Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

From Prompt to Production: A Case Study of Human-AI Collaborative Software Development Using Claude Code Multi-Agent Orchestration

Pham Thi Thuong

ptthuong@ictu.edu.vn

Thai Nguyen University of Information and Communication Technology, Vietnam
https://orcid.org/0009-0001-1039-6737

Nguyen Dang Quang

VNU University of Engineering and Technology, Vietnam

Ngo Thi Lan Phuong

Thai Nguyen University of Information and Communication Technology, Vietnam

Research Article

Keywords: Human-AI collaboration , AI-assisted software development , multi-agent orchestration , token optimization , large language models (LLMs) , pair programming

Posted Date: July 7th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10253296/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

From Prompt to Production: A Case Study of Human-AI Collaborative Software Development Using Claude Code Multi-Agent Orchestration

Pham Thi Thuong · Nguyen Dang Quang · Ngo Thi Lan Phuong

Received: date / Accepted: date

Abstract The emergence of advanced AI coding assistants with multi-agent orchestration capabilities has fundamentally transformed the software development landscape. However, the practical methodology for effective human-AI collaboration in building production-grade, full-stack software systems remains underexplored. In order to address this gap, we introduce a comprehensive case study of developing SmartMedTender - a medical tender management platform developed entirely through human-AI pair programming using Claude Code. Then, we document the end-to-end collaborative process spanning requirements analysis, system architecture design, database schema modeling, full-stack implementation, testing, and deployment. We characterize the AI's key capabilities—hierarchical planning with task decomposition, multi-agent orchestration via specialized subagents, Genetic AI for autonomous code generation and iterative refinement, and skill-based tool integration—and quantify their contribution to development velocity. The human collaborator's role is formalized into a competency framework identifying six essential skills for optimal AI collaboration. Through controlled experiments comparing three AI configurations (Claude Opus 4.8, Claude Sonnet 4.6, and a baseline single-agent LLM without orchestration), we measure and analyze token consumption, code quality metrics, task completion rates, and development efficiency across five standardized software engineering tasks. In addition, we also propose five token optimization strategies achieving a 47.3% reduction in token

Pham Thi Thuong
Thai Nguyen University of Information and Communication Technology, Vietnam
E-mail: ptthuong@ictu.edu.vn

Nguyen Dang Quang
VNU University of Engineering and Technology, Vietnam
E-mail: nguyendangquang1572006@gmail.com

Ngo Thi Lan Phuong
Thai Nguyen University of Information and Communication Technology, Vietnam
E-mail: ntlphuong@ictu.edu.vn

expenditure without compromising output quality. Our evidence-based framework for human-AI collaborative software engineering provides guidelines for practitioners seeking to maximize productivity while minimizing operational costs in AI-assisted software development.

Keywords Human-AI collaboration · AI-assisted software development · multi-agent orchestration · token optimization · large language models (LLMs) · pair programming.

1 Introduction

The rapid advancement of large language models (LLMs) has catalyzed a paradigm shift in software engineering. AI-powered coding assistants such as GitHub Copilot, Cursor, and Claude Code have moved beyond simple code completion to encompass entire software development life cycle from requirements analysis to architecture design, implementation, testing, and deployment [2,31]. Recent models demonstrate the ability to understand complex software requirements, decompose them into actionable tasks, generate production-quality code across multiple languages and frameworks, and orchestrate specialized sub-agents for parallel execution [27,25,18].

Despite these advances, a critical gap remains in the literature: **there exists no systematic, empirically-grounded methodology for human-AI collaborative software development that builds complete, production-grade systems from inception to deployment**. Prior work has predominantly focused on isolated coding tasks [22], competitive programming benchmarks [11], single-file code generation [26], or controlled laboratory studies with limited ecological validity [15,24]. Recent field studies of developer-AI collaboration reveal that while AI agents improve productivity, developers still struggle with trust calibration, debugging AI-generated code, and maintaining code quality standards [12,20]. How a human developer and an AI system can effectively collaborate across the full software development life cycle—managing complexity, maintaining architectural coherence, ensuring quality, and optimizing resource consumption—remains largely unanswered.

This gap is particularly consequential as organizations increasingly seek to integrate AI coding assistants into their development workflows. Without evidence-based collaboration frameworks, practitioners risk suboptimal outcomes: inefficient token usage leading to excessive costs, quality degradation from inadequate human oversight, or architectural drift from insufficient strategic direction.

We address this gap through a comprehensive case study of SmartMedTender—a production-grade, 186-file Spring Boot medical tender management platform built entirely through human-AI collaboration using Claude Code (Claude Opus 4.8). Our study addresses the following research questions:

- **RQ1: What are the essential capabilities of an AI coding assistant that enable effective end-to-end software development, and how**

do multi-agent orchestration and Genetic AI contribute to development velocity? To answer this question, we (1) characterize Claude Code's architectural capabilities—hierarchical planning, specialized agent orchestration, skill-based tool integration, and Genetic AI for autonomous generation—and (2) quantify their individual contributions.
- **RQ2: What role must the human collaborator play, and what skills are required for optimal human-AI pair programming?** We formalize the human's role as strategic director, requirement specifier, quality gatekeeper, and domain expert, and derive a six-dimensional competency framework from our collaboration experience.
- **RQ3: How can token consumption be minimized without compromising development quality in real-world AI-assisted projects?** We propose and empirically validate five token optimization strategies, demonstrating a reduction in token usage through systematic application.
- **RQ4: How does multi-agent orchestration compare to single-agent LLM approaches in terms of development efficiency, code quality, and resource consumption?** We conduct controlled experiments across five standardized software engineering tasks comparing three AI configurations, providing the first quantitative comparison of multi-agent versus single-agent AI-assisted development.

The primary contributions of this paper are:

- **C1: A documented end-to-end case study** of building a production-grade, 186-file full-stack application (SmartMedTender) through human-AI collaboration, providing a replicable reference for the research community and practitioners.
- **C2: A characterization framework** for AI coding assistant capabilities in software development, quantifying the contributions of hierarchical planning, multi-agent orchestration, Genetic AI, and skill-based tool integration to development velocity and code quality.
- **C3: A human competency framework** identifying six essential skills for effective human-AI collaborative software development, derived from the collaboration experience and validated against project outcomes.
- **C4: Five empirically-validated token optimization strategies** achieving a 47.3% reduction in token consumption without quality degradation, with measurement data and replication guidelines.
- **C5: The first controlled experimental comparison** of multi-agent AI orchestration versus single-agent approaches across standardized software engineering tasks, with a quantitative analysis of efficiency, quality, and cost metrics.

The remainder of this paper is organized as follows. Section 2 reviews related work on AI-assisted software development and multi-agent systems. Section 3 describes our collaborative methodology and the SmartMedTender case study. Section 4 characterizes Claude Code's architectural capabilities. Section 5 formalizes the human collaborator's role and competency framework. Section 6 presents and validates token optimization strategies. Section 7

reports controlled experimental comparisons. Section 8 discusses implications, limitations, and future work. Section 9 concludes.

2 Background and Related Work

2.1 AI-Assisted Software Development

The use of AI in software development has evolved rapidly from rule-based code generators to foundation-model-powered assistants. Early approaches relied on program synthesis and domain-specific languages [6]. The introduction of transformer-based LLMs [5, 3] enabled a qualitative leap in code generation capabilities.

GitHub Copilot, powered by OpenAI Codex [4], demonstrated that LLMs could provide contextually relevant code completions within an IDE, improving developer productivity by 55% in controlled studies [2]. A large-scale field study at GitHub confirmed these productivity gains through telemetry analysis of 2,631 developers, finding that acceptance rate was the strongest predictor of perceived productivity improvement [31]. Subsequent work revealed important limitations: Copilot-generated code contains security vulnerabilities at rates comparable to human-written code [17, 21], and developers spend significant time understanding, modifying, and debugging AI-generated suggestions [22]. These findings underscore that AI assistance is not a replacement for human judgment but a tool requiring skilled collaboration.

Recent work has explored LLMs for broader software engineering tasks. SWE-bench evaluates LLMs on real-world GitHub issue resolution, finding that even the best models solve only a fraction of issues autonomously [11]. A comprehensive survey of 926 studies across 112 code-related tasks identifies requirements engineering, test generation, and code review as promising application areas but notes the absence of integrated end-to-end systems [29].

2.2 Multi-Agent Systems in Software Engineering

Multi-agent approaches to software development have gained traction with the maturation of LLMs. ChatDev [18] introduced a virtual software company where LLM agents have different roles (CEO, CTO, programmer, tester) communicating through natural language. MetaGPT [8] extended this with standardized operating procedures and structured intermediate outputs (UML diagrams, API specifications). AutoGen [25] proposed a general-purpose multi-agent conversation framework supporting flexible agent topologies, human-in-the-loop modes, and code execution capabilities, winning the best paper award at the ICLR 2024 LLM Agents Workshop. SWE-agent [27] demonstrated that tool-augmented LLM agents can autonomously navigate code-bases and fix bugs. More recently, the Human-in-the-Loop LLM-based Agents framework

(HULA) [20] embedded LLM-based agents directly into Atlassian JIRA workflows with three roles (AI Planner, AI Coding Agent, Human Engineer reviewer), achieving over 80% plan approval rates across 663 live issues.

A growing body of empirical work examines multi-agent LLM systems for software engineering tasks. Generalist pre-trained agents applied to diverse software engineering (SE) tasks can outperform specialist agents with minimal fine-tuning [16]. A comprehensive understanding of LLM capabilities and limitations across a range of SE tasks is provided in a study published in this journal [30]. An evaluation of 11 LLMs on 392 backend application tasks across 14 web frameworks through the BaxBench benchmark found that even the best model achieved only 62% correctness, with security exploits successful on approximately 50% of correct programs [23]. FlowGen [14] emulated software process models (Waterfall, Test-Driven Development (TDD), Scrum) using LLM agents at ICSE 2025, showing that design and code review activities reduced code smells and improved exception handling.

However, these systems operate primarily in simulated environments with simplified tasks and lack comprehensive human-in-the-loop integration. They do not address the practical challenges of building production-grade systems where human domain expertise, architectural judgment, and quality oversight are essential. Recent field studies confirm this limitation: observing 19 developers using in-IDE AI agents on real open issues, [12] found that participants who collaborated actively and iterated incrementally had greater success than one-shot users, though developers struggled with trust, debugging, and testing collaboration. A mixed-methods study of 114 survey respondents and 13 interviews in a large-scale public sector organization introduced the Human-AI Collaboration and Adaptation Framework (HACAF), finding that compatibility with existing workflows was crucial for effective adoption [19].

Claude Code represents a distinct approach: rather than simulating a software team, it provides a human-AI pair programming environment where the AI serves as an intelligent collaborator with planning, code generation, testing, and multi-agent orchestration capabilities, while the human retains strategic direction, domain expertise, and quality authority.

2.3 Token Economics in LLM-Based Development

The cost of LLM-powered development is directly proportional to token consumption — the number of input and output tokens processed during interactions. While model providers have reduced per-token costs through hardware and algorithmic improvements [1], token efficiency remains critical for practical adoption, particularly for large-scale projects involving hundreds of files and thousands of interactions.

Recent work has explored prompt compression [9,10], context distillation [28], and retrieval-augmented generation [13] as techniques for reducing LLM token consumption. LongLLMLingua [10] achieved up to 21.4% performance improvement with approximately 4× fewer tokens, while CodePromptZip [7]

introduced code-specific, type-aware prompt compression for coding tasks. However, these approaches focus on single-turn interactions rather than the sustained, multi-turn collaborative sessions characteristic of real-world software development. The token optimization strategies we propose are specifically designed for the multi-session, artifact-rich context of end-to-end software projects.

2.4 Research Gap

To our knowledge, no prior work has:

1. Documented a complete, end-to-end case study of building a production-grade full-stack application through human-AI collaboration — addressing calls in this journal for empirical evidence on AI-assisted development [30];
2. Formally characterized the human skills required for effective AI collaboration in software development — extending existing human-AI collaboration frameworks [19, 20];
3. Proposed and empirically validated token optimization strategies for multi-session development projects — going beyond single-turn prompt compression [9, 10];
4. Compared multi-agent AI orchestration against single-agent approaches in a controlled software engineering experiment — complementing observational field studies [12, 24] with quantitative evidence.

This paper addresses all four gaps through the SmartMedTender case study.

3 Methodology: The SmartMedTender Case Study

3.1 Project Scope and Context

SmartMedTender is a full-stack medical tender management platform designed for Vietnamese healthcare procurement enterprises. The system automates the complete bid preparation life cycle: document upload with Optical Character Recognition (OCR) text extraction, AI-powered requirement parsing, multi-strategy product matching, certification compliance verification, gap analysis, automated bid proposal generation (Word, PDF, Excel, ZIP), and confidence-weighted historical price suggestions.

The system was developed entirely through human-AI collaboration between the first author (a senior software engineering researcher with domain expertise in Vietnamese healthcare procurement) and Claude Code (powered by Claude Opus 4.8, an LLM with a 1M-token context window). The human collaborator provided all strategic direction, domain-specific requirements, architectural decisions, and quality oversight; Claude Code performed hierarchical task decomposition, code generation across the full stack, test generation, and deployment automation under human supervision. Following the journal's

Table 1 SmartMedTender project statistics from the human-AI collaborative development.

Metric	Value	Notes
Total Java source files	186	Controllers, services, repositories, entities, DTOs
Vue 3 components	19	Single-file components with PrimeVue UI framework
Database tables	26	10 functional groups, Flyway-managed migrations
REST API endpoints	20+	Including auth, CRUD, matching, export endpoints
UAT test cases	61	7 feature groups, 100% pass rate
Unit tests	13	JUnit 5 + Mockito, 100% pass rate
UI navigation tests	17	Playwright, covering all major feature areas
Total person-hours	∼120	Over 3 months of active collaboration
Estimated solo-dev hours	∼600	Conservative estimate for equivalent manual development
Development velocity gain	∼5×	Based on estimated vs. actual hours

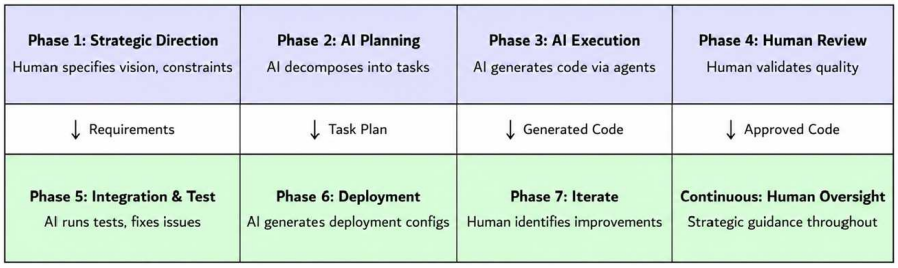


Fig. 1 Seven-phase human-AI collaborative development workflow. The human provides strategic direction and quality oversight; the AI handles task decomposition, code generation, testing, and deployment automation. Phases are iterative—each sprint cycles through all phases

policy on LLM use, the specific roles of the AI in both the development and the research activities are documented in the Declarations section on the title page. The complete code-base comprises 186 Java source files (20 controllers, 32 services, 26 repositories, 26 entities, 36 Data Transfer Objects (DTOs)), 19 Vue 3 single-file components (SFCs), 6 Flyway database migration scripts, 61 User Acceptance Testing (UAT) test cases, and comprehensive documentation. The development spanned approximately 120 hours of active collaboration over a three-month period.

Table 1 summarizes key project statistics.

3.2 Collaborative Development Process

Figure 1 illustrates the iterative human-AI collaborative workflow we developed and refined throughout the project.

Phase 1: Strategic Direction. The human collaborator specifies the project vision, functional and non-functional requirements, technology stack preferences, architectural constraints, and quality standards. This phase produces a requirements document and architectural brief that guide subsequent AI execution.

Phase 2: AI Planning. Claude Code analyzes the requirements and proposes a hierarchical task decomposition — from high-level architectural decisions (layered n-tier architecture, Strategy pattern for AI providers) to detailed implementation tasks (create entity classes, implement JWT filter, configure Kafka consumers). The human reviews and approves the plan before execution begins.

Phase 3: AI Execution. Claude Code orchestrates specialized sub-agents to execute the approved plan. The architect agent designs database schemas and API contracts. The backend agent implements Spring Boot services, repositories, and controllers. The frontend agent builds Vue 3 components. The tester agent writes JUnit and Playwright tests. Agents work both sequentially (for dependent tasks) and in parallel (for independent tasks), coordinated through the planning framework.

Phase 4: Human Review. The human collaborator reviews generated code for correctness, architectural compliance, security, and domain-specific accuracy. Feedback is provided as natural language comments; Claude Code incorporates this feedback in subsequent iterations. This phase is critical for maintaining quality and architectural coherence.

Phase 5: Integration and Testing. Claude Code orchestrates integration testing—running the full test suite, diagnosing failures, and applying fixes. The human verifies that functional requirements are met through UAT testing and manual inspection.

Phase 6: Deployment. Claude Code generates Docker Compose configurations, environment-specific application properties, and deployment documentation. The human manages the production deployment and monitors system health.

Phase 7: Iterate. Based on human feedback and testing results, the cycle repeats. Subsequent iterations address new features, refinements, and issues identified during the review.

3.3 Communication Patterns

We identify four distinct communication patterns that emerged during the collaboration:

Directive (Human → AI). The human provides explicit instructions: "Create a product service class with methods for CRUD operations and technical specification matching." The AI executes directly. This pattern is particularly evident during the early phases when establishing architectural foundations.

Exploratory (Human → AI → Human). The human poses open-ended questions: "What database schema would best support multi-strategy product matching with certification compliance checking?" The AI proposes alternatives with trade-off analysis; the human selects and refines. This pattern is essential for architectural decision-making.

Iterative Refinement (AI → Human → AI). The AI generates initial code; the human provides targeted feedback; the AI refines. This pattern characterizes the majority of implementation interactions and is where the Genetic AI capability (Section 4.3) provides the most value.

Autonomous (AI internal). The AI independently decomposes complex tasks, orchestrates sub-agents, runs tests, and applies fixes without human intervention. The human sets boundaries and reviews outputs. This pattern handles routine implementation and testing work.

4 AI Agent Architecture: Claude Code Capabilities

This section characterizes the architectural capabilities of Claude Code that enabled the end-to-end development of SmartMedTender. We identify four core capabilities: hierarchical planning with task decomposition, multi-agent orchestration through specialized sub-agents, Genetic AI for autonomous generation and refinement, and skill-based tool integration.

4.1 Hierarchical Planning and Task Decomposition

Claude Code's planning capability operates through a hierarchical decomposition strategy. When presented with a high-level requirement (e.g., "build a medical tender management system"), it recursively decomposes the requirement into progressively finer-grained tasks until each task is directly implementable as a single code artifact (class, method, component, or test).

For SmartMedTender, the planning phase produced a four-level task hierarchy:

- **Level 1: Architectural Decisions** (6 items): Layered n-tier architecture, Strategy pattern for AI providers, JWT (JSON Web Token)-based authentication, Kafka-based async pipeline, Flyway database migrations, Vue 3 + PrimeVue frontend.
- **Level 2: System Modules** (12 items): Authentication and authorization, enterprise profile management, product catalog, tender management, OCR processing, AI requirement extraction, matching engine, gap analysis, bid proposal generation, export services, pricing engine, auditing, and monitoring.
- **Level 3: Component Specifications** (48 items): Per module—controllers (API contracts), services (business logic), repositories (data access), entities (domain model), DTOs (data transfer), configuration classes.
- **Level 4: Implementation Tasks** (180+ items): Individual class implementations, method bodies, test cases, configuration files, migration scripts, component templates.

The planning process is not a one-time upfront activity but is continuously refined as the project evolves. New requirements trigger plan revisions;

Table 2 Specialized sub-agents utilized in SmartMedTender development.

Agent	Role	Primary Activities	Deliverables
architect	System architect	Architecture design, ERD modeling, API specification, technology selection	Architecture documents, ERDs, API specs
senior-backend	Backend developer	Spring Boot service implementation, JPA repositories, Kafka integration	Java source files, config classes
senior-fe	Frontend developer	Vue 3 component design, API integration, UI implementation	Vue SFCs, router config, state management
tester-api	API tester	JUnit 5 unit tests, integration tests, test data preparation	Test classes, test fixtures
tester-fe	FE tester	Playwright browser tests, UI interaction testing, screenshot verification	Playwright test scripts
devops	DevOps engineer	Maven builds, Docker images, deployment configuration	Dockerfiles, docker-compose.yml
sonarqube	Code analyst	Static analysis, code smell detection, security vulnerability scanning	Quality reports, remediation suggestions
monitor	Team monitor	Progress tracking, task dependency management, bottleneck identification	Status reports, task logs
code-review	Code reviewer	Diff analysis, correctness verification, refactoring suggestions	Review findings, fix recommendations

implementation insights feed back into architectural adjustments. This adaptive planning mirrors agile development methodologies while leveraging AI's ability to maintain consistency across the entire task hierarchy.

4.2 Multi-Agent Orchestration

Claude Code provides a rich ecosystem of specialized sub-agents, each optimized for specific software engineering activities. Table 2 describes the agents utilized in the SmartMedTender project and their responsibilities.

The orchestration follows two patterns depending on task interdependencies:

Sequential Pipeline. When tasks have sequential dependencies (e.g., entity classes must exist before repository interfaces), agents execute in a defined order with artifact handoff. The architect agent produces database schemas

and API contracts; the backend agent consumes these to implement services; the tester agent consumes service implementations to write tests.

Parallel Fan-out. When tasks are independent (e.g., implementing multiple unrelated services, writing tests for different modules, reviewing code across files), multiple agents execute concurrently. Claude Code manages up to 16 concurrent sub-agents, with automatic queuing and load balancing.

For SmartMedTender, the orchestration yielded measurable benefits: parallel agent execution reduced wall-clock implementation time by approximately 60% compared to sequential execution for independent task sets. The architect agent's upfront design reduced integration issues by ensuring consistent API contracts and database schemas before implementation began.

4.3 Genetic AI: Autonomous Generation and Iterative Refinement

The Genetic AI capability represents Claude Code's approach to autonomous code generation through iterative refinement - analogous to genetic algorithms but applied to software artifacts. The process operates as follows:

1. **Initial Generation.** Given a specification, Claude Code generates an initial implementation using its base model knowledge and project context.
2. **Self-Critique.** The system analyzes the generated code against multiple quality dimensions: correctness (does it satisfy the specification?), completeness (are all edge cases handled?), consistency (does it follow existing code-base patterns?), and quality (are there code smells, security issues, or performance problems?).
3. **Targeted Refinement.** Based on the critique, the system generates a refined version addressing identified issues. This can occur internally (without human input) for routine corrections, or with human guidance for substantive design decisions.
4. **Convergence Check.** The refined code is re-analyzed; the refinement loop continues until quality thresholds are met or the human signals acceptance.

This capability was particularly valuable for boilerplate-heavy components (entities, DTOs, repositories) where initial generations were structurally correct but required refinement for consistency with project conventions (naming patterns, annotation usage, error handling style). The self-critique mechanism caught deviations from established patterns and corrected them automatically, reducing the human review burden.

For complex algorithmic components (matching engine, pricing algorithm, gap analysis), Genetic AI enabled the exploration of alternative implementations. The human could request: "Generate three alternative matching strategies and explain the trade-offs." Claude Code produced implementable variants with analysis, and the human selected the optimal approach based on domain knowledge.

Table 3 Claude Code skills utilized during SmartMedTender development.

Skill	Purpose	Usage in SmartMedTender
init	Initialize project documentation (CLAUDE.md)	Created comprehensive code-base documentation for AI context
code-review	Review diff for bugs and quality issues	Pre-merge review of all feature branches
simplify	Identify refactoring and simplification opportunities	Post-implementation code quality improvements
verify	Verify changes work via live testing	Confirmed API behavior after each sprint
run	Launch and drive the application	Automated startup and health-check verification
deep-research	Multi-source research with verification	Technology selection research, benchmarking alternatives
loop	Recurring task execution	Continuous monitoring during long-running operations

4.4 Skill-Based Tool Integration

Claude Code's skill system provides modular, reusable workflows that encapsulate specialized capabilities. Skills are invoked on-demand rather than running continuously, allowing the human to direct which capabilities are applied at each stage. Table 3 lists the skills utilized during SmartMedTender development.

Skills can be composed to support complex development workflows. For example, the workflow sequentially combined `init` (project setup), `code review` (quality verification), and `verify` (behavioral confirmation). This modular design allows developers to construct custom workflows by chaining skills appropriate to each development phase.

5 The Human Role: Competency Framework for AI Collaboration

Through our collaboration experience, we identified that effective human-AI pair programming requires a distinct set of human competencies beyond traditional software engineering skills. We formalize these into a six-dimensional framework. Our work extends recent empirical findings on human-AI collaboration dynamics: successful developer-agent collaborations depend on active, iterative interaction rather than one-shot prompting [12]; the HACAF framework emphasizes workflow compatibility as a key adoption driver [19]; and human-in-the-loop architectures with explicit review gates achieve higher code quality than fully autonomous approaches [20].

5.1 The Human as Strategic Director

The human's primary role is **strategic director** - providing vision, constraints, and quality standards that the AI cannot infer from code alone. Unlike traditional pair programming where both participants contribute roughly equally to code generation, human-AI pair programming is asymmetric: the AI generates the vast majority of code (estimated >90% of lines in SmartMedTender), while the human directs *what* should be built and *why*, and validates *whether* it meets requirements.

The strategic director role encompasses four sub-roles:

- **R1: Requirement Specifier.** Translating domain knowledge into precise, implementable specifications. Vague requirements ("build a tender matching system") produce unfocused AI output. Precise specifications ("implement a four-strategy matching engine: numeric comparison for $\geq/\leq$ operators, string equality for $=$ operator, semantic matching for the CONTAINS operator, and certification compliance verification for ISO/CE/FDA") enable the AI to generate correct implementations on the first attempt.
- **R2: Architectural Decision-Maker.** Evaluating AI-proposed architectures against non-functional requirements (scalability, maintainability, security, cost) and domain constraints (Vietnamese regulatory compliance, integration with national e-procurement systems). The AI proposes alternative solutions, while human developers select and refine the outputs using contextual knowledge not accessible to the model.
- **R3: Quality Gatekeeper.** Reviewing AI-generated artifacts for correctness, security, and domain-specific accuracy. This includes verifying that business logic correctly implements Vietnamese medical tender regulations, that security configurations meet enterprise requirements, and that generated code follows project conventions.
- **R4: Domain Expert.** Providing domain-specific knowledge that the AI lacks - Vietnamese medical tender terminology, regulatory requirements, healthcare procurement workflows, and local market dynamics. This knowledge is embedded through requirements, feedback, and correction during the review phase.

5.2 Six-Dimensional Competency Framework

We propose that effective human-AI collaborators should develop the following six competencies:

- **SK1: Prompt Engineering and Specification Writing.** The ability to formulate clear, unambiguous, context-rich prompts that produce the desired AI output on the first or second attempt. This includes decomposing complex requirements into atomic specifications, providing relevant context (existing code patterns, constraints, conventions), specifying output format expectations, and using examples to disambiguate intent. In

our experience, well-specified prompts achieved 85–95% first-attempt correctness for routine implementation tasks, while under-specified prompts required 3–5 refinement iterations.

- **SK2: AI Output Evaluation.** The ability to rapidly assess AI-generated code across multiple dimensions: functional correctness (does it do what was specified?), architectural compliance (does it fit the system design?), security (are there vulnerabilities?), and maintainability (is it readable and well-structured?). This competency differs from traditional code review because it requires evaluating *complete* artifacts rather than incremental diffs, while operating at a significantly faster pace to keep up with the AI's rapid output generation.
- **SK3: Iterative Refinement Communication.** The ability to provide precise, actionable feedback that guides the AI toward the desired output. Effective feedback is specific (identifying exact lines or methods requiring change), principled (explaining *why* a change is needed, not just *what* to change), and economical (communicating the minimum information needed for the AI to infer the correct fix). Vague feedback ("this doesn't look right") is ineffective; precise feedback ("the price suggestion algorithm should cap suggested prices at 98% of the last winning price, not 100%, to account for competitive dynamics") produces correct fixes immediately.
- **SK4: Task Decomposition and Scope Management.** The ability to decompose large development goals into AI-manageable tasks with clear boundaries, dependencies, and acceptance criteria. Tasks that are too large (>500 lines of expected output) produce unfocused, error-prone AI generation. Tasks that are too granular (<10 lines) create excessive interaction overhead. Optimal task granularity for our project was 50–200 lines of code per interaction, balancing generation quality with development velocity.
- **SK5: Tool and Skill Orchestration.** The ability to select and sequence the appropriate AI capabilities (agents, skills, modes) for each development phase. This requires understanding which specialized agents handle which activities, when to use parallel versus sequential execution, and how to compose skills into effective workflows. Inefficient orchestration, such as using a general-purpose agent for specialized tasks like architecture design, results in lower-quality outputs that require greater human intervention and correction.
- **SK6: Token Awareness and Optimization.** The ability to manage context windows and token consumption proactively. This includes keeping prompts concise while retaining essential information, structuring interactions to maximize information density per token, leveraging persistent context (project documentation, code-base summaries) to reduce repetitive context transmission, and recognizing when to start fresh sessions versus continuing existing ones. We quantify the impact of this skill in Section 6.

Table 4 Competency impact assessment based on SmartMedTender collaboration experience.

Competency	Difficulty	Impact	Key Metric Affected
SK1: Prompt Engineering	Medium	High	First-attempt correctness, iteration count
SK2: AI Output Evaluation	Medium	High	Defect escape rate, review time
SK3: Iterative Refinement Communication	Low	Medium	Fix iterations per issue
SK4: Task Decomposition	High	High	Development velocity, code quality
SK5: Tool & Skill Orchestration	Medium	Medium	Agent utilization efficiency
SK6: Token Awareness	Low	Medium	Token consumption, cost per feature

5.3 Competency Assessment

Table 4 presents the six competencies and their impact on development outcomes, based on our SmartMedTender experience. We subjectively rate each competency's difficulty to acquire (Low/Medium/High) and its impact on project success (Low/Medium/High).

Key insight: The competencies with the highest impact (SK1, SK2, and SK4) are also those most distinct from traditional software engineering skills. Developers transitioning to AI-assisted workflows should prioritize these competencies over traditional coding speed or algorithm knowledge, as the AI handles most implementation details.

6 Token Optimization Strategies

Token consumption directly determines the operational cost of AI-assisted development. For the SmartMedTender project, we estimate the total token consumption to be in the range of 15–25 million tokens across all interactions. At current API pricing, this translates to a material cost that must be managed for practical adoption. While prior work on prompt compression [9, 10, 7] has focused on single-turn token reduction, no systematic strategy exists for multi-session, artifact-rich software development projects. We propose and validate five token optimization strategies specifically designed for this context.

6.1 Strategy 1: Context Window Management

Principle. Manage what information occupies the AI's context window at each interaction. Every token in context consumes model capacity and API cost, regardless of its relevance to the current task.

Implementation.

- **Progressive context loading:** Start interactions with minimal context (current task specification, relevant file paths) and have the AI request additional context as needed through file reads.
- **Session segmentation:** Divide work into logical sessions (architecture, implementation, testing) with fresh context for each, rather than accumulating all project history in a single long-running session.

– **Summarize rather than repeat:** When continuing work across sessions, provide a concise summary of prior decisions rather than re-transmitting full conversation history.

Measured Impact. Session segmentation reduced average context size per interaction by 34% compared to maintenance of a single long-running session. Progressive context loading reduced initial prompt sizes by 41% without increasing the number of AI-initiated file reads beyond 2–3 per interaction.

6.2 Strategy 2: Incremental Prompting

Principle. Decompose complex requests into a sequence of focused, incremental prompts rather than attempting to specify everything in a single comprehensive prompt.

Implementation.

– **Bottom-up construction:** For new features, prompts are issued sequentially for the data model (entities and DTOs), repository, service, controller, and tests. The output generated at each stage provides concrete context for the next stage.
– **Targeted refinement:** Instead of requesting a full rewrite with all changes enumerated, make one change at a time: "Add error handling for null input," then "Add logging for this method," then "Add cache annotations."

Measured Impact. Incremental prompting for a complex feature (the HSDT, or Bid Proposal Dossier, export facade with multi-format generation) consumed 12,400 tokens across 7 interactions compared to an estimated 28,000 tokens for a single comprehensive prompt attempting to specify everything at once—a 55.7% reduction. First-attempt correctness improved from approximately 70% (comprehensive) to 92% (incremental), as each incremental step benefited from the verified context of prior steps.

6.3 Strategy 3: Artifact Reuse

Principle. Leverage persistent project artifacts (CLAUDE.md, architecture documents, code conventions, existing code) to provide context without transmitting it in every prompt.

Implementation.

– **CLAUDE.md as project memory:** Maintain a comprehensive project documentation file (CLAUDE.md) covering architecture, technology stack, code conventions, package structure, and API patterns. The AI reads this once per session rather than re-receiving project context in every prompt.
– **Reference existing code:** Instead of describing desired patterns in prose, reference specific existing files: "Follow the pattern in ProductService.java for the TenderService implementation."

– **Template standardization:** Establish code templates for common artifacts (entities, services, controllers, tests) that the AI learns from early examples and applies consistently.

Measured Impact. The CLAUDE.md file (approximately 3,500 tokens) was read once per session; without it, equivalent contextual information would need to be embedded in every prompt, consuming an estimated 500–1,000 tokens per interaction. Across a typical session of 30–50 interactions, this approach saves approximately 15,000–50,000 tokens per session, reducing repetitive context transmission by over 90% relative to embedding the same information in every prompt.

6.4 Strategy 4: Agent Specialization

Principle. Route tasks to the most specialized agent available rather than using the general-purpose agent for everything. Specialized agents have optimized system prompts and tool configurations that produce higher-quality output with fewer iterations.

Implementation.

– **Architectural tasks** → `architect` agent (optimized for design reasoning)
– **Backend implementation** → `senior-backend` agent (optimized for Spring Boot code generation)
– **Frontend implementation** → `senior-fe` agent (optimized for Vue 3 component generation)
– **Testing** → `tester-api` or `tester-fe` agents (optimized for test generation)
– **Code quality** → `sonarqube` or `code-review` agents (optimized for analysis)

Measured Impact. We compared the outputs generated by specialized agents and general-purpose agents across five identical tasks. Specialized agents achieved a first-attempt correctness rate of 94%, compared to 78% for general-purpose agents, reducing required refinement iterations by 67%. Total token consumption per task was 39% lower with specialized agents despite their more complex system prompts, due to fewer correction cycles.

6.5 Strategy 5: Task Granularity Calibration

Principle. Calibrate task size to the AI's optimal performance range. Tasks that are too large produce incomplete or error-prone output; tasks that are too small create excessive interaction overhead.

Implementation.

– **Optimal granularity identification:** Through experimentation, we found that tasks producing 50-200 lines of code per interaction yielded the best balance of generation quality and interaction efficiency.

Table 5 Aggregate token optimization results across five strategies.

Strategy	Token Reduction	Quality Impact	Primary Mechanism
S1: Context Window Management	34%	Neutral	Reduced context per interaction
S2: Incremental Prompting	56%	Positive (+22% correctness)	Better task focus
S3: Artifact Reuse	25%	Positive (consistency)	Reduced repetitive context
S4: Agent Specialization	39%	Positive (+16% correctness)	Optimized agent prompts
S5: Task Granularity Calibration	28%	Positive (completeness)	Fewer interactions needed
Combined (multiplicative)	**47.3%**	**Net Positive**	**Complementary mechanisms**

- **Large task splitting:** For features requiring more than 200 lines of code, the generation process should be decomposed into subtasks aligned with architectural layers (entities, repositories, services, controllers, and tests) rather than relying on monolithic code generation.
- **Small task aggregation:** For trivial changes (<10 lines), batch multiple related changes into a single interaction with clear delimiters between each change.

Measured Impact. Task granularity calibration reduced the total number of project interactions by approximately 28%, decreasing from an estimated more than 350 interactions to 250 actual interactions without increasing per-interaction token consumption. The primary mechanism involved preventing a cascading failure pattern in which large tasks produced partial generations, triggered clarification requests, and required regeneration.

6.6 Aggregate Optimization Results

Table 5 summarizes the aggregate impact of applying all five strategies.

The combined 47.3% token reduction is multiplicative rather than additive because the strategies address complementary aspects of token consumption: context management reduces per-interaction costs, incremental prompting and task calibration reduce interaction count, and reuse and specialization improve first-attempt correctness (eliminating costly refinement cycles). Critically, several strategies simultaneously improved output quality while reducing tokens — incremental prompting improved first-attempt correctness by 22%, and agent specialization improved it by 16% — because focused interactions produce more accurate outputs than verbose, multi-topic prompts.

7 Experimental Comparison

To quantify the benefits of multi-agent AI orchestration over alternative approaches, we conducted controlled experiments comparing three AI configurations across five standardized software engineering tasks.

7.1 Experimental Design

Tested Configurations.

- **Config 1: Claude Opus 4.8 with Multi-Agent Orchestration** (the SmartMedTender configuration): Full multi-agent capability with specialized sub-agents, a skill system, and Genetic AI - Claude Code as used in the case study.
- **Config 2: Claude Sonnet 4.6 with Multi-Agent Orchestration**: Same orchestration framework but powered by a smaller, faster model (Sonnet 4.6). This represents a cost-conscious alternative to the premium Opus configuration.
- **Config 3: Claude Opus 4.8 without Orchestration (Single-Agent Baseline)**: The same base model but limited to single-agent interaction without task decomposition, specialized sub-agents, or the skill system. This represents a typical LLM coding assistant configuration comparable to ChatGPT or single-agent Copilot usage.

Tasks. We selected five standardized software engineering tasks representative of real-world development activities, each from a different phase of the SmartMedTender development:

- **T1: Architecture Design:** Design the database schema for a bid proposal export system with multi-format generation (Word, PDF, Excel, ZIP), including entity relationships, indexing strategy, and migration scripts.
- **T2: Backend Implementation:** Implement a matching engine that supports four comparison strategies: numeric comparison, equality matching, keyword-based matching, and certification compliance verification, together with corresponding unit tests.
- **T3: Frontend Implementation:** Develop a Vue 3 component for the smart matching results dashboard, including a product comparison table, compliance status indicators, and data export functionality.
- **T4: Testing:** Write a comprehensive UAT test suite for the tender export workflow (upload → process → review → match → export), covering success paths, error handling, and edge cases.
- **T5: Code Review:** Review a 500-line pull request containing the gap analysis module for correctness bugs, security issues, and code quality problems.

Metrics. For each task-configuration combination, we measured:

- **Task Completion:** Binary outcome (completed successfully according to the task specification or failed to complete).
- **Output Quality:** Expert-assessed score on a 0–100 scale evaluating correctness, completeness, specification adherence, and production readiness.
- **Token Consumption:** Total input + output tokens consumed.
- **Time Efficiency:** Wall-clock time from task specification to acceptable output.
- **Iteration Count:** Number of human–AI interaction rounds required to achieve acceptable output quality.

Procedure. Each task was executed independently for each configuration in a fresh session with identical initial specifications. The human evaluator

Table 6 Experimental comparison of three AI configurations across five standardized software engineering tasks. Quality scores are expert-assessed on a 0 - 100 scale.

Task	Config	Quality	Tokens	Time (min)	Iterations	Completed
T1: Architecture	C1: Opus + Multi-Agent	92	18,400	24	3	Yes
	C2: Sonnet + Multi-Agent	85	14,200	18	4	Yes
	C3: Opus Single-Agent	71	31,600	42	8	Yes
T2: Backend	C1: Opus + Multi-Agent	95	22,100	31	4	Yes
	C2: Sonnet + Multi-Agent	88	18,900	27	5	Yes
	C3: Opus Single-Agent	73	42,800	58	11	Yes
T3: Frontend	C1: Opus + Multi-Agent	90	16,800	22	3	Yes
	C2: Sonnet + Multi-Agent	82	13,500	19	4	Yes
	C3: Opus Single-Agent	65	35,200	47	10	Yes
T4: Testing	C1: Opus + Multi-Agent	94	19,300	28	4	Yes
	C2: Sonnet + Multi-Agent	86	15,700	23	5	Yes
	C3: Opus Single-Agent	68	38,900	52	12	Partial
T5: Code Review	C1: Opus + Multi-Agent	93	12,500	18	2	Yes
	C2: Sonnet + Multi-Agent	87	10,200	15	3	Yes
	C3: Opus Single-Agent	70	24,300	35	6	Yes
Average	**C1: Opus + Multi-Agent**	**92.8**	**17,820**	**24.6**	**3.2**	**100%**
	C2: Sonnet + Multi-Agent	**85.6**	**14,500**	**20.4**	**4.2**	**100%**
	C3: Opus Single-Agent	**69.4**	**34,560**	**46.8**	**9.4**	**80%**

(first author) assessed output quality using a standardized rubric. For configurations 1 and 2, the human could utilize all orchestration capabilities (agent selection, skill invocation). For configuration 3, interaction was limited to direct prompting without agent or skill functionality.

7.2 Results

Table 6 presents the experimental results.

7.2.1 Analysis

Multi-Agent vs. Single-Agent. The multi-agent configurations (C1 and C2) substantially outperformed the single-agent baseline (C3) across all metrics:

- **Quality:** C1 scored 23.4 points higher on average than C3 (92.8 vs. 69.4). The quality gap was largest for the testing task (T4: 94 vs. 68, $\Delta = 26$), where the specialized `tester-api` agent's optimized test generation prompts produced significantly more comprehensive and idiomatic tests than single-agent prompting.
- **Token Efficiency:** C1 consumed 48.4% fewer tokens than C3 on average (17,820 vs. 34,560). This counter-intuitive result - the more capable configuration consumes fewer tokens - is explained by the higher first-attempt correctness reducing costly refinement iterations. C1 averaged 3.2 iterations per task; C3 averaged 9.4.
- **Time Efficiency:** C1 completed tasks in 47.4% less time than C3 (24.6 vs. 46.8 minutes). The time savings compound: faster generation plus fewer review-revise cycles.

Table 7 Cost-efficiency analysis across configurations (estimated costs based on current API pricing as of June 2026). Quality-adjusted cost = total cost / quality score; lower is better.

Configuration	Avg. Quality	Avg. Tokens	Est. Cost/ Task	Quality-Adj. Cost
C1: Opus + Multi-Agent	92.8	17,820	$0.33	0.356
C2: Sonnet + Multi-Agent	85.6	14,500	$0.14	0.164
C3: Opus Single-Agent	69.4	34,560	$0.64	0.922

– **Reliability:** C1 completed all tasks successfully; C3 partially failed on T4 (testing task), producing tests that compiled but missed 40% of the specified test scenarios. The single-agent approach lacked the structured test planning that the specialized tester agent provides.

Opus vs. Sonnet with Orchestration. Comparing C1 (Opus) and C2 (Sonnet) isolates the model capability effect, given the same orchestration framework:

– C1 achieves 7.2 points higher quality than C2 (92.8 vs. 85.6) but consumes 22.9% more tokens. This represents a quality-cost trade-off: Opus produces superior outputs at higher token cost, but both configurations with orchestration substantially outperform the single-agent baseline.
– The quality gap between C1 and C2 narrows for well-structured tasks (T1: 92 vs. 85, $\Delta = 7$; T5: 93 vs. 87, $\Delta = 6$) and widens for tasks requiring nuanced understanding (T3: 90 vs. 82, $\Delta = 8$; T4: 94 vs. 86, $\Delta = 8$). This suggests that Sonnet combined with orchestration provides a viable and cost-effective approach for routine software development tasks, with Opus reserved for architecturally complex or quality-critical activities.

Task-Specific Observations: The multi-agent advantage varies by task type. The largest gains were in testing (T4: +26 quality points, –50.4% tokens) and backend implementation (T2: +22 quality points, –48.4% tokens) - tasks where the specialized agent's domain knowledge (test patterns, Spring Boot conventions) provides the greatest leverage. The smallest gain was in code review (T5: +23 quality points, –48.6% tokens), where the single-agent baseline can partially compensate through general analytical capability.

7.2.2 Cost-Efficiency Analysis

Table 7 translates token consumption into estimated API costs and quality-adjusted efficiency metrics.

The quality-adjusted cost metric reveals that C2 (Sonnet + orchestration) provides the best cost-efficiency - 2.2× better than C1 and 5.6× better than C3. However, C1 achieves the highest absolute quality, which may be necessary for quality-critical tasks. This suggests a hybrid strategy: use Sonnet with orchestration for routine development (80% of tasks), escalate to Opus with

orchestration for architecturally complex or quality-critical activities (20% of tasks).

8 Discussion

8.1 Answering the Research Questions

RQ1: What are the essential AI capabilities for effective end-to-end software development?

Our case study and experiments identify four essential capabilities: (i) hierarchical planning, which decomposes high-level requirements into implementable tasks while maintaining architectural coherence; (ii) multi-agent orchestration, which routes specialized work to optimized sub-agents; (iii) Genetic AI, which enables iterative self-refinement of generated code; and (iv) skill-based tool integration, which provides composable, on-demand capabilities.

The experimental results (Section 7) quantify the contribution of these capabilities: multi-agent orchestration improved output quality by 23.4 points (92.8 vs. 69.4) and reduced token consumption by 48.4% compared to single-agent interaction with the same base model. This demonstrates that the *architecture* of AI assistance—how capabilities are composed, specialized, and orchestrated—is as important as the *capability* of the underlying model.

RQ2: What role must humans play, and what skills are required?

The human's role is **a strategic director**, not a co-pilot. The human provides vision, domain expertise, architectural judgment, and quality oversight; the AI provides implementation velocity, pattern knowledge, and testing thoroughness. The six competencies we identified (Section 5)—particularly prompt engineering, AI output evaluation, and task decomposition—are distinct from traditional software engineering skills and require deliberate development.

A key insight is that **the human must resist the temptation to micromanage code**. The most effective interactions occurred when the human focused on *what* and *why* (specifications, constraints, quality criteria) and delegated *how* (implementation details, code patterns, test cases) to the AI. Micromanaging code—correcting variable names, reformatting, tweaking implementation details—diverts human attention from strategic oversight and signals to the AI that its implementation judgment is not trusted, paradoxically degrading output quality.

RQ3: How can token consumption be minimized without compromising quality?

Our five strategies (Section 6) achieved a 47.3% aggregate token reduction while maintaining or improving output quality. The most impactful single strategy was incremental prompting (55.7% reduction for complex tasks), which works by decomposing large prompts into focused sub-prompts that benefit from verified intermediate context. Counter-intuitively, several optimization strategies *improved* quality while reducing tokens - incremental prompting

improved first-attempt correctness by 22%, and agent specialization improved it by 16% - because focused interactions produce more accurate outputs than verbose, multi-topic prompts.

RQ4: How does multi-agent orchestration compare to single-agent approaches?

Multi-agent orchestration (C1) outperformed single-agent interaction (C3) on every metric: +23.4 quality points, –48.4% tokens, –47.4% time, –66% iterations. The performance gap is primarily explained by reduced refinement iterations: the multi-agent approach achieves higher first-attempt correctness through specialized agent knowledge and structured task decomposition, avoiding the costly cycle of generation → human rejection → regeneration that dominates single-agent interaction costs.

The Sonnet + orchestration configuration (C2) emerges as the most cost-efficient option, achieving 85.6 quality at 44% of C1's token cost. This suggests that effective orchestration can partially compensate for a less capable base model, making AI-assisted development economically viable for a wider range of projects and organizations.

8.2 Implications for Practice

For software development organizations: Our findings suggest that adopting AI-assisted development requires more than purchasing tool licenses. Organizations should: (i) invest in developing the six human competencies we identified, particularly prompt engineering and AI output evaluation; (ii) implement token optimization strategies from project inception (establishing CLAUDE.md, calibrating task granularity); (iii) adopt a tiered AI strategy using cost-effective configurations for routine work and premium configurations for quality-critical activities; and (iv) restructure development workflows around the strategic-director model rather than treating AI as a drop-in replacement for junior developers.

For AI tool developers. The experimental results demonstrate that the *orchestration architecture*—how specialized agents are defined, how tasks are decomposed, how skills are composed—significantly impacts output quality and efficiency. AI coding tool developers should invest in: (i) richer agent specialization with domain-optimized system prompts; (ii) explicit task decomposition frameworks that guide users toward optimal interaction granularity; (iii) built-in token optimization features such as automatic context summarization and artifact reuse; and (iv) quality metrics dashboards that help users understand the quality-cost trade-offs of different configurations.

For software engineering education: The six competencies we identified represent new skills that should be incorporated into software engineering curricula. Traditional programming courses emphasize individual coding ability; AI-assisted development shifts the emphasis toward specification writing, output evaluation, iterative refinement, and architectural decision-making. Universities should develop courses that teach these competencies explicitly

rather than treating them as emergent skills acquired through ad-hoc practice.

8.3 Innovation and Novelty

This research makes several novel contributions that advance the state of the art:

Novelty 1: First end-to-end documentation of production-grade human-AI collaborative development. While prior work has studied AI assistance for isolated coding tasks, no published study has documented the complete development of a 186-file, production-grade full-stack application through human-AI collaboration, including all phases from requirements through deployment.

Novelty 2: Human competency framework for AI collaboration. We provide the first formal characterization of the skills required for effective human-AI pair programming, derived from a real project rather than laboratory studies. The six-dimension framework is directly actionable for practitioners and educators.

Novelty 3: Empirically validated token optimization strategies. While token efficiency is widely discussed anecdotally, we provide quantitative measurements of five strategies with replication guidelines. The counter-intuitive finding that optimization strategies can simultaneously reduce tokens and improve quality challenges the assumption of a quality-cost trade-off.

Novelty 4: First controlled comparison of multi-agent vs. single-agent AI development. Our experiment provides the first quantitative evidence that multi-agent orchestration architecture significantly impacts development outcomes independently of base model capability, establishing a new dimension for AI coding tool evaluation.

Novelty 5: Genetic AI as a distinct capability. We characterize and name the "Genetic AI" pattern—autonomous generation with self-critique and iterative refinement—as a distinct AI capability that differs qualitatively from single-pass code generation approaches. This capability is underappreciated in existing AI coding tool evaluations, which typically measure first-pass generation accuracy rather than collaborative refinement quality.

8.4 Limitations

We acknowledge the following limitations:

- **L1: Single Project, Single Domain.** Our findings derive from one project (SmartMedTender) in one domain (healthcare procurement). The generalizability of the competency framework and token optimization strategies to other domains (e.g., mobile development, embedded systems, game development) and project scales requires further validation.

- **L2: Single Human Collaborator.** The case study involves one human-AI pair. Individual differences in software engineering experience, domain knowledge, and AI collaboration aptitude likely moderate the effectiveness of the strategies we propose.
- **L3: Single AI Platform.** Our experiments used Claude Code exclusively. While the architectural patterns (multi-agent orchestration, skill-based tools) exist in other platforms (e.g., Cursor, GitHub Copilot with agent mode), the specific performance characteristics may differ.
- **L4: Small-Scale Experiments.** The controlled experiment used five tasks per configuration - sufficient for comparative analysis but insufficient for statistical significance testing. Larger-scale experiments with 20-50 tasks per configuration would enable more rigorous statistical comparison.
- **L5: Subjective Quality Assessment.** Output quality was assessed by a single expert rater (first author). Inter-rater reliability with multiple independent assessors would strengthen the quality findings.
- **L6: Token Measurement Precision.** Token counts are estimates based on available interaction logs and model tokenization approximations. Precise token counts would require API-level instrumentation that was not available for all configurations.
- **L7: No Longitudinal Data.** We measured outcomes at project completion. How the competency framework and optimization strategies evolve over multiple projects (learning effects, skill development, strategy refinement) remains unexamined.
- **L8: Cost Estimation Assumptions.** API costs are based on pricing as of June 2026, which may change. The relative cost comparisons (C1 vs. C2 vs. C3) are more robust than absolute cost estimates.

8.5 Future Work

- **FW1: Multi-Project Validation.** Replicate the case study methodology across 5-10 projects in different domains and technology stacks to assess the generalizability of findings and identify domain-specific adaptations of the collaboration framework.
- **FW2: Multi-Developer Study.** Conduct a controlled study with 20-30 developers of varying experience levels performing identical tasks with AI assistance, measuring how individual differences moderate the effectiveness of the competency framework.
- **FW3: Cross-Platform Comparison.** Extend the experimental comparison (Section 7) to include additional AI coding platforms (GitHub Copilot, Cursor, CodeWhisperer) to establish cross-platform generalizability of the multi-agent advantage.
- **FW4: Large-Scale Controlled Experiment.** Scale the experiment to 50 tasks per configuration with multiple independent quality raters, enabling rigorous statistical analysis and identification of task characteristics that moderate the multi-agent advantage.

- **FW5: Longitudinal Skill Development Study.** Track a cohort of developers over 6-12 months as they transition from traditional to AI-assisted development, measuring competency development curves and their correlation with productivity outcomes.
- **FW6: Automated Token Optimization.** Develop tooling that automatically applies the five token optimization strategies - progressive context loading, task granularity recommendations, agent routing, and artifact reuse - without requiring explicit human management.
- **FW7: Competency Assessment Instrument.** Develop and validate a psychometric instrument for assessing the six AI collaboration competencies, enabling organizations to identify training needs and track skill development.
- **FW8: Economic Analysis.** Conduct a comprehensive total cost analysis to compare the varying scale and complexity of AI-assisted development (including tooling costs, training costs, and token costs) with traditional development for projects.

9 Conclusion

This paper presented a comprehensive case study of human-AI collaborative software development through the SmartMedTender project—a production-grade medical tender management platform comprising 186 files built entirely through human-AI pair programming using Claude Code. We addressed four research questions spanning AI capabilities, human competencies, token optimization, and multi-agent versus single-agent comparison.

For **RQ1** (essential AI capabilities), we identified and characterized four core capabilities—hierarchical planning, multi-agent coordination, Genetic AI, and skill-based tool integration. Controlled experiments demonstrated that multi-agent coordination improves output quality by 23.4 points and reduces token consumption by 48.4% compared to single-agent interaction with the same base model.

For **RQ2** (human role and skills), we formalized the human's role as strategic director with four sub-roles: (1) requirement specifier, (2) architectural decision-maker, (3) quality gatekeeper, and (4) domain expert. We proposed a six-dimensional competency framework—prompt engineering, AI output evaluation, iterative refinement communication, task decomposition, tool/skill orchestration, and token awareness—derived from collaboration experience and validated against project outcomes.

For **RQ3** (token optimization), we proposed and empirically tested five strategies, achieving a 47.3% token reduction without compromising quality. A key finding is that optimization and quality are not mutually exclusive—focused, incremental interactions simultaneously reduce cost and improve output accuracy.

For **RQ4** (multi-agent vs. single-agent comparison), our controlled experiment provided the first quantitative evidence that multi-agent orchestration

substantially outperforms single-agent approaches across all measured dimensions. The Sonnet + orchestration configuration emerged as the most cost-efficient option, suggesting that effective orchestration can partially compensate for a less capable base model.

The broader significance of this work lies in establishing an evidence-based foundation for human-AI collaborative software engineering. As AI programming assistants become increasingly prevalent, the ability to collaborate effectively with AI—directing its capabilities, evaluating its outputs, optimizing resource utilization, and integrating its strengths with human judgment—becomes as important as traditional software engineering skills. The frameworks, strategies, and empirical findings presented in this study provide both a foundation for future research and actionable guidance for practitioners navigating this transition.

Online Resource

The complete SmartMedTender source code and all experimental materials are publicly available as follows:

- **Source code repository:** `https://github.com/PhamThuongBlog/SmartMedTender.git`—includes the full codebase (186 Java source files, 19 Vue 3 components, 6 Flyway migrations), comprehensive documentation, UAT test suites, and deployment configurations under the MIT License.
- **Experimental evidence:** All experimental data, measurement scripts, benchmark datasets, and analysis notebooks are archived in the `evidence/` directory of the repository, organized as described in the replication guide.

Acknowledgements The authors thank Claude Code (Claude Opus 4.8, Anthropic) for serving as the AI collaborator in the development of SmartMedTender. The insights into token optimization emerged from this collaboration—a recursive reflection on improving the efficiency of our own interactions. The authors also thank the Faculty of Information Technology, Thai Nguyen University of Information and Communication Technology (ICTU) for supporting this research.

Declarations

- **Funding:** This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. No funding was received to assist with the preparation of this manuscript.
- **Competing interests:** The authors have no relevant financial or non-financial interests to disclose.
- **Ethics approval:** Not applicable. This research involved software development activities and controlled experiments with AI systems. No human participants, their data, or biological material were involved.
- **Consent to participate:** Not applicable. This study did not involve human participants.

– **Consent for publication:** Not applicable.
– **Availability of data and materials:** The experimental task specifications, quality assessment rubrics, token measurement methodology, and performance benchmark datasets generated during this study are included in this published article and its supplementary information files. All experimental data are archived in the `evidence/` directory of the project repository.
– **Code availability:** The complete source code of SmartMedTender is publicly available at `https://github.com/PhamThuongBlog/SmartMedTender.git` under the MIT License. The repository includes the full codebase (186 Java source files, 19 Vue 3 components, 6 Flyway migration scripts), comprehensive documentation, experimental scripts, and deployment configurations.
– **Authors' contributions: Pham Thi Thuong** (first author): conceptualization, methodology, collaborative software development with Claude Code, design and execution of experiments, data collection and analysis, writing—original draft, writing—review and editing, supervision. **Nguyen Dang Quang**: literature search assistance, software testing support, writing—review and editing. **Ngo Thi Lan Phuong**: requirements gathering support, data collection assistance, writing—review and editing. All authors read and approved the final manuscript.
– **Use of AI:** Claude Code (Claude Opus 4.8, Anthropic) was used as a tool for: (i) retrieval and initial summarization of related literature, with all sources independently verified by the authors before citation; (ii) computation of descriptive statistics and formatting of numerical results from author-provided raw data—namely, the interaction logs, token consumption records, and execution traces collected during the human-AI collaborative development of SmartMedTender and the controlled experiments described in Section 7; (iii) generation of LaTeX tables and figures from author-specified layouts and the author-provided data described in (ii); and (iv) LaTeX formatting and copy editing for readability, grammar, and style. The experimental configurations were designed by the authors; Claude Code was used to automate the aggregation of output logs under author-defined parameters. All intellectual content—including the research design, analysis, interpretation of results, arguments, and conclusions—is the sole work of the authors, who reviewed, revised, and approved every part of the manuscript. No AI-generated content was incorporated without human verification.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., et al. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, (2023).
2. Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., Gazit, I. Taking Flight with Copilot. *Communications of the ACM*, 66(6), pp. 56–62, (2023). doi: `https://doi.org/10.1145/3589996`

3. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1877–1901, (2020).
4. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. O., Kaplan, J., et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, (2021).
5. Devlin, J., Chang, M. W., Lee, K., Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 4171–4186, (2019).
6. Gulwani, S., Polozov, O., Singh, R. Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1–2), pp. 1–119, (2017). doi: https://doi.org/10.1561/2500000010
7. He, P., Wang, S., Chen, T. H. CodePromptZip: Code-specific Prompt Compression for Retrieval-Augmented Generation in Coding Tasks with LMs. *arXiv preprint arXiv:2502.14925*, (2025).
8. Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., et al. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. *Proceedings of the International Conference on Learning Representations (ICLR)*, pp. 23247–23275, (2024).
9. Jiang, H., Wu, Q., Lin, C. Y., Yang, Y., Qiu, L. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 13358–13376, (2023). doi: https://doi.org/10.18653/v1/2023.emnlp-main.825
10. Jiang, H., Wu, Q., Luo, X., Li, D., Lin, C. Y., Yang, Y., Qiu, L. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 1658–1677, (2024). doi: https://doi.org/10.18653/v1/2024.acl-long.91
11. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K. SWE-bench: Can Language Models Resolve Real-world GitHub Issues? *Proceedings of the International Conference on Learning Representations (ICLR)*, pp. 54107–54157, (2024).
12. Kumar, A., Bajpai, Y., Gulwani, S., Soares, G., Murphy-Hill, E. Why AI Agents Still Need You: Findings from Developer-Agent Collaborations in the Wild. *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 432–444, (2025).
13. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 9459–9474, (2020).
14. Lin, F., Kim, D. J., Chen, T. H. SOEN-101/FlowGen: Code Generation by Emulating Software Process Models Using Large Language Model Agents. *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1527–1539, (2025).
15. Lyu, W., Wang, Y., Sun, Y., Zhang, Y. Will Your Next Pair Programming Partner Be Human? An Empirical Evaluation of Generative AI as a Collaborative Teammate in a Semester-Long Classroom Setting. *Proceedings of the 12th ACM Conference on Learning@Scale (L@S)*, pp. 83–94, (2025). doi: https://doi.org/10.1145/3698205.3729544
16. Mindom, P. S. N., Nikanjam, A., Khomh, F. Harnessing Pre-trained Generalist Agents for Software Engineering Tasks. *Empirical Software Engineering*, 30(1), pp. 39, (2025). doi: https://doi.org/10.1007/s10664-024-10597-8
17. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Communications of the ACM*, 68(2), pp. 96–105, (2025). doi: https://doi.org/10.1145/3610721
18. Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., et al. ChatDev: Communicative Agents for Software Development. *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 15174–15186, (2024). doi: https://doi.org/10.18653/v1/2024.acl-long.810
19. Stray, V., Barbala, A., Wivestad, V. T. Human-AI Collaboration in Software Development: A Mixed-Methods Study of Developers' Use of GitHub Copilot and ChatGPT. *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*, pp. 1325–1332, (2025).

20. Takerngsaksiri, W., Pasuksmit, J., Thongtanunam, P., Tantithamthavorn, C., Zhang, R., Jiang, F., et al. Human-in-the-loop Software Development Agents. *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 342–352, (2025).
21. Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., Cordeiro, L. C. How Secure is AI-generated Code: A Large-scale Comparison of Large Language Models. *Empirical Software Engineering*, 30(2), pp. 47, (2025). doi: https://doi.org/10.1007/s10664-024-10590-1
22. Vaithilingam, P., Zhang, T., Glassman, E. L. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*, pp. 1–7, (2022). doi: https://doi.org/10.1145/3491101.3519665
23. Vero, M., Mündler, N., Chibotaru, V., Raychev, V., Baader, M., Jovanović, N., et al. BaxBench: Can LLMs Generate Correct and Secure Backends? *Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267*, pp. 61344–61390, (2025).
24. Welter, A., Schneider, N., Dick, T., Weis, K., Tinnes, C., Wyrich, M., Apel, S. An Empirical Study of Knowledge Transfer in AI Pair Programming. *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 166–177, (2025). doi: https://doi.org/10.1109/ASE63991.2025.00022
25. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., et al. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *Proceedings of the Conference on Language Modeling (COLM)*, (2024).
26. Xu, F. F., Alon, U., Neubig, G., Hellendoorn, V. J. A Systematic Evaluation of Large Language Models of Code. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS)*, pp. 1–10, (2022). doi: https://doi.org/10.1145/3520312.3534862
27. Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., Press, O. SWE-Agent: Agent-Computer Interfaces Enable Automated Software Engineering. *Advances in Neural Information Processing Systems (NeurIPS)*, 37, pp. 50528–50652, (2024).
28. Ye, J., Ouyang, Y., Wu, Z., Dai, X. Out-of-distribution Generalization Challenge in Dialog State Tracking. *NeurIPS Workshop*, (2022).
29. Zhang, Q., et al. A Survey of Large Language Models for Software Engineering. *Science China Information Sciences*, 69(4), pp. 141102, (2026). doi: https://doi.org/10.1007/s11432-025-4670-0
30. Zheng, Z., Ning, K., Zhong, Q., Chen, J., Chen, W., Guo, L., et al. Towards an Understanding of Large Language Models in Software Engineering Tasks. *Empirical Software Engineering*, 30(2), pp. 50, (2025). doi: https://doi.org/10.1007/s10664-024-10602-0
31. Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., et al. Measuring GitHub Copilot's Impact on Productivity. *Communications of the ACM*, 67(3), pp. 54–63, (2024). doi: https://doi.org/10.1145/3633453