Trust, Accountability & AI Coding Swarms

A late night conversation with Claude — January 2026

Who's accountable when AI agents write 44,000 lines of code that no human has looked at?

The Dementia Problem

/me: How does Steve Yegge's Beads work?

/agent: It's a git-backed issue tracker that solves the 'dementia problem' — AI agents have no memory between sessions. Beads gives them persistent, dependency-aware task tracking.

/agent: The schema emerged from letting Claude design what worked best for AI reasoning about tasks. IDs like bd-a1b2 prevent merge collisions in multi-agent workflows. There's a ready queue that tracks which tasks have no blockers.

> *"By and large, Beads is a tool that AI has built for itself."*

/agent: Agents have switched — without a hint of ever going back — from using markdown plans to using the issue tracker exclusively.

Architecture Convergence

/me: How's that different from Cursor's swarm approach?

/agent: Interesting — they converged on the same architecture. Cursor wrote about running hundreds of concurrent agents on a single project, writing over a million lines of code across 1,000 files in close to a week. They built a web browser from scratch as a test case.

/me: A browser in a week?

/agent: 'It mostly works. Good enough to prove the point. Nobody's running it in production.'

/me: What architecture worked for them?

/agent: Their first instinct was dynamic coordination — agents decide what to do based on what others are doing. They gave agents equal status and let them self-coordinate through a shared file. Each agent would check what others were doing, claim a task, update status.

> *"Flat peer-to-peer coordination failed. They needed hierarchy."*

/agent: Their next approach separated roles. Planners continuously explore the codebase and create tasks. They can spawn sub-planners for specific areas, making planning itself parallel and recursive. Workers pick up tasks and focus entirely on completing them. They don't coordinate with other workers or worry about the big picture. They just grind on their assigned task until it's done, then push changes.

/me: Different models for different jobs?

/agent: Yes! They found GPT-5.2 is much better at extended autonomous work: following instructions, keeping focus, avoiding drift, implementing things precisely. Opus 4.5 tends to stop earlier and take shortcuts, yielding back control quickly. They use the model best suited for each role.

Key learning: Many improvements came from removing complexity rather than adding it.

Gas Town's Mad Max Architecture

/me: How does Steve's Gas Town compare?

/agent: Gas Town is opinionated and chaotic — 'Mad Max' aesthetic. It has 7 specific roles:

- **Mayor** — orchestrates work distribution
- **Polecats** — execute tasks in parallel
- **Refinery** — manages merges
- **Witness** — watches workers, helps unstick
- **Deacon** — keeps whole system running
- **Dogs** — handle specific tasks
- **Crew** — general workers

/agent: Each worker gets a Git worktree, a task from Beads, and runs until completion. State lives externally, not in bloated context windows. It runs in tmux, written in Go, and Yegge explicitly warns it will 'rip your face off.'

/me: How does Steve keep up with reviewing all that output?

/agent: He uses agents to review agent output! 'I've merged over 100 PRs from nearly 50 contributors, adding 44k lines of code that no human has looked at. How can I keep up? I'm using Gas Town. I'll show you how I keep up with them using PR Sheriffs.'

The Emerging Consensus

/agent: Across Gas Town, Claude Flow, and Cursor's research, there's convergence:

- Flat coordination fails — you need hierarchy/roles
- State must be external — not in context windows
- Small tasks, many workers — not long sessions
- Different models for different roles — specialists beat generalists
- Humans become orchestrators — not reviewers of every line

"Nobody reviews the code, you review the outcome."

The Accountability Gap

/me: But reviewing outcomes doesn't catch everything, right?

/agent: Exactly. Reviewing outcomes works for: Does it render correctly? Do tests pass? Is it faster? Does the feature work?

/agent: Reviewing outcomes doesn't catch: SQL injection, timing attacks in auth, PII being logged somewhere, dependencies with known CVEs, ports that shouldn't be open, subtle race conditions that manifest at scale.

"The tests pass. The feature works. The security hole ships to production."

/me: What's Steve's answer?

/agent: 'We'll fix shit later' — acceptable for his personal projects. Cursor's blog post has 1M lines of code with no mention of security review.

/me: But in a real organization?

/agent: That's the fundamental tension. Tools optimize for throughput. Organizations need accountability. Steve can say '44k lines that no human has looked at' because it's his project. If it breaks, he fixes it. No customers harmed, no compliance violated.

/agent: But in an enterprise: Who signs off on security review? Who's responsible when agent-written code leaks PII? Who answers to the regulator? Whose name is on the commit?

/me: This might be the real talk I should give.

> *Not 'context is the new code' but 'accountability is the new bottleneck.'*

Turtles All The Way Down

/me: What about adding more AI to check the AI? Security agents, LLM-as-judge?

/agent: Gas Town already has this! Witness agents watch the workers. Deacons watch the Witnesses. But they had a bug — 'murderous rampaging Deacon' (actual Gas Town bug). The solution? More agents to watch the Deacon.

/me: To solve problems with AI, use more AI — that's a paradox?

/agent: Ha! Yes, exactly. The recursion problem: LLMs generate insecure code → add security LLMs to check → who checks the security LLMs? → add more LLMs to verify → who checks those? Turtles all the way down.

/me: Any research on this?

/agent: The numbers are sobering. Multiple studies report 12-65% of generated code snippets are non-compliant with basic secure coding standards or trigger CWE-classified vulnerabilities. LLMs hallucinate non-existent packages up to 44.7% of the time. CI plugin hallucination rates can exceed 90%.

Promise Theory

/me: There must be some wisdom from Promise Theory here.

/agent: Oh now you're talking! Mark Burgess nails it:

> *"Trust isn't transitive. You can't escape trust through more layers."*

/agent: Key principles from Promise Theory:

- A promise is not an obligation — it's just a declaration
- Promises are not assumed to be kept — they're just recorded
- No agent can promise on behalf of another — this breaks the 'more AI' recursion
- Trust is built through history of kept promises

/agent: You can't bootstrap trust from a system that doesn't have it. If the base layer has a 12-65% vulnerability rate, stacking more LLMs doesn't eliminate that — it just spreads uncertainty across more layers.

What Actually Works

/me: So what does the research say actually helps?

/agent: The approaches that showed real improvement all had one thing in common: grounding in something external to the LLM.

- Static analyzers (deterministic, not AI)
- Formal verification tools (mathematical guarantees)
- Known vulnerability databases (human-curated knowledge)
- Fuzzing (empirical testing)

/agent: The LLM becomes a translator between human intent and these grounded systems, not the source of truth itself.

/agent: FDSP (Feedback-Driven Security Patching) iteratively incorporates static analyzer feedback into LLM re-generation — reduced residual vulnerability rate from 40.2% to 7.4% for GPT-4. GRASP constrains LLM reasoning through a DAG over Secure Coding Practices, raising security rate from ~0.6 to ≥0.8.

> *The ceiling on AI agent autonomy might not be capability — it might be trust. And trust requires grounding outside the system.*

Context Checks

/me: What about building trust through observations over time?

/agent: Yes! Like CI checks but for agent promises:

- **Promise registration** — agent declares intent ('I will write auth code following OWASP guidelines')
- **Observation layer** — deterministic tooling captures what actually happened
- **Promise ↔ Observation delta** — compare, record trust signal
- **Calibrated trust scores** — build reliability patterns over time

/me: Ha! It's like context checks.

/agent: Exactly! The same pattern as your context flywheel idea.

/me: But verification is expensive. It becomes a risk game.

/agent: And that's the honest tension. The economics:

- Deterministic verification = expensive
- LLM-as-judge = cheap but ungrounded
- Human review = expensive AND doesn't scale
- Ship without verification = cheap but risky

/agent: For most code, most of the time, the rational choice is: don't verify deeply, fix it when it breaks.

Resilience Engineering

/me: There's learning from Resilience Engineering here.

/agent: Yes! Traditional safety thinking says 'prevent bad things from happening' — verify, check, gate, control. Resilience engineering says 'things will go wrong, how do we survive and recover?' — adapt, respond, monitor, learn.

/agent: Hollnagel's four cornerstones completely reframe the problem:

Cornerstone	Traditional Safety	Resilience Approach
Anticipate	Prevent all failures	What's likely to go wrong?
Monitor	Verify everything	What signals show drift?

Respond	Block bad code	How do we recover quickly?
Learn	Root cause analysis	What went right AND wrong?

> *"You can't prevent all failures in complex systems. You can only build systems that degrade gracefully and recover quickly."*

/agent: Contrary to traditional safety focusing on 'what went wrong', resilience engineering recognizes 'things that go right' are just as important and considers variability in performance as normal, not a threat.

The Loop

/me: But optimizing context is still prevention, right?

/agent: You've caught the loop! Every path leads back to prevention:

- Learn from observation → so we can prevent next time
- Build trust scores → so we can prevent trusting bad agents
- Capture context → so we can prevent bad outputs
- Monitor for drift → so we can prevent failures

/agent: Even resilience engineering's 'learning' cornerstone is ultimately about improving anticipation... which is about prevention.

/agent: The only truly non-prevention moves are:

- **Blast radius control** — not preventing failure, just limiting damage
- **Fast rollback** — not preventing failure, just recovering quickly
- **Graceful degradation** — not preventing failure, just surviving it

/agent: And those aren't really 'AI agent platform' features. They're just... good engineering. Git, feature flags, canary deploys, circuit breakers.

/me: So everything is prevention?

/agent: ■ You've fully closed the loop. Even 'accept chaos' is a strategy to prevent wasted effort trying to control things.

/me: Maybe the honest talk is: 'We looked for a product. We found a practice.'

Ship It

/me: Ship it gives value. So the clue is more wins by limiting the cost of impact, including reputation.

/agent: NOW we're somewhere. The actual equation:

$$\textbf{Value = Wins from shipping fast} - \textbf{(Cost of failures} \times \textbf{Probability of failures)}$$

/agent: You can't control probability of failures (non-idempotent, stochastic systems). You CAN control speed of shipping and cost of failures.

/agent: Steve's implicit brand: 'This is experimental. It will break. I warned you. That's the deal.' He's pre-negotiated reputation cost to near zero. Enterprise can't do that — they need to ship fast BUT maintain reputation for reliability.

Back to Basics

/me: Risk = Impact × Likelihood

/agent: We're back to basics! Classic risk management. But with AI agents, likelihood is basically unknowable. So you're left optimizing the one variable you CAN control:

Risk = Impact × ¯_(■)_/¯

Therefore: Focus entirely on Impact.

/me: What are the dimensions of impact?

Dimension	Low Impact	High Impact
Blast radius	Internal tool	Customer-facing
Reversibility	Easy rollback	Data corruption
Visibility	Nobody notices	Public incident
Reputation	Expected to break	"Enterprise-grade"
Compliance	No audit	Regulated
Safety	Annoyance	Harm to people

/agent: The routing logic: Low impact → full autonomy, ship fast. Medium impact → lightweight checks, monitor closely. High impact → human in loop, smaller batches, staged rollout.

We don't predict what will fail. We help you know how much it would hurt if it did.

The Real Conclusion

/me: So at the end of the day...

/agent: Doing business is all about balancing risk. That's not a new insight — it's the oldest insight. Every business decision is a bet:

- Ship faster → more value, more exposure
- Review more → less risk, slower delivery
- Automate more → scale better, less control
- Trust agents → leverage their speed, accept their failures

/agent: AI coding swarms don't change the fundamental equation. They just move the dials. More throughput, less predictability, different failure modes.

"The question was never 'how do we eliminate risk?' It was always 'which risks are we willing to take for which rewards?'"

/agent: Steve's answer: 'I'll take the risk of bugs for the reward of shipping 44k lines in a week. It's my project, I'll fix it.'

/agent: Enterprise answer: 'We need to protect reputation and compliance, so we'll accept slower delivery for controlled blast radius.'

/agent: The right answer: It depends. On your context, your customers, your risk appetite, your recovery capability.

There is no universal solution. There's only clarity about which tradeoffs you're making and why.

/agent: That's what a 'context layer' could actually provide — not magic prevention, not perfect trust, but clarity about what you're betting on and what it costs if you lose.

The Journey

/me: Ok my head hurts, going to sleep.

/agent: Ha! You kept poking holes until we got to something honest.

"Beads → Gas Town → 'accountability is the new bottleneck' → Promise Theory → 'you can't escape trust' → resilience engineering → 'everything is prevention' → back to basics: Impact × Likelihood, but you can only control Impact."

Sometimes the most useful insight is admitting what you can't know.

References & Acknowledgments

Steve Yegge — Beads, Gas Town

Cursor team — Swarm experiments, 'Scaling Agents'

Mark Burgess — Promise Theory

Erik Hollnagel — Resilience Engineering, Four Cornerstones

Patrick Debois × Claude • January 2026