From Agile to AI-Native

A problem in both process & perspective.

ACT 1

Questions & Concerns

What happens when the emphasis shifts?

For my whole career, our entire methodology assumed one thing:

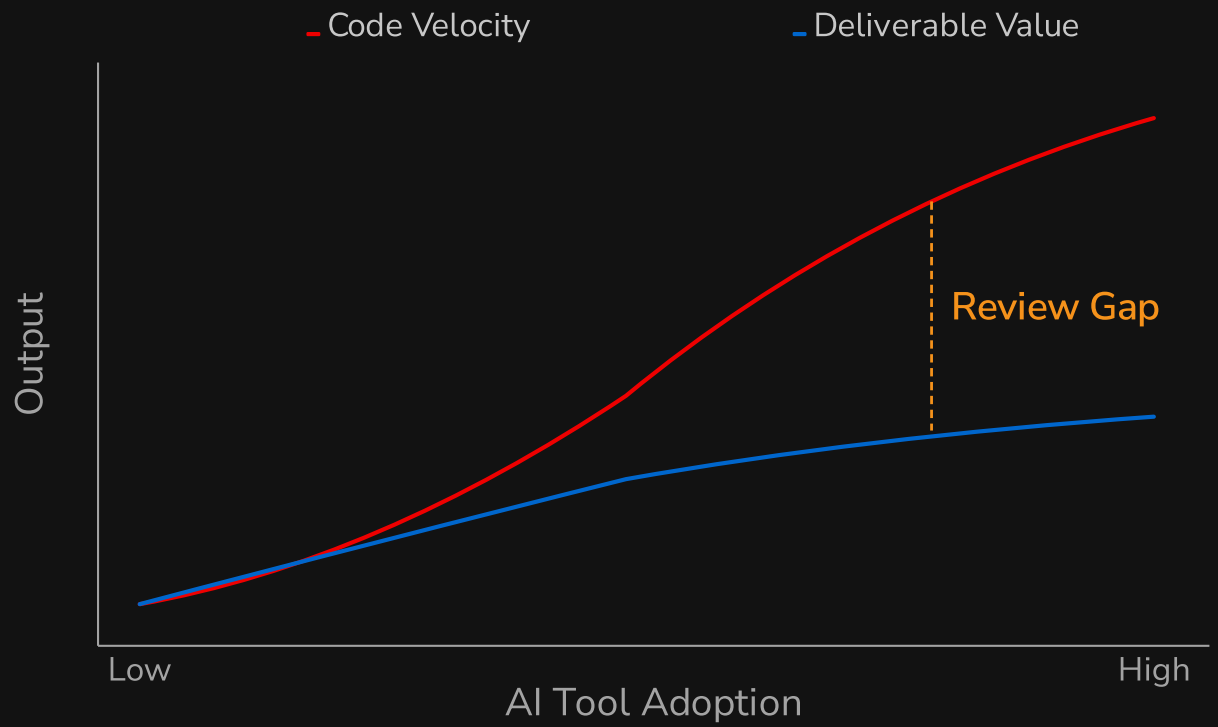
Problem solving is the hard part.

Sprints, story points, velocity, stand-ups -- all designed around the assumption that 'coding' is a main factor affecting capacities and timelines.

What happens when it isn't?

How do we communicate about it?

Where do I watch for technical debt?



We have to change our perspective of "review" & we have to own it.

Sprints would fall apart because they're designed to end-load "review".

Is estimation still possible? Important?

Before AI

- 8-point story = ~1 week of work
- Stand-ups surface blockers
- Velocity charts track capacity
- Sprint commitments are meaningful

With AI

- 8-point story takes... 2 hours? 2 days?
- Blockers resolve in 30 seconds
- Velocity charts become meaningless
- Sprint commitments are probably theater

> Story points estimate **coding effort**. When coding effort drops 10x but review effort grows, does the system break?

Is vibe coding a meme or legitimate concern?

Defining it

Iterative, ad-hoc prompting where the output "feels" correct. No spec, no plan, no verification criteria.

Where It's Good

Prototypes, one-off scripts, personal tools, hackathons. Anywhere throwaway code is acceptable.

Where It Fails

Teams, production systems, anything with history. Bus factor of 1, no source of truth, context drift between sessions.

Most people start here. That’s fine, but how do we grow from here?

Will we lose our ability to understand?

The act of typing code forces me to build a mental model of systems.

Traditional Loop

Read code → reason about state → write code → debug → **understand**

Insufficient Agentic Loop

Describe intent → agent generates → skim output → ship → ???

Engineers risk becoming **"passengers" in their own codebase** – capable of generating features but incapable of explaining their implementation or recognizing / debugging complex failures.

ACT 2

My 'AHA!' Moment

Here be math dragons

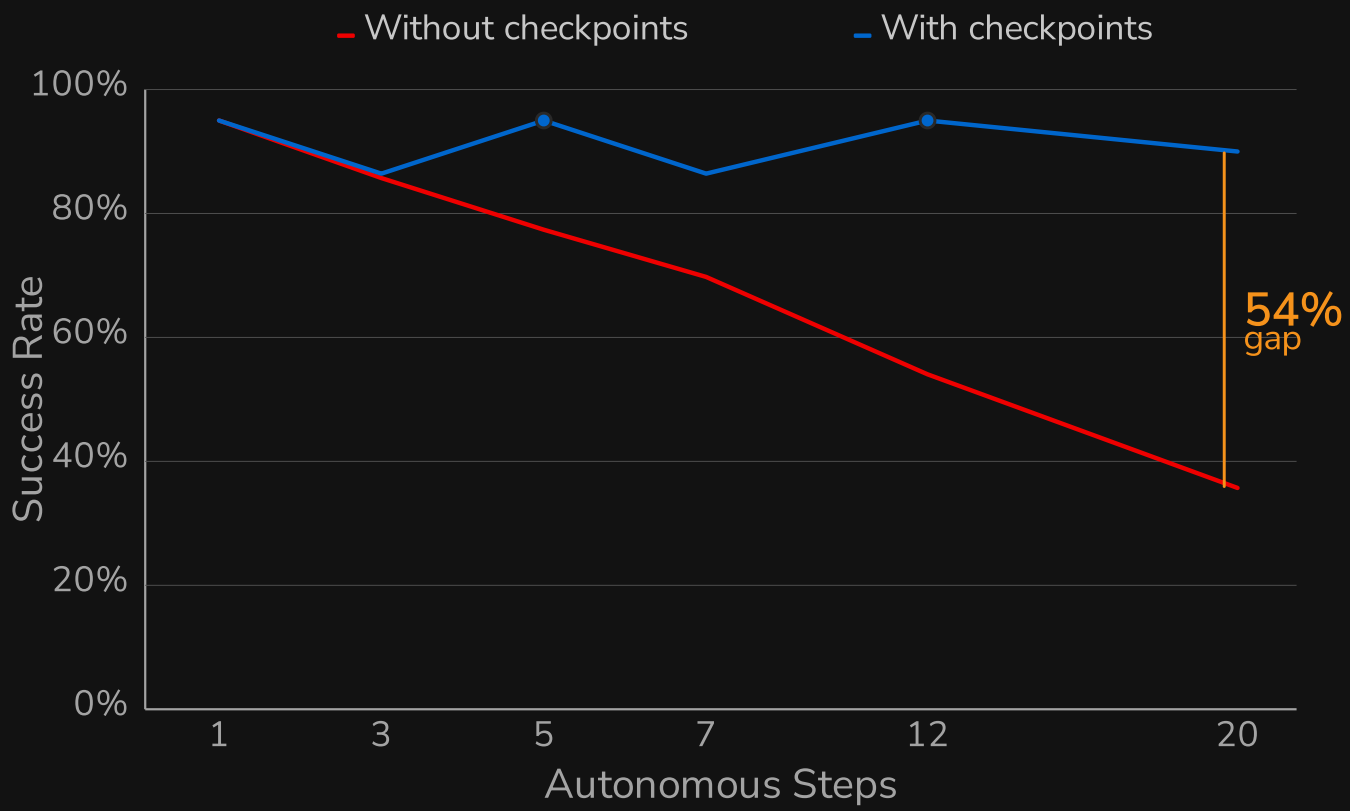
Autonomous iteration becomes a numbers game.

If each autonomous step has **95% reliability**:

95%	77%	54%	36%
1 step	5 steps	12 steps	20 steps
0.95^{1}	0.95^{5}	0.95^{12}	0.95^{20}

> "Migrate the entire frontend to React" will fail. Every step building on an error makes recovery harder.

Checkpoints are critical!



ACT 3

So What Was My Journey?

This wasn’t a planned curriculum…

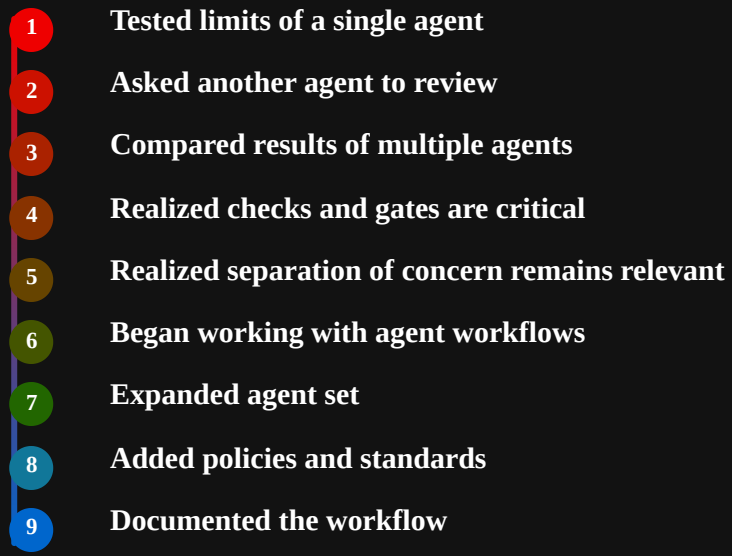
It was necessary discovery

through LOTS of learning and trial-and-error.

I started where you probably did & I’m definitely not an expert.
That is to say, my goal is to learn with others.

9 steps ~6 months MANY mistakes

Hands-On Progression



Step 1: Tested Limits of a Single Model Session

I started just like you probably did. Fired up a session, asked it to build something.

It *was* impressive -- until it wasn't.

What worked

- Small, well-defined tasks
- Code with clear patterns
- Standard library usage

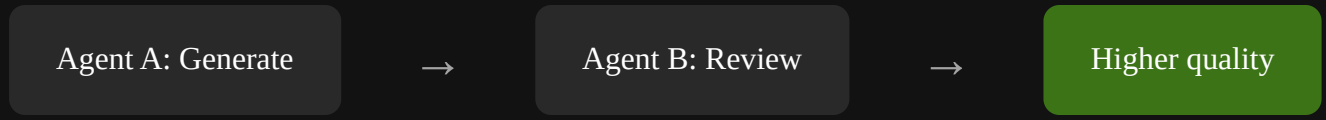
What didn't

- Large, ambiguous requests
- Cross-cutting concerns
- Anything needing project context

Step 2: Asked Another Model Session to Review

The first time I had one AI review another's work, it caught real problems, but not all of them.

> That was the moment I realized my focus shouldn't be **generation**, but **verification**.



Step 3: Compared Multiple Model Session Outputs

What if I document a prompt with results from various models and compile?

Different models, different strengths. Not all agents are equal.

This explained the need for **specialized prompting** – matching the right model and prompt to the right task.

Step 4: Checks and Gates Are Critical

> So, I had an AHA! moment – now what do I do with that concept?

What do we know…

- Without gates, each step compounds errors.
- With gates, we can reset the chain and catch problems earlier.
- The cost of a checkpoint is minutes, but the cost of uncaught compounding is days.

Tie it together… specialized prompting with gating!

Step 5: Separation of Concern is bigger than code

> The product manager shouldn’t make architecture decisions, be it human or AI agent.

Real example -- product plan deliverable started specifying technology choices, dependency maps and user stories. The agent was doing project management inside a product document.

Specialized prompting & agents is good, but at some point, context size becomes a concern. We need room for reasoning! The solution changed from better prompting to better scope discipline.

Before We Go Further: What is an Agent?

An **agent** is an AI model instance given:

- A **role** — a system prompt that defines what it does and how it behaves
- **Tools** — the ability to read files, write code, run commands, search the web
- **Constraints** — boundaries on what it can and cannot do
- **A task** — a specific, scoped piece of work with a verifiable exit condition

> An agent is **not** a chatbot. A chatbot answers questions. An agent **takes actions** — it reads your codebase, writes code, runs tests, and iterates on the results.

Quantifying Constraints

Context Horizon

3-5 files

If a task touches more than 5 files, it's over-scoped. More context dilutes attention, not improves it.

Deterministic Exit

pass / fail

The agent's goal is not "finish the task" but "pass the check." Every chunk needs a verification script.

One-Hour Rule

≤ 1 hr

Long sessions cause "context drift" -- the agent forgets instructions or gets confused by prior outputs.

Bad vs. Good Exit Conditions

Bad (Subjective)	Good (Machine-Verifiable)
"Refactor the code to be cleaner"	`ruff check src/` exits 0
"Implementation is complete"	`pytest tests/unit/test_foo.py` passes
"Endpoint works correctly"	`curl -s localhost:3000/health | jq .status` returns `"ok"`
"Code follows conventions"	`npx tsc --noEmit` exits 0

> If you can’t define a machine-verifiable exit condition, the task is probably underspecified.

Back to the Journey

So where were we?

We had specialized prompting, gating, and separation of concern. We've stated what an agent is and what keeps it on the rails.

My next question: what happens when I put multiple agents together?

Step 6: Agent Workflows

Multi-agent orchestration. Sequencing matters.

> If I chain a project manager, architect, and engineer together, can I get better results?

What do you mean 'chain' together?

- **Sequential execution** — tasks execute in strict order, output feeds the next
- **Parallel fan-out** — independent tasks run concurrently, then sync
- **Review gates** — mandatory review before proceeding
- **Iterative loops** — profile, fix, verify, repeat until targets are met

Step 7: Expanding the Agent Set

> This is where your journey has to get personal. What roles fit your need?

It depends on the use case!

I've settled on 17 agents -- each with a defined role, model tier, and tool set:

- Product Manager
- Requirements Analyst
- Architect
- Tech Lead
- Project Manager
- Backend Developer
- Frontend Developer
- Database Engineer
- API Designer
- Code Reviewer
- Test Engineer
- Security Engineer
- Performance Engineer
- DevOps Engineer
- SRE Engineer
- Debug Specialist
- Technical Writer

Step 8: The "Constitution"

Policies and standards every agent reads before doing anything.

We need to deal with commonalities.

- Architecture patterns
- Code style & conventions
- Security baseline
- Testing standards
- Error handling
- API conventions
- Review governance

Step 9: The Scaffold

I needed a consistent, reusable template.

The rules, agents, workflows, and conventions could be packaged as a project scaffold that I could adapt as required – a powerful agent network preloaded with my common needs (AI policies, code style, procedures, etc.)

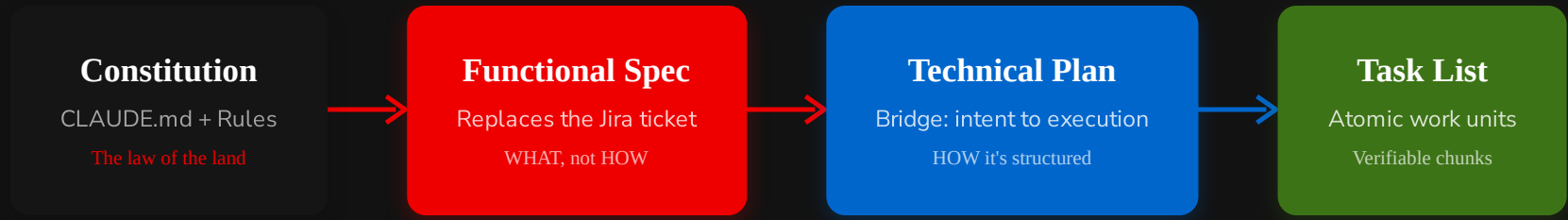
> **Reminder: I am no expert. I have success with this formula, but YMMV!**

ACT 4

Spec-Driven Development

Where you getting this crap, Jeremy?

The Artifact Chain



Each artifact **constrains** the next. Docs elevated to code status.

The Human Gate

The most critical intervention point is plan review, not code review.

Correcting a bad plan	Refactoring bad code
Minutes	**Days**

Changing perspective to focus on review is the fundamental shift. First-class citizen, no longer a simple validation.

Scope Discipline

Each layer stays in its lane. Product says WHAT, architecture says HOW the system is structured.

Product Scope (correct)	Architecture Leak (violation)
"Document storage with retrieval"	"MinIO S3-compatible object storage"
"Real-time chat responses"	"SSE streaming" or "WebSockets"
"Workflow orchestration with checkpointing"	"LangGraph with PostgresSaver"
"16 features organized by priority"	"16 epics with dependency maps and phase assignments"

> When a product plan starts specifying technology or breaking work into epics, the agent has escaped its scope. The fix isn't better prompting – it's clearer boundaries.

Constitution in Practice

What goes in the "law of the land" – what every agent reads before doing anything:

CLAUDE.md
Project context, goals, constraints, key decisions

architecture.md
Monorepo structure, package dependencies

code-style.md
Naming, formatting, import order

security.md
Input handling, auth, transport, secrets

testing.md
Coverage targets, naming, isolation

error-handling.md
RFC 7807, status codes, error hierarchy

api-conventions.md
REST design, pagination, versioning

review-governance.md
PR size, anti-rubber-stamping, review gates

agent-workflow.md
Task chunking, context engineering

ACT 5

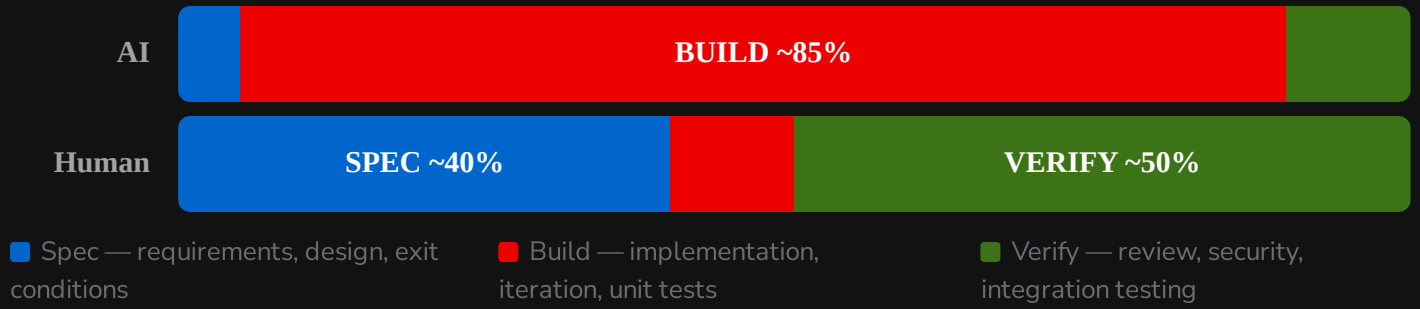
Thinking about and communicating concepts

I don't have answers, just suggestions.

Bolts, Not Sprints

Scope-boxed, not time-boxed. A 2-hour bolt for a bug, a 3-day bolt for a feature.

Syncs replace stand-ups: "What is the agent stuck on?" and "Is the plan valid?" – not "What are you working on?"



Role Evolution

Engineers

Become: AI Supervisors

- Take a spec, prompt the agent
- Rigorously verify the output
- Learn by reading AI code and finding flaws

Seniors

Become: Specifiers + Reviewers

- Specify features precisely
- Review results critically
- Ensure agents get quality context

Principals+

Become: Process Architects

- Write and maintain the constitution
- Debug the AI process
- Ensure high-quality context feeding

> The conversation shifts from "How do I write this loop?" to "Is this the right design pattern for this service?"

New Metrics

Metric	Signal
Code survival rate	% of AI output still in codebase after 3 months
Review-to-coding ratio	Was ~1:4, now should be 1:1 or 2:1

Less focus on...

- **Lines of code**
  AI inflates this to meaninglessness
- **Velocity in story points**
  Measures generation speed, not quality

Governance: Three Rules

Three rules to maintaining our mental model:

01

Gated Human Review

Progress stops until concensus is reached betweeh human and machine. No rubber stamping.

02

Small PRs Only

Hard size limits. Reject agent-generated large changesets. ~400 lines max for meaningful human review.

03

"Explain It to Me"

Engineer must be able to explain logic in conversation (no model). PR's should be rejected if unable -- full-stop. Hold yourself accountable, rubber ducking helps.

These aren’t bureaucracy – they’re the artificial friction that prevents engineers from becoming passengers.

BREAK TIME!

Stretch, Refill, Recharge

ACT 6

Same Prompt, Different Process

What does each level of rigor actually produce?

The Prompt

> "I’m building a SaaS platform. Product wants users to be able to schedule reports — pick a report type, set a cadence, and have it delivered to their team automatically. Help me plan and build this feature."

Same prompt, given to four different setups. No extra guidance, no follow-up.

Gemini 3 Pro

Produced: 3 planning documents (README, plan, architecture)

- Went straight to architecture
- Named specific tools (BullMQ, SendGrid, Puppeteer) without asking what's in the stack
- Included SQL schemas with indexing strategy
- Flagged open questions – at the end

Strengths

- Practical distributed systems thinking
- Security-aware (RLS, pre-signed URLs)
- Honest about what it doesn't know

Gaps

- Zero clarifying questions asked
- No API contracts (request/response shapes)
- Missing DST, month-boundary edge cases

Claude Sonnet 4.5

Produced: 4 documents (~2,000 lines) including a decision guide with code examples

- Jumped to architecture + implementation details
- Included Redis commands, S3 lifecycle policies, retry delay arrays
- Proposed 6-phase roadmap (15-21 weeks)
- Put UI in Phase 5 (last)

Strengths

- Most comprehensive single-pass output
- Practical code examples save implementation time
- Risk-aware with mitigation strategies

Gaps

- Zero clarifying questions asked
- Over-specified before validating requirements
- Recipients as JSONB (data model problem)
- UI last -- users can't validate until Phase 5

Claude Opus 4.6

Produced: 1 comprehensive architecture doc (447 lines)

- Jumped to architecture, but flagged 5 assumptions and 6 open questions explicitly
- Included state machines, risk register, phased rollout
- More focused – half the output of Sonnet, more strategic depth

Strengths

- Explicit about what it assumed vs. what needs answers
- Operational thinking (monitoring, SLIs, capacity)
- Timezone handling as day-1 concern, not afterthought

Gaps

- Still didn't ask questions -- just flagged them
- Skipped requirements phase entirely
- Assumptions not validated before designing

Opus Reviews Sonnet’s Work

What happens when we add one review gate? Opus reviewed Sonnet’s plan and found:

Problems Found

- Zero questions asked -- "most serious mistake"
- Recipients stored as JSONB -- should be a proper table
- UI in Phase 5 -- users can't validate until the end
- Timezone handling deferred to Phase 3 -- should be foundational
- Implementation details masquerading as planning

Opus's Corrections

- Identified 7 questions that should have been asked first
- Proposed proper recipients table with foreign keys
- Moved UI to Phase 1 -- validate with users early
- Made timezone a day-1 data model concern
- 3 authorization design options with trade-offs

> One review pass caught structural problems that would have taken days to refactor in code.

My Scaffold System

Produced: 12 documents across 3 review gates

1. Product Manager writes product plan
2. Architect, API Designer, Security Engineer review it
3. Architect designs system architecture
4. Security Engineer, API Designer review it
5. Requirements Analyst writes requirements
6. Product Manager, Architect review them

What Changed

- Product plan has zero technology names
- Architecture stays in its lane
- Requirements have testable acceptance criteria
- 8 reviewer passes -- no rubber-stamping
- Process indicates a technical design still to go

Still Not Perfect

- No stakeholder interview recorded
- Optional resolutions via human review or architect
- Report generation is still a stub

The Comparison

	Gemini 3 Pro	Sonnet 4.5	Opus 4.6	Scaffold
Artifacts	3 docs	4 docs	1 doc	12 docs (4 primary + 8 reviews)
Questions asked	0	0	0 (flagged 11)	Resolved through review gates
First action	Architecture	Architecture + code	Architecture	Product plan
Scope discipline	Mixed	Mixed	Mixed	Clean lane separation
Review gates	0	0	0	3 gates, 8 passes
Tech in product scope	Yes	Yes	Inherited	No

Every model produced useful output. The difference is how many decisions were made *for* you without asking – and how many of those could or would have been wrong.

The Takeaway

The models aren't the problem. The process is the differentiator.

Solo

Model makes every ambiguous decision for you. You find out later which ones were wrong.

+ Review

A second model catches structural problems. Cheaper than refactoring code.

+ Process

Decisions surface at the right layer. Each artifact constrains the next. Humans gate progression.

ACT 7

Show & Tell

My scaffolding

ACT 8

Wrap-Up & Adoption Path

10 minutes

Ease Into It

Don’t try the full workflow from day one. Build trust incrementally.

1. **Start with plan mode**
   Have the AI plan before it codes
2. **Try two-agent review**
   Have a second agent review the first's output
3. **Write a spec before prompting**
   Even a short one dramatically improves output quality
4. **Add policies as you learn**
   Standards and conventions emerge from what goes wrong
5. **Full SDD cycle when ready**
   The scaffold is there when you need the complete workflow

Resources

References

- **Agent Scaffold Repository**
  The template for AI-native projects
- **AI-Native Team Playbook**
  docs/ai-native-team-playbook.md
- **AI Compliance Checklist**
  docs/ai-compliance-checklist.md
- **Google Spec Kit**
  External reference for spec-driven approaches

Closing

Use case matters. My workflow is full-on concept design and documentation. Yours might differ.

That’s fine.

> It’s all about **gating for understanding and guidance**. The tools change. The models improve. But the need for humans to understand what they’re shipping never goes away.

Q&A

Questions?

My questions for you:

What's one thing you'll try differently this week?
What comes after that?

You got this.