AI MADE IT FASTER.
YOU WENT BACK TO WATERFALL.

How Agile Discipline Restores Flow

AI coding agents write code faster than teams can review it. The instinct is to hand off more work in bigger batches. That is waterfall. This paper argues for the opposite: small stories, one at a time, reviewed and tested before the next begins. It presents the solo practitioner as the atomic unit of this model and shows how the same principles scale to teams. The constraint is no longer coding speed. It is story quality, review rigor, and testing discipline.

Sheila Eckert (with AI editing assistance)
Author, Agile Meets AI | Founder, Frankee.ai

Executive Summary

AI coding agents can translate a well-defined feature into working software in short order. This is a significant shift in how software gets built.

However, many teams are responding to this speed by increasing batch size. Instead of working in small increments, they are writing large sets of requirements, handing them to AI agents, and deferring validation until the end. This recreates the same front-loaded planning and back-loaded validation that made waterfall development slow and unreliable. The speed of AI does not eliminate these constraints. It intensifies them.

The alternative isn't to limit AI usage, but to pair it with established agile principles: small, well-defined stories, continuous integration, and rapid feedback loops. When AI agents are applied to tightly scoped work, teams can generate, review, test, and ship increments in short cycles, maintaining both speed and control.

This paper presents a model built around that approach. It begins with the solo practitioner as the clearest expression of the pattern, where one story is completed, reviewed, and validated at a time. It then extends these principles to teams, where the separation between story definition and code review and parallel iteration cycles increases throughput without sacrificing quality.

Teams that use AI to increase batch size will recreate waterfall dynamics. Teams that use it to increase iteration speed will achieve sustained improvements in flow, quality, and delivery.

1. AI Coding Agents

AI coding agents are a significant breakthrough in software development. Tools like Claude Code, GitHub Copilot Workspace, Cursor, and others can take a natural language description of a feature and produce working code, complete with tests, documentation, and even deployment configurations.

AI can code an entire microservice, crank through complex logic, or clean up a legacy mess in no time. **The question now is, how do we organize work around that capability?**

And this is where many teams are making a critical mistake.

2. The Waterfall Trap

Because an AI agent can implement an entire feature in hours, the natural reaction is to give it even more. The reasoning goes like this: if the AI can code that fast, we should spend our time writing thorough requirements and comprehensive test cases upfront, then hand the whole thing off. Sound familiar?

This is waterfall. The workflow follows the same pattern that failed for decades: a long requirements phase, a compressed build phase, and then an enormous validation phase at the end.

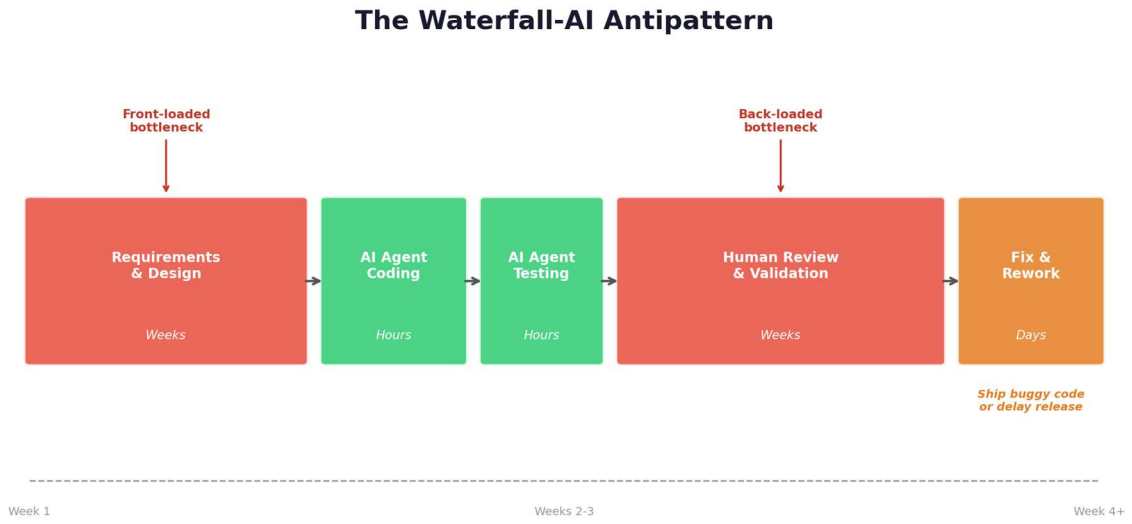

Figure 1: The Waterfall-AI Antipattern: speed in the middle creates bottlenecks at both ends.

2.1 The Front-Loaded Bottleneck

To hand a large chunk of work to AI coding agents, you need real requirements. Not hand-wavy ideas. Every edge case must be handled. Every integration point must be understood. Acceptance criteria need to be clear. This is requirements engineering at its most ambitious and its most fragile.

There is a competing view that says none of this is needed. The promise is that you can describe what you want in plain language and get working software back. A sentence. A paragraph. A rough idea. Vibe coding. For a personal project or a quick prototype, it works. The cost of being wrong is low. You tweak it until it looks right.

Production software plays by different rules. A vague prompt will generate code. The problem is that you don't know if it does the right thing. You don't know how it handles edge

cases. You don't know if it integrates cleanly, performs under load, or behaves securely. That only shows up when someone checks.

At a small scale, that checking is manageable, but when you hand off a large batch of work, it becomes a bottleneck. Validation gets pushed to the end. What looked fast upfront slows everything down later.

The answer is not to swing back to massive upfront requirements. Agile already showed us how that fails. People aren't good at predicting what they need before they see it. The answer is better requirements at the story level. Small, focused stories with clear acceptance criteria, edge cases, and assumptions. Written just in time. That's a very different discipline from either vibe coding or big upfront requirements. It's also what makes these ai-paired iteration cycles work.

2.2 The Back-Loaded Bottleneck

The AI agents do their job. In a matter of hours, they can produce thousands of lines of code, dozens of test files, updated configurations, and new API endpoints. Now what?

Someone still has to review it. Not skim it. Actually read it, understand it, check the edge cases, confirm the integrations, and assure it aligns with the true business intent. That is a lot of cognitive load that lands back on the humans. And now this is a bottleneck.

Software engineers generally prefer writing code over reading it. Creating feels productive. Reviewing thousands of lines generated by someone else, or by something else, is slow, tedious, and mentally draining. Most engineers will tell you privately that it's the part of the job they like least. It’s why, as a consultant, I often saw code-review as a bottleneck in the workflow.

AI-assisted development accelerates what people enjoy and expands what they avoid. The review burden grows faster than teams can keep up.

2.3 The Impossible Choice

Teams hit a point where they are staring at a massive batch of AI-generated code. At that moment, there are two options, and neither works well.

Trust it and ship. This is the fast path. It's also a risky one. You're putting code into production that no one has fully validated. Not just correctness, but behavior under load, security gaps, unintended side effects, and how it interacts with the rest of the system.

When something goes wrong, you react in production instead of learning earlier, when it was cheaper and safer.

Review everything. This is the responsible path. It's also a slow one. The larger the batch, the harder it is to reason about. Review quality drops. Important details get missed. The time it takes to build confidence in the code can easily exceed the time it would have taken to build it iteratively in the first place.

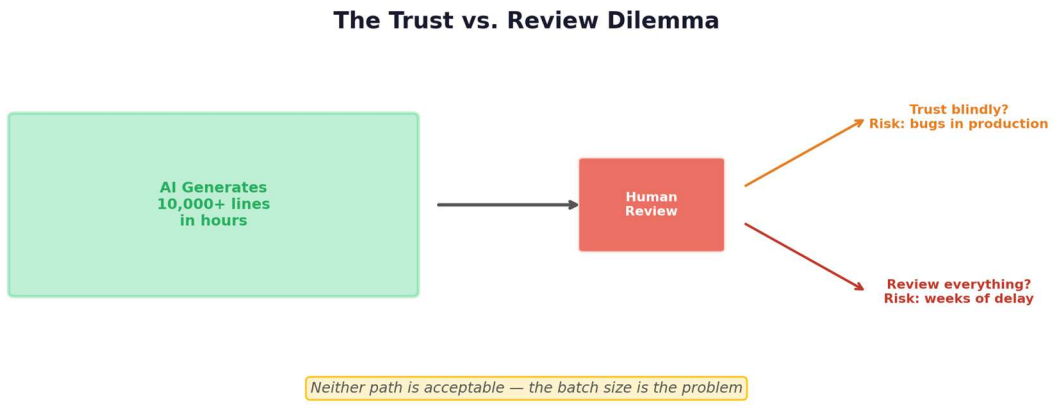

Figure 2: The Trust vs. Review Dilemma: large AI output forces an impossible choice.

Either you move fast and take on risk, or you slow down to manage it and lose speed.

3 We Have Been Here Before

We've been here before. From the mid-90s through the early 2010s, Agile pushed out waterfall as the way of working in most software organizations. The shift was simple. Smaller batches. Faster feedback. Continuous delivery. That produced better outcomes than the exhaustive upfront planning followed by building for months, only to find out at the end if you got it right.

Waterfall failed not because people were bad at coding. It failed because the feedback loop was too long. By the time a team discovered that requirements were wrong, they had already built the wrong thing. Agile shortened the feedback loop from months to weeks.

The irony of the current AI moment is that some teams are now extending their feedback loops back to waterfall timescales, not because the technology demands it, but because the speed of AI code generation makes large batches seem efficient. They're not.

4 The Agile-AI Flow

The right approach is to combine the speed of AI agents with the discipline of agile iteration. Instead of handing off a week's worth of work, you hand off a single story. Instead of writing exhaustive upfront requirements, you maintain a well-structured, prioritized backlog. Instead of a massive review at the end, you review a small, focused change. Common agile practices.

4.1 The Iteration Cycle

Each iteration follows a tight, repeatable cycle:

1. Pick a single story from the prioritized backlog. Small, focused, with clear acceptance criteria.

2. Hand the story to the AI agent. The agent codes the solution across all layers (frontend, backend, database), writes and runs tests, and produces a working increment.

3. Review the change. Because the diff is small and contained to one story, review is fast and meaningful. Read through every file change. Understand what AI did, new code, deleted code, changed code.

4. Test the increment. Run automated tests, then manually verify against acceptance criteria. Check edge cases. Look for usability issues the automated tests won't catch.

5. Build and deploy. Verify the build passes and ship to a staging or production environment.

6. Move to the next story.

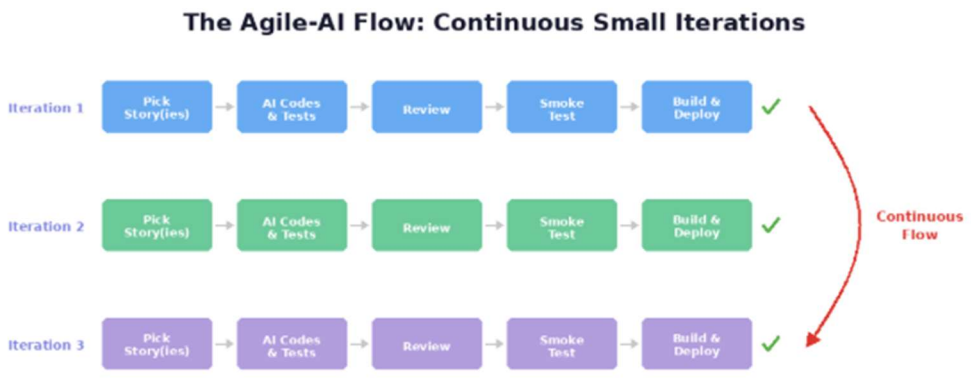

Figure 3: The Agile-AI Flow: small iterations create continuous delivery.

Each cycle is short and focused, and produces a tested, reviewed, shippable increment. A practitioner working at this cadence can complete multiple stories per day, each one reviewed and validated before the next one begins.

4.2 What Makes Stories AI-Ready

The iteration cycle depends on something practitioners consistently underestimate: stories that are already small enough, focused enough, and specific enough for an AI coding session. Writing stories at that level of precision is skilled work. It requires understanding what the AI agent needs to produce meaningful output, what a reviewer needs to evaluate the result, and what a tester needs to validate the increment.

What does an AI-ready story look like? It follows INVEST principles: independent, negotiable, valuable, estimable, small, and testable. It has specific acceptance criteria that both the AI agent and the human reviewer can evaluate against. Edge case scenarios are defined so the AI doesn't have to guess at boundary behavior. Assumptions are stated explicitly rather than left for the AI to infer. UX and UI treatment information gives the agent ample context to produce an interface that actually works for users, not just one that compiles. Dependencies on other stories or system components are called out so the AI doesn't build on foundations that don't exist yet.

When foundational work is needed, such as establishing APIs, data models, infrastructure, or architectural scaffolding, it is written as separate enabler stories and completed first. This gives the AI agent well-defined building blocks to work with when the vertical slice stories begin.

Stories at this level of specificity produce better AI output. The agent has clear boundaries, the reviewer has clear criteria, and the tester knows exactly what to validate. Vague stories produce vague code that requires multiple rounds of correction, eroding the productivity gains the AI was supposed to deliver.

4.3 Why This Process Works

Small batches solve every problem that large batches create:

- **Requirements stay lightweight.** There's no need to anticipate everything upfront because working software appears within hours. If something is wrong, the practitioner catches it immediately and adjusts.
- **Review is manageable.** A focused diff from a single story is something a human can genuinely understand and validate. A diff spanning an entire feature is something a human can only skim.
- **Feedback is immediate.** If the AI misunderstood the intent, you find out in hours, not weeks. The cost of correction is a single iteration, not a major rework effort.
- **Risk is contained.** If a single iteration produces poor results, you have lost hours, not days or weeks. The rest of the codebase remains stable and shippable.

5 The Solo Practitioner Model

The clearest demonstration of the Agile-AI flow is a solo practitioner building production software with AI coding agents. There's no handoff to hide behind. No team process to absorb mistakes. Every strength and weakness of the model is immediately visible. This is where the principles are proved, and it is the foundation from which team models are built.

5.1 The Solo Workflow

The rhythm is simple. Take one story. Hand it to the AI agent. Review the output. Test it, both the automated tests and manual validation. Ship it. Move to the next story. One at a time, each is reviewed and tested before the next begins.

This cadence matters more than it might seem. The temptation to queue up multiple stories or let the AI run ahead is strong, especially when the agent works fast. But queuing stories creates exactly the batch size problem this paper warns against, just at a smaller scale. A solo practitioner who lets three stories pile up before reviewing any of them is a one-person waterfall.

For a solo practitioner, the hard part is wearing both hats: product owner and developer. As the PO, you write and refine the story, or you use a tool like Frankee to generate it and then

refine it until the acceptance criteria, edge cases, and UX treatment are right. That's one mindset. As the developer, you give the story to the AI, then review what it produces against the story you wrote. That's a different mindset. The key discipline is making that switch cleanly. Review the diff as if someone else wrote it. Read it against the acceptance criteria line by line. Some practitioners walk through the output systematically against the story before approving it. Others use a checklist. The method matters less than the discipline: the mindset that defined what to build isn't the right one for evaluating whether it was built correctly. Creating distance between the two is what makes a solo review meaningful rather than a rubber stamp.

5.2 Right-Sizing Scope

Solo practitioners working with AI agents need to think carefully about scope at two levels: the individual story and the goal that groups stories together.

At the story level, keeping scope small is critical. Stories sized at five points or below work well for AI coding sessions. They produce changes that are reviewable in a sitting and testable in a short cycle. Occasionally, a story can't be split further and lands at eight points, but this should be the exception, not the rule. Larger stories produce larger diffs, which lead to less thorough reviews, which in turn lead to more bugs.

At the goal level, practitioners benefit from organizing stories into features or epics of four to seven stories. This provides a meaningful unit of progress: a complete capability delivered over a day or two, without the scope creep that comes from larger goals. Working with larger scope goals spanning multiple epics introduces complexity that becomes difficult for a single person to manage effectively. Smaller goals that a practitioner can complete in a focused day keep the work tractable and the feedback loops tight.

The right scope for a solo practitioner isn't the smallest possible unit. It's the largest unit that allows for in-depth review and testing of every story. When things start getting lightly skimmed instead of read, the scope is too large.

5.3 The Testing Reality

AI coding agents generate tests alongside implementation. These tests run, they pass, and coverage numbers look healthy. This creates a false sense of quality.

In practice, AI-generated tests have consistent gaps. They tend to test the happy path thoroughly, while missing edge cases that a human tester would catch. They verify that the code does what it was written to do rather than what it was intended to do. Integration points between components are frequently undertested. Performance under realistic load

is rarely addressed. And usability issues, the kind that make a user pause, backtrack, or reach for a workaround, are invisible to any automated test.

Practitioners who rely exclusively on AI-generated tests will ship bugs. Some will be minor. Some will drive users to workarounds that become permanent habits. Anyone who uses SaaS products regularly has experienced this: the feature that almost works, the workflow that requires an unintuitive extra step, the behavior that was clearly never tested by a human who actually tried to use the product. These aren't just annoyances. They're evidence that somewhere in the development process, automated tests passed, and nobody checked whether the software actually worked for people. Anyone who has watched a product degrade after a release cycle has seen what happens when automated testing replaces human verification rather than supplementing it.

The discipline is adding some manual testing on every feature. Not every individual story necessarily warrants a full manual pass, but every feature, the goal-level unit of four to seven stories, demands hands-on verification. Run through the feature as a user would. Try the edge cases. Attempt the things a user might do that nobody explicitly specified. This is where the bugs live, and it's where product quality is won or lost.

AI-assisted development doesn't eliminate the need for human testing. It makes human testing more important because the volume of code being produced is higher, and the automated safety net, while broad, has predictable blind spots.

6 Scaling to Teams

6.1 From Solo to Team

The solo practitioner model is the foundation of the Agile-AI flow. Everything that works at the solo level (small stories, one-at-a-time execution, full-stack review, manual testing) scales directly to teams. What teams add is separation of concerns that a solo practitioner must simulate through discipline alone.

The most important addition is that the roles a solo practitioner must juggle are held by different people. The Product Owner (or PO and team, or a tool like Frankee) writes and refines the stories. Developers give those stories to AI agents and review the output. The person who defined what should be built isn't the same person evaluating whether AI built it correctly. That natural separation is the single most important quality control mechanism in AI-assisted team development, and it happens without the discipline overhead that solo practitioners must impose on themselves.

Teams can also run iteration cycles in parallel. While one developer builds stories from one set, another works on stories from another. While developers iterate, a QA practitioner tests completed increments. The throughput gain is real, but only if the core discipline (small stories, thorough review, manual testing) remains intact.

6.2 Team Roles in the Agile-AI Flow

The **Product Owner** role gets harder, not easier. If the team can complete so much in a day, the PO has to stay ahead. That means continuously refining what's next while the current work is in flight, writing clear acceptance criteria, and making real-time prioritization calls as working software shows up throughout the day.

The **Scrum Master** is invested in optimizing flow. The SM monitors iteration throughput across the team, identifies bottlenecks, and facilitates improvement. Batch size discipline remains one of the Scrum Master's most important contributions. A team without it quietly drifts back toward waterfall, one oversized piece at a time. In enterprise environments, this drift is even easier because the organizational environment provides constant pressure toward larger batches: dependencies that encourage bundling, environments that reward infrequent deployments, and governance processes designed around release events rather than continuous flow.

Developer work follows the same principles as the solo model. The story is the prompt. A developer gives a fully loaded story to the AI agent, and the AI produces the full vertical slice. The developer then reviews the output against the story, evaluating whether the AI met the acceptance criteria across all layers. Because the PO defined what should be built and the developer evaluates what was built, the quality gate is natural rather than forced. Over time, the AI-assisted workflow accelerates cross-functional capability as developers review full-stack output and build fluency across areas they may not have worked in before.

Quality assurance in the Agile-AI model has two dimensions. Automated test validation ensures that AI-generated tests actually test meaningful behavior rather than merely achieving coverage goals. Manual and exploratory testing catches what automated tests can't: usability issues, visual regressions, unexpected interactions between features, and whether the implementation is consistent with the business intent behind the story. Both matter more in AI-assisted development, not less. Organizations that run developer-owned quality will recognize the same accountabilities distributed across the development team. The structure differs, but the discipline doesn't.

6.3 Full-Stack Discipline

Agile stories have always been end-to-end. That's not new. A single story delivers value across the whole application. It can touch the database, the API, the business logic, and the user interface. All of it ships together because that is what it takes to deliver real user value.

In an AI-assisted model, it's a mistake to split that story back into frontend and backend work and have developers prompt AI separately. That creates handoff and coordination overhead. You end up stitching the work back together and hoping it holds, and when it doesn't, the integration bugs surface late, in review or in production, where they are most expensive to fix.

Let AI generate the full vertical slice needed to meet the story. If the work spans the stack, the review has to span the stack. A frontend-only perspective isn't enough to validate backend changes, even if the UI looks right. Splitting the review by specialization slows things down and misses the real risk: how the parts work together.

Full-stack review capability is both more efficient and more effective. One person following the flow from database to API to UI can validate the integrity of the whole change. Otherwise, you're either rubber-stamping what you don't understand or adding multiple reviewers to a single story and slowing the cycle.

AI doesn't respect layer boundaries. Review can't either.

The upside is that AI makes this easier. Reviewing code in unfamiliar areas is one of the fastest ways to build that capability. The patterns show up quickly, and AI can explain its reasoning when asked. Over time, the hard lines between frontend and backend start to fade, and developers become genuinely versatile across the stack.

7 Side-by-Side: Two Approaches to the Same Feature

Consider building a user management module with authentication, a dashboard, notifications, settings, and reporting capabilities. Here is how the two approaches compare:

Dimension	Waterfall + AI	Agile + AI
Batch size	Entire feature (weeks)	Single story (hours)
Requirements	Exhaustive upfront	Just-in-time, per story

AI coding time	Hours (large batch)	Minutes (small batch)
Review effort	Massive, overwhelming	Small, meaningful
Time to first ship	3–4 weeks	Day 1
Risk of rework	High (late discovery)	Low (fast feedback)
Product bloat	Likely (speculative scope)	Unlikely (prioritized)
Team flow	Blocked by bottlenecks	Continuous delivery

Timeline comparison: same scope, dramatically different flow and risk profiles.

7.1 A Field Test: Seven Stories, Two Approaches

The theoretical comparison above reflects what I observed in practice during a small experiment on the Frankee codebase. I took a set of 7 user stories and ran them through an AI coding agent in two ways: first as a consolidated plan, then as individual stories.

For the consolidated approach, I merged all seven stories into a single requirements document and handed it to the agent. I had the agent create a plan, which took a couple of hours. The result would have added roughly 800 lines of code and removed about 100, spread across 9 files. It was a large, interconnected change that would have landed as a single review.

I'll be honest, the idea of committing that much AI-generated code at once made me uneasy. Eight hundred lines across nine files is a lot to validate in one pass. That discomfort itself is data. If someone who builds with AI agents daily hesitates at that batch size, which isn't massive, I imagine most others will too.

For the story-by-story approach, I used the seven well-scoped user stories written by Frankee and ran them individually. The results were dramatically different in character. The agent plan for each took minutes. The largest story touched seven files, but most of those changes were just one to three lines. The total for that story was a bit over one hundred lines added or deleted. The smallest, a single-point story, consisted of 30 lines of code added to one file. Every story produced a change I could review in minutes, not hours. Something easy to follow and not overwhelming.

The most telling difference was what happened after each story. Because I could review, test, and validate each story quickly, I spotted opportunities to improve usability that

emerged only from using the feature in small pieces. These were not bugs or oversights in the code. They were insights that came from the tight feedback loop between implemented software and actual use, exactly what agile is designed to create.

I never committed the code from the consolidated experiment, so I can't know whether those same refinements would have surfaced buried inside an eight-hundred-line review. But the mechanism matters: small working increments surface insights that large batches bury. You discover things by touching working code that you miss in comprehensive planning.

This was a small test on a single codebase, not a controlled study. But the pattern it revealed is consistent with everything this paper argues: small working increments surface insights that large batches bury.

8 Preventing Product Bloat

One underappreciated risk of large-batch AI development is product bloat. When it costs little for an AI to add a feature, the temptation is to add everything. Requirements documents grow to include speculative features, nice-to-haves, and edge cases that may never matter.

Agile prioritization acts as a natural filter. When a practitioner or team works from a prioritized backlog and delivers in small increments, every feature must earn its place. Working software is delivered after each iteration, and well-informed decisions about what to build next can be made based on what actually works, not on what sounded good in a planning session. Features that seem valuable in a requirements document but don't deliver value in practice get deprioritized before the AI ever writes them.

This is about building the right product, the right way.

9 Practical Recommendations

For practitioners and teams adopting AI coding agents, the following practices maximize value while avoiding the waterfall trap:

1. Structure your backlog for AI handoff. Stories should be small, specific, and self-contained. Each story should be completable in a single AI coding session. Keep story points at five or below, with eight only when a story genuinely can't be split further.

2. Invest in story quality, not specification volume. The human effort that matters most is guaranteeing each story is AI-ready: INVEST-compliant, with specific acceptance criteria, defined edge cases, stated assumptions, UX/UI treatment, and clear dependencies. Exhaustive requirements documents are the wrong investment. A well-maintained backlog of precise, AI-ready stories is the right one.

3. Work one story at a time. Review and test each story before moving on to the next. Queuing multiple stories before reviewing any of them recreates waterfall at a smaller scale. The discipline of completing each cycle before starting the next is what keeps quality high and rework low.

4. Review the full stack. Allow AI agents to produce vertical slices that span frontend, backend, and database. The reviewer must be able to evaluate changes across all layers. Don't split AI work by layer. It introduces coordination overhead and integration risk.

5. Don't trust AI-generated tests alone. AI-generated tests pass. That doesn't mean they test what matters. Manually test every feature. Run through it as a user would. Try the edge cases nobody specified. This is where bugs live.

6. Right-size your goals. Organize stories into features or epics of four to seven stories. This provides valuable progress units without the complexity that comes from larger scopes. For solo practitioners, goals that can be completed in a focused day keep work tractable.

7. Treat AI speed as a throughput multiplier, not a batch size multiplier. The goal is more iterations per day, not more code per iteration.

8. Ship continuously. Every completed iteration should be deployable. Use feature flags if needed but maintain the discipline of continuous delivery.

10 Conclusion

AI coding agents are transformative. They will fundamentally change how software is built. But the technology doesn't dictate the workflow. Practitioners have a choice: use AI speed to build bigger batches and recreate waterfall, or use AI speed to iterate faster and amplify agile.

The modern constraint is the quality of the work going into AI and coming out of it: story precision, review rigor, and testing discipline. Practitioners who get those right will get the most out of AI.

The future of AI-assisted development isn't agents writing entire features in one pass. It's agents completing focused stories in hours, practitioners reviewing small changes they can actually understand across the full stack, manual testing catching what automated tests miss, and working software shipping continuously. **That's not a new idea. It's agile, turbocharged.**

About the Author

Sheila Eckert is a practitioner and author with over thirty years of experience in software development and agile delivery. She is the author of Agile Meets AI: A Pragmatic Guide for Modern Teams and the founder of Frankee, an AI-powered agile teammate that helps teams maintain the backlog discipline required for effective AI-assisted development. Sheila built Frankee using the exact methodology described in this paper. She works one story at a time, reviewing every change across the full stack and manually testing every feature. More at thesheilaverse.com.

Sheila Eckert | frankee.ai