifies operational parameters: a description that tells the IDE when to activate the agent, a temperature setting that controls response variability, and tool permissions that restrict what actions the agent can perform. The Markdown body contains the agent's instructions: its identity, behavioral rules, and constraints.

This structure allows agent behavior to be version-controlled alongside project code, reviewed in pull requests, and modified without specialized tooling. A researcher can read an agent definition to understand exactly what instructions it receives; a practitioner can adjust behaviors by editing text files.

4.2. Response Variability

Large language models use a parameter called *temperature* to control the randomness of their outputs. Higher temperature settings produce more varied and creative responses, while lower settings yield more predictable and focused outputs. This parameter directly affects how agents behave: a creative brainstorming assistant benefits from higher variability, while a code generator requires precision.

Agile-AI calibrates each agent's temperature to match its role. The Visionary operates at the highest setting, encouraging exploration of possibilities and creative suggestions during requirements discussions. The Architect uses a moderate setting, balancing analytical precision with the flexibility to explain concepts in different ways. The Sprinter operates at the lowest setting, prioritizing deterministic code generation where consistency matters more than variation. This progressive decrease in temperature reflects the workflow's transition from open-ended exploration toward precise execution.

4.3. Tool Permissions

Agents receive only the tools necessary for their phase. The Visionary has read access to templates and write access to vision artifacts—but cannot execute code or modify system files. The Architect gains additional permissions to analyze existing codebases when integrating with legacy systems. The Sprinter receives full tool access: reading, writing, editing files, and executing shell commands to run tests and start servers. This progressive permission model prevents accidental scope violations—an agent literally cannot perform actions outside its designated role.

4.4. Command Interface

Users interact with agents through slash commands that mark phase transitions. Each phase has two commands: a *start* command that initiates the phase, and a *complete* command that finalizes it. For example, `/visionary-start` begins

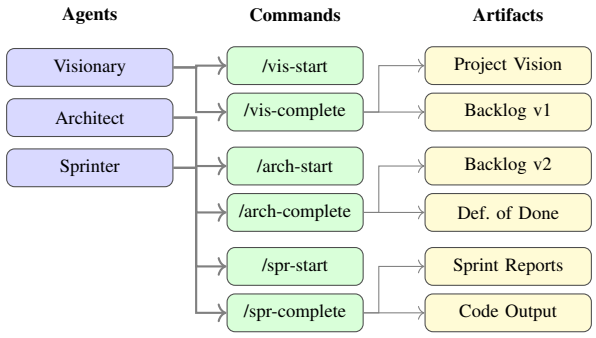

Figure 2. Framework architecture showing the relationship between agents, their commands, and the artifacts each phase produces. Arrows indicate which commands belong to which agent and what artifacts they generate.

the requirements interview, and `/visionary-complete` concludes it after generating the required artifacts.

When a user invokes a start command, the agent initiates a structured dialogue, guiding the user through the phase's workflow step by step. The agent asks questions, proposes solutions, and waits for confirmation before proceeding. Once the phase objectives are achieved, either the user can invoke the complete command directly, or the agent suggests doing so, indicating that sufficient information has been gathered to proceed to the next phase. This explicit handoff ensures that users consciously acknowledge phase transitions rather than drifting between activities.

4.5. Cross-Phase Communication

Agents do not communicate directly with each other. Instead, they exchange information through artifacts—Markdown documents that persist in the project directory. The Visionary writes a product backlog; the Architect reads that backlog and produces an enhanced version with technical details; the Sprinter reads the enhanced backlog and generates sprint reports.

This artifact-based approach offers three benefits. First, transparency: users can inspect exactly what information passes between phases by reading the documents themselves. Second, persistence: work survives session boundaries, allowing projects to span multiple days or weeks without losing context. Third, auditability: the evolution of requirements from initial idea through technical refinement to implementation is fully documented, creating a project history that can be reviewed at any time.

Figure 2 illustrates how agents, commands, and artifacts relate to each other within the framework's directory structure.

4.6. Platform Independence

A deliberate design goal was avoiding vendor lock-in. Commercial AI coding assistants such as Claude Code or Codex CLI offer powerful capabilities but create dependencies on specific providers—both in terms of API access and the proprietary infrastructure built around their tools. Agile-AI instead targets OpenCode, an open-source AI coding assistant that supports multiple LLM providers through a unified interface.

This choice offers three advantages. First, model flexibility: users can switch between Claude, GPT-4, Gemini, or other models without changing their workflow, selecting the best model for their budget and requirements. Second, transparency: both the framework and the underlying tool are open source, allowing inspection and modification. Third, longevity: the framework does not depend on any single company's continued support or pricing decisions.

The framework is distributed as a project template that users clone and customize. Agent definitions use standard Markdown files that remain human-readable and portable. Should users prefer a different AI coding assistant, adapting the definitions requires only minor reformatting to match the target platform's conventions.

5. Evaluation

The Agile-AI framework was evaluated through a pilot study conducted at the University of Applied Sciences Emden/Leer during the 2025 academic year. Approximately 30 Master's students participated, using the framework to develop software projects as part of their coursework.

5.1. Study Design

Participants were enrolled in a project management course that traditionally emphasized Scrum methodology through team-based robot development projects. For this cohort, the practical component was modified: instead of building physical robots, students used Agile-AI to develop browser-based multiplayer games. The project framing was consistent across teams: each group implemented a different classic arcade game (such as Pac-Man or Snake) designed for demonstration at the university's open house event. Requirements included browser playability, visitor participation via QR code scanning, and game control through mobile phones—creating technically demanding projects within a shared conceptual framework.

Students received an introduction to the framework and then worked in small teams, using the three-agent workflow to progress from initial idea to working software. The instructor observed their interactions with the agents, reviewed the artifacts produced at each phase, and collected informal feedback throughout the semester.

5.2. Observations

The following observations emerged from interactions with students throughout the semester. These are qualitative impressions gathered through informal conversations, artifact reviews, and classroom discussions—not systematic data collection. They provide preliminary insights but should not be interpreted as rigorous empirical findings.

Comparison with previous cohorts. In earlier years, when students built autonomous robots themselves, a consistent pattern emerged: teams initially followed Scrum practices diligently—writing reports, holding meetings, maintaining backlogs—but progressively abandoned these methods as deadlines approached. By the final weeks, project management gave way to crisis-mode problem-solving focused solely

on "making it work." With Agile-AI, this pattern was notably reduced. By delegating technical implementation to AI agents, students were effectively placed in a project leader role where their primary interaction with the project *was* project management. They could not bypass the methodology by diving into code themselves; instead, they had to communicate requirements clearly, validate outputs systematically, and maintain the artifact trail. The learning outcomes for project management appeared stronger than in previous cohorts, though this observation awaits formal measurement.

Several additional patterns emerged:

Structure reduced anxiety. Students unfamiliar with programming reported that the phased approach made AI-assisted development less overwhelming. Rather than facing a blank prompt and wondering what to ask, they followed the Visionary's interview process, which guided them toward concrete requirements. Several students noted that they would not have known where to start without this structure.

Scope discipline required enforcement. Despite the Sprinter's refusal to accept new features mid-sprint, some students attempted workarounds—rephrasing requests as "bug fixes" or starting new conversations to bypass the scope guard. This behavior highlighted both the temptation of instant feature generation and the importance of the architectural constraint.

Artifacts provided project continuity. Students who worked on their projects across multiple sessions found the artifact documents valuable for resuming work. The written records of decisions made during earlier phases eliminated the need to re-explain context to the AI assistant.

Non-programmers achieved working software. Several students with no prior programming experience successfully delivered functional applications. While the code quality varied, the projects met their stated requirements and could be demonstrated to peers.

5.3. Limitations of the Study

This pilot study provides preliminary evidence but has important limitations. The sample consisted of students in a controlled educational setting, which may not reflect professional development contexts. All observations are qualitative, gathered through informal interactions rather than systematic data collection; no formal metrics, surveys, or statistical analyses were conducted. The comparison with previous cohorts is based on instructor recollection rather than controlled measurement. A rigorous comparison with unstructured AI-assisted development was not performed, making it difficult to isolate the framework's specific contribution from confounding factors such as project type, student motivation, or the novelty effect of using AI tools.

6. Limitations and Future Work

6.1. Current Limitations

The framework has several limitations that constrain its applicability:

Single-user focus. Agile-AI currently supports one user working with AI agents. Real software projects often involve teams, requiring coordination mechanisms that the framework does not provide. Extending the artifact-based communication model to support multiple human participants represents a significant architectural challenge.

LLM behavioral variability. Despite temperature calibration and detailed instructions, large language models do not always follow agent definitions precisely. Agents occasionally drift from their prescribed roles, requiring user intervention to redirect behavior. This variability makes outcomes less predictable than traditional software tools.

Overhead for small projects. The three-phase workflow adds structure that benefits complex projects but may feel burdensome for simple tasks. A quick script or single-file utility does not require a formal vision document and sprint planning. The framework currently lacks adaptive mechanisms to scale its formality based on project scope.

Lack of quantitative evaluation. The current evidence for the framework's effectiveness is entirely qualitative. While observations from the pilot study are encouraging, measurable improvements in learning outcomes, code quality, or project success rates have not been demonstrated. Establishing such metrics and conducting controlled comparisons remains essential for validating the framework's benefits.

6.2. Future Directions

Several directions could extend the framework's capabilities:

Improved determinism through prompt orchestration. Current agent behavior depends entirely on natural language instructions interpreted by the underlying LLM. Future versions could integrate structured prompt orchestration systems—such as Promos [24], a prompt operating system that provides deterministic control flow for LLM interactions—to achieve more reliable agent behavior. Such integration would allow complex workflows to be specified declaratively, with the orchestration system ensuring that steps execute in the correct order regardless of LLM variability.

Formal evaluation. Rigorous empirical studies comparing Agile-AI to unstructured AI-assisted development would quantify the framework's benefits and identify areas for improvement. Metrics could include requirement coverage, code quality, time to completion, and user satisfaction. The ongoing pilot study with Master's students will be expanded, and additional evaluations in industrial contexts are planned.

Multi-user coordination. Supporting team-based development would require mechanisms for dividing work across multiple users, merging artifacts produced in parallel, and resolving conflicts when different team members make incompatible decisions. The artifact-based communication model provides a foundation, but significant extensions would be needed.

Programmatic Guardrails. Currently, agent constraints rely on instructions within the system prompt (soft guardrails). While effective, they are not inviolable. Future iterations could enforce critical constraints programmatically—for example, using a Python-based middleware that validates agent outputs against defined rules before executing them. This would ensure that boundaries like the scope guard are technically enforced rather than just instructed.

Adaptive formality. The framework could detect project complexity and adjust its structure accordingly—using a lightweight single-phase process for simple tasks while engaging the full three-phase workflow for substantial projects. This adaptation could be based on initial user descriptions or learned from patterns in completed projects.

7. Conclusion

This paper presented Agile-AI, a framework that brings structured project management to AI-assisted software development. By adapting Scrum methodology to human-AI collaboration, the framework addresses the requirements drift, scope creep, and chaotic dialogue that characterize unstructured approaches.

The key insight is that constraints enable productivity. The three-agent architecture—Visionary, Architect, and Sprinter—enforces separation of concerns that prevents the mode-switching responsible for many AI-assisted development failures. Each agent has a defined role and explicit boundaries: the Visionary cannot make technical decisions, the Architect cannot write production code, and the Sprinter cannot accept scope changes. These restrictions are not bureaucratic overhead but architectural safeguards that maintain project coherence.

The framework's implementation as Markdown-based agent definitions provides portability across AI coding assistants while remaining accessible to non-programmers. Users can inspect, modify, and version-control agent behavior alongside their project code. The artifact-based communication between phases creates a persistent project history that survives session boundaries and enables multi-day development efforts.

A pilot study with Master's students demonstrated that non-programmers can use the framework to deliver working software. While formal metrics await future studies, observations suggest that the structured approach reduces the anxiety of beginning AI-assisted projects and provides valuable scaffolding for users unfamiliar with software development.

As AI systems assume increasing responsibility for technical implementation, the engineering profession must develop new practices for guiding these tools effectively. Agile-AI contributes to this evolution by showing that established project management principles—clear roles, defined artifacts, iterative refinement—remain valuable even when the "development team" includes artificial intelligence.

References

[1] K. Schwaber and J. Sutherland, "The Scrum Guide," Scrum.org, 2020.

[2] M. Grimheden, "Can Agile Methods Enhance Mechatronics Design Education?" *Mechatronics*, vol. 23, no. 8, pp. 967–973, 2013.

[3] V. Mahnic, "Teaching Scrum through Team-Project Work: Students' Perceptions and Teacher's Observations," *International Journal of Engineering Education*, vol. 26, no. 1, pp. 96–110, 2010.

[4] T. Dingsøyr, S. Nerur, V. Balijepally, and N. B. Moe, "A Decade of Agile Methodologies: Towards Explaining Agile Software Development," *Journal of Systems and Software*, vol. 85, no. 6, pp. 1213–1221, 2012.

[5] J. Chen, A. Kolmos, and X. Du, "Forms of Implementation and Challenges of PBL in Engineering Education: A Review of Literature," *European Journal of Engineering Education*, vol. 46, no. 1, pp. 90–115, 2021.

[6] P. Denny, V. Kumar, and N. Giacaman, "Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language," in *Proc. 54th ACM Technical Symposium on Computer Science Education*, 2023, pp. 1136–1142.

[7] H. Xu, W. Gan, Z. Qi, J. Wu, and P. S. Yu, "Large Language Models for Education: A Survey," arXiv:2405.13001, 2024.

[8] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming Is Hard — Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation," in *Proc. 54th ACM Technical Symposium on Computer Science Education*, 2023, pp. 500–506.

[9] P. Deemer, G. Benefield, C. Larman, and B. Vodde, "The Scrum Primer: A Lightweight Guide to the Theory and Practice of Scrum (Version 2.0)," 2012. [Online]. Available: http://www.scrumprimer.org

[10] Anthropic, "Claude 3.5 Sonnet Model Card," 2024. [Online]. Available: https://www.anthropic.com

[11] OpenAI, "GPT-4 Technical Report," arXiv:2303.08774, 2024.

[12] Cursor, "Cursor: The AI-first Code Editor," 2024. [Online]. Available: https://cursor.sh

[13] GitHub, "GitHub Copilot Documentation," 2024. [Online]. Available: https://docs.github.com/copilot

[14] A. Karpathy, "Vibe Coding," X (formerly Twitter), February 2025. [Online]. Available: https://x.com/karpathy/status/1886192184808149383

[15] F. Song, A. Agarwal, and W. Wen, "The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence from GitHub Copilot," arXiv:2410.02091, 2024.

[16] K. Z. Cui, M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz, "The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers," MIT Working Paper, 2024.

[17] J. He, C. Treude, and D. Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead," arXiv preprint arXiv:2404.04834, 2024.

[18] S. Hong, M. Zhuge, J. Chen, X. Zheng, *et al.*, "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework," in *Proc. International Conference on Learning Representations (ICLR)*, 2024.

[19] M. H. Nguyen, T. P. Chau, P. X. Nguyen, and T. N. Nguyen, "AgileCoder: Dynamic Collaborative Agents for Software Development Based on Agile Methodology," in *Proc. IEEE/ACM Int. Workshop on Large Language Models and Software Engineering*, 2025.

[20] K. Cinkusz, J. A. Chudziak, and E. Niewiadomska-Szynkiewicz, "Cognitive Agents Powered by Large Language Models for Agile Software Project Management," *Electronics*, vol. 13, no. 19, 2024.

[21] L. Regenwetter, A. H. Nobari, and F. Ahmed, "Deep Generative Models in Engineering Design: A Review," *ASME Journal of Mechanical Design*, vol. 144, no. 7, 2022.

[22] Z. Wang, F. P.-W. Lo, Q. Chen, Y. Zhang, C. Lin, X. Chen, Z. Yu, A. J. Thompson, E. M. Yeatman, and B. P. L. Lo, "An LLM-enabled Multi-Agent Autonomous Mechatronics Design Framework," arXiv preprint arXiv:2504.14681, 2025.

[23] H. Cheng, J. H. Husen, Y. Lu, T. Racharak, and M. L. Keet, "Generative AI for Requirements Engineering: A Systematic Literature Review," *Software: Practice and Experience*, 2025.

[24] A. Haja, "Promos: A Prompt Operating System for Deterministic LLM Workflows," 2025. [Online]. Available: https://github.com/andreashaja/promos