Table of Contents

1 AI-Augmented Development Methodology

1.1 Introduction: State of the Art Meets Production Reality

Modern software development has reached an inflection point. AI coding agents—particularly Anthropic's Claude Code CLI and similar tools—have transitioned from experimental curiosities to production-grade development accelerators. However, the gap between casual experimentation and effective production usage is measured in 1000+ hours of practice, not blog posts written after weekend trials.

This document outlines my approach to AI-augmented development workflows, the economic rationale behind systematic tooling adoption, and the professional requirements necessary for these methodologies to succeed in team environments.

1.2 Depth of Expertise vs. Internet Noise

1.2.1 Investment Profile

1000+ hours of hands-on practice across multiple AI coding platforms:

- **Claude Code CLI**: Extensive Agent Skills development, custom workflow design, and adoption advocacy
  - **Usage attestation**: 60,374+ sessions since July 2025 (https://share.cleanshot.com/xN1DCR9X)
  - 5.28 billion tokens processed, $2,953 API costs demonstrating systematic production usage
- **Production deployment**: Not prototyping or experimentation—these tools run critical business workflows daily
- **Cross-platform experience**: Claude Code, Codex, Cursor, and other AI-augmented IDEs
- **Production systems built**: Multiple business-critical systems developed entirely using AI-augmented workflows

Contrast with industry discourse: Most commentary about AI coding agents comes from developers who've used them casually for days or weeks. The noise-to-signal ratio is extremely high—many strong opinions based on minimal practice. 1000+ hours reveal capabilities and limitations that aren't visible in surface-level usage.

1.2.2 Claude Code CLI Specialization

Agent Skills Development: Active contribution to Claude Code ecosystem through:

- Custom Agent Skills creation for specialized workflows
- Internal popularization at Eon Labs (team adoption and training)
- Deep understanding of Claude Code's tool use patterns, context window management, and autonomous agent capabilities
- Experience with both successful and failed automation attempts—understanding each AI model's capability boundaries through iterative audit and constraint refinement

What "mastery" actually means:

- Understanding prompt engineering patterns that produce consistent results

- Structuring prompts to delegate all implementation decisions to AI agents while maintaining human oversight on business requirements
- Designing context-aware workflows that leverage large context windows effectively
- Building reliable systems on top of non-deterministic AI outputs
- Managing API costs through efficient prompt design and caching strategies

1.2.3 My Practice: AI-First Development Methodology

Through 1000+ hours of production usage, my workflow has evolved to maximize AI agent capabilities while focusing human effort where it delivers unique value.

Human Role—Bridging Business Reality to AI Implementation:

- **Business requirements interpretation**: Understanding real-world needs and translating them into clear specifications
- **Architecture selection**: Choosing appropriate architectural patterns based on business constraints, scalability needs, and team capabilities
- **Prompt engineering**: Crafting precise, thorough prompts that communicate requirements and context to AI agents
- **Output review and iteration**: Auditing AI-generated implementations, identifying issues, refining prompts based on results

AI Role—All Discretionary Implementation Decisions:

In my practice, AI agents handle:

- Complete implementation (code generation, testing, documentation)
- Architecture execution and detailed design decisions
- Requirements analysis and technical specification
- Code auditing and quality assessment
- Refactoring and optimization decisions

Key Principle: Humans make business-driven decisions (what to build, why it matters, which architecture fits business needs). AI makes all discretionary technical decisions (how to implement, which libraries, specific patterns, code structure).

Why This Works: The interface between business reality and AI capability is human prompting. My focus is making that bridge as accurate and thorough as possible—not second-guessing technical implementation details that AI agents handle more consistently at scale.

Important Clarification: This describes my evolved methodology after 1000+ hours of practice. It's not a universal prescription. Teams adopting AI-augmented development typically progress gradually, and different practitioners find different balances based on their experience level and comfort with agent autonomy.

1.3 Strategic Framework—Why Software Robustness Requires Different Principles than Business Advantage

1.3.1 Classical Strategy vs. Modern Software Engineering

In classical strategy, abnormal profits arise when a firm controls resources and capabilities that are valuable, rare, and difficult to imitate. These often take the form of proprietary knowledge, data, and processes that competitors cannot easily access or replicate.

By contrast, in modern software engineering practice, robustness and maintainability often come from conformity to shared idioms: widely used languages, standard libraries, and community-validated patterns.

The fundamental tension: In most businesses, margins come from secrets—proprietary knowledge, non-obvious insights, and idiosyncratic processes create advantage. That is **not** how modern software engineering works.

In software today, the opposite is usually true: the more we conform to shared idioms, the **more future-proof** we become.

- Idiomatic UI patterns → easier for users to understand and designers to extend
- Idiomatic language features → easier for other engineers and tools to reason about
- Widely adopted libraries used idiomatically → easier to hire for, easier to maintain, easier to replace

So while business advantage is often *bespoke*, software robustness and maintainability are often *idiomatic*.

1.3.2 The Three-Axis Framework for AI-Augmented Development

Through 1000+ hours of production AI agent usage, I've identified three critical tensions not adequately addressed in current industry discourse.

We write **A ⊣ B** to mean: "A is constrained / bounded by B."

Exploration ⊣ Specification: The agent's ability to explore new methods, libraries, and architectures is constrained by how prescriptive we are in the specification. If we stuff the prompt with bespoke, highly specific implementation details, we increase Specification and shrink Exploration. We prevent the agent from roaming the current ecosystem, discovering state-of-the-art, future-proof idioms.

Autonomy ⊣ Supervision: The agent's autonomy is constrained by the level and frequency of human supervision. If we intervene at every small decision, require approvals at every step, or constantly redirect it, we reduce Autonomy. That defeats the point of using an autonomous AI to free up human time.

Idiomatic patterns ⊣ Bespoke constraints: The agent's ability to choose idiomatic, community-tested patterns is constrained by bespoke constraints we impose. If we insist on custom frameworks, unusual architectures, and local "house styles" that diverge from the ecosystem, we block it from using the idioms that maximize compatibility, tooling support, and long-term maintainability.

1.3.3 Design Rule for Effective AI Agent Usage

If we overload an AI agent with bespoke instructions and micro-specifications, we do two things at once:

1. We import the "business secret sauce" mindset into a domain where it is often harmful—telling the agent to ignore the ecosystem and instead re-implement our quirks.
2. We mathematically reduce the left-hand side of each relation: heavy specification throttles exploration, frequent supervision throttles autonomy, bespoke constraints throttle idiomaticity.

As AI coding agents get smarter, the optimal balance tilts further left. We as humans must learn to let go while keeping just the right elements on the right side.

The design rule:

- Keep goals, constraints, and safety requirements **clear**
- Keep implementation details **non-bespoke and non-prescriptive** where possible

1.3.4 Iterative Boundary Discovery

The design rule above suggests maximizing the left side (Exploration, Autonomy, Idiomaticity) while maintaining essential constraints. But how do we know which constraints are essential?

My practice after 1000+ hours: Start maximally left, discover boundaries empirically through audit, constrain only when necessary.

The Process:

1. **Default to maximum left**: Begin with minimal specification, minimal supervision, minimal bespoke constraints. Let AI agents explore freely within business requirements.
2. **Audit outputs systematically**: Review generated code, test results, architecture decisions. Don't assume problems—discover them through evidence.
3. **Identify boundary conditions**: When audit reveals issues (inconsistent patterns, suboptimal choices, misunderstood requirements), these indicate model capability boundaries.
4. **Add targeted constraints**: Constrain only the specific areas where audit revealed problems. Don't pre-emptively restrict based on assumptions about what might go wrong.
5. **Iterate as models improve**: As AI models get smarter (new versions, better training), loosen constraints and re-test boundaries. Yesterday's necessary constraint may be tomorrow's unnecessary restriction.

Key Insight: Each AI model (Anthropic's Sonnet 4.5 Thinking for coding, OpenAI's GPT-5 Thinking for holistic reasoning, OpenAI's GPT-5 Deep Research for comprehensive analysis, etc.) has unique capability boundaries. You discover these through practice, not prediction. Start loose, audit carefully, constrain precisely.

Why This Works: Pre-emptive restriction limits what agents can accomplish. Empirical boundary discovery maximizes agent capabilities while maintaining quality through systematic audit. You only pay the supervision cost where it demonstrably adds value.

1.3.5 Implications for Netstrata's Software Development

For software engineering agents working toward the end-2026 completion deadline:

Maximize Exploration: Let agents discover current best practices for NSW compliance automation, testing frameworks, and deployment patterns rather than prescribing outdated approaches.

Maximize Autonomy: Reduce intervention overhead—trust agents to make routine implementation decisions, reserving human judgment for architectural choices and domain-specific NSW regulatory requirements.

Maximize Idiomatic Patterns: Competitive advantage comes from domain knowledge (decade of encoded NSW regulatory expertise), not from custom frameworks. Use community-standard tools idiomatically to maximize maintainability of the $12-14M investment.

The insight: Netstrata's moat is the NSW strata management knowledge embedded in the software, not the technical implementation patterns. Letting AI agents apply idiomatic, state-of-the-art software engineering practices protects that investment better than insisting on bespoke technical approaches.

1.4 Knowledge Transfer and Team Capability Building

1.4.1 Training Offerings for Netstrata's Software Team

Onboarding Workshops: Structured introduction to AI-augmented workflows:

- Tool installation and configuration (Claude Code CLI setup)
- Basic prompt engineering patterns for common development tasks
- Context window management strategies
- Integration with existing development workflows

Pair Programming Sessions: Hands-on learning through collaboration:

- Real-world problem solving using AI-first workflows
- Demonstrating AI agent orchestration that eliminates implementation bottlenecks
- Building intuition through repetitive practice
- Troubleshooting common failure modes

Custom Workflow Design: Tailoring AI tools to Netstrata's specific needs:

- Identifying high-value automation opportunities in current development process

- Building custom Agent Skills for recurring tasks
- Establishing team conventions for AI tool usage
- Creating feedback loops for continuous improvement

Assessment and Iteration: Not one-time training, but ongoing capability building:

- Measuring adoption rates and productivity impacts
- Identifying barriers to effective usage
- Adjusting workflows based on team feedback
- Building internal champions for best practices

1.4.2 Realistic Expectations

Not a silver bullet: AI coding agents don't replace software engineering judgment—they reallocate it. Human judgment shifts from implementation details to business requirements, architecture selection, and quality assessment. AI handles discretionary technical decisions.

Learning curve exists: Team members need weeks of practice to become proficient, months to develop mastery.

Cultural fit matters: Some developers embrace these tools quickly; others resist. Effective adoption requires collective buy-in.

1.5 Assessment and Adoption Strategy

1.5.1 Why Demonstration Matters More Than Debate

Industry discourse about AI coding tools is dominated by surface-level usage—developers who've experimented for days or weeks forming strong opinions based on minimal practice. The noise-to-signal ratio is extremely high.

The Reality Gap: 1000+ hours of production usage reveals capabilities and limitations invisible in weekend trials. The difference between casual experimentation and professional mastery is substantial—similar to the gap between writing your first Python script and architecting production systems.

Rather than debate capabilities: Let measurable results speak. Industry skepticism becomes irrelevant when faced with production evidence: working systems, quantified time savings, demonstrable velocity gains.

My approach: Skip theoretical arguments about what AI agents can or cannot do. Instead, work on real Netstrata challenges and document objective outcomes. Let the team assess value based on evidence from their actual codebase, not industry blog posts.

Key advantage: While competitors debate whether these tools are "ready," early adopters accumulate 12-24 months of practical experience and velocity gains. By the time industry consensus shifts, Netstrata will have already integrated AI-first development into the end-2026 completion timeline.

1.5.2 Demonstration-First Approach

I know AI-first development works—1000+ hours and multiple production systems provide that evidence. The question isn't whether the methodology is viable. The question is whether Netstrata's team can adopt it effectively and whether the timing aligns with end-2026 completion priorities.

Demonstration reveals fit:

First 30-60 Days: Work with Tom Bakani's software team on Phase 1 completion tasks using AI-augmented workflows. Approach can be tailored based on team's current priorities and the specific challenges they're facing.

Team Observation: Pair programming sessions and live demonstrations where team members can see these tools in action on the actual Netstrata codebase. Real code, real challenges, real results—not theoretical examples.

Measurable Outcomes: Document time savings, code quality improvements, and development velocity gains. Quantifiable data rather than subjective impressions.

Results-Based Decision: After 30-60 days, the team has direct experience with AI-first workflows on their actual codebase. If the methodology accelerates Phase 1 completion and team members see value, we explore knowledge transfer and broader adoption. If the fit isn't there—timing, team readiness, or other factors—we have clear evidence to inform that decision.

1.5.3 Why This Matters for Netstrata's Timing

Competitive Window: Strata software competitors (Strata Master, :Different, external vendors) are unlikely to be using cutting-edge AI development practices yet. Industry-wide skepticism means early adopters gain 12-24 months of velocity advantage.

Critical Insight: By the time industry consensus shifts to "these tools are production-ready" (which may take 12-24 months given current skepticism levels), Netstrata will already have that time period's worth of accumulated advantage. You're not adopting unproven technology—you're adopting ahead of the curve while competitors wait for consensus.

End-2026 Deadline Pressure: Phase 1 completion timeline benefits immediately from accelerated development velocity. This isn't theoretical future benefit—it's practical advantage on current priorities with concrete deadlines.

Protection of $12-14M Investment: AI-assisted code review, comprehensive testing, and systematic quality assurance become especially valuable as the codebase approaches completion. The cost of bugs and technical debt increases as you get closer to external rollout.

1.5.4 Mutual Assessment Process

This opportunity requires fit assessment on both sides:

For Netstrata to Assess:

- Do AI-augmented workflows deliver measurable value for software completion timeline?
- Does the velocity improvement justify the learning investment for Tom Bakani's team?
- Are the competitive timing advantages real for the strata software market?

For Me to Assess:

- Is the team open to exploring modern development practices through demonstration?
- Will leadership (Tom Bakani, Andrew Tunks) support capability-building efforts?
- Can we establish feedback loops for continuous improvement?

Better to identify misalignment through demonstration and data than assume compatibility or incompatibility without evidence.

1.5.5 No Pressure, Just Results

I've practiced AI-first development for 1000+ hours across production systems. I'm confident in the methodology. But confidence means demonstration, not persuasion.

My offer:

1. **Work on real Netstrata challenges** (Phase 1 completion tasks) using AI-first workflows
2. **Document measurable outcomes** (time savings, code quality, velocity gains)
3. **Let results inform decisions** (team sees value or identifies misfit based on evidence)

If Tom Bakani's team sees value after demonstrations and wants to explore adoption, I'm equipped to support that transition through the training and knowledge transfer approaches outlined in Part 3.

If they remain unconvinced after seeing production usage over 30-60 days, that's valuable signal about team readiness and fit. Some teams need longer adoption curves or different timing—assessment through demonstration reveals whether this is the right methodology and moment for Netstrata.

1.5.6 Team-Wide Effectiveness vs. Individual Heroics

One observation from extensive experience: AI-augmented development works best as a team practice rather than individual capability. When one person uses these tools and others don't, it can create coordination friction (different development speeds, code review challenges, knowledge silos).

However, this doesn't mean "all or nothing" adoption. It DOES mean:

Gradual, organic adoption: Start with demonstrations, let interested team members experiment, build capability progressively. Not forcing anyone, but creating environment where exploration is encouraged.

Leadership support: Tom Bakani and technical leadership actively encouraging the team to evaluate these approaches fairly, not dismissing them as hype.

Patience for learning curves: Understanding that proficiency takes weeks/months of practice, not days. Allow time for team members to develop intuition.

Assessment based on results: If demonstrations show clear value, team explores further. If results are marginal, we reassess. Data-driven decisions.

1.6 Conclusion: Methodology Informs Contribution

This AI-first methodology directly shapes how I'd contribute to Netstrata's three phases:

Phase 1 (Software Completion): Apply AI-first workflows to Phase 1 completion tasks, document measurable velocity gains, transfer knowledge to interested team members, accelerate progress toward end-2026 deadline through systematic agent orchestration.

Phase 2 (WA Migration): AI agents excel at migration infrastructure—documentation generation, validation scripts, data transformation pipelines. Apply AI-first development to WA customer rollout preparation.

Phase 3 (External Rollout): If Phase 1-2 demonstrations prove value, scale AI-first practices across broader external rollout efforts. Let results from earlier phases inform adoption decisions.

The core insight: Competitive advantage comes from adopting evolved methodologies while competitors wait for consensus. For a $12-14M software investment approaching end-2026 completion, AI-first development offers measurable velocity gains at the most critical moment.

The window is now—while industry debates whether these tools are "ready," early adopters accumulate 12-24 months of practical advantage. Let production results on Netstrata's actual codebase speak louder than industry discourse.