Spencer Burleigh

Constraints beat prompts for AI code

Feb 13, 2026 • spencerburleigh.com/crosscheck

Coding with AI has an entropy problem

A few hours into coding with AI, a pattern shows up. The code looks clean. Abstractions make sense. Tests pass. And yet under concurrency, your rate limiter collapses because the counter isn't atomic.

Here's the formula nobody talks about: **failure= error rate × variance**

Coding with AI massively increases variance. Even if error rate stays constant (big if), failures scale dramatically with output. After a few hours, you have 200 files of mostly okay spaghetti code and an AI who responds to your prompts with things like:

```
● You're absolutely right — I was making up nonsense.
```

That asymmetry is the core problem.

AI systems are very good at generating plausible structure: clean abstractions, proper separation of concerns, thoughtful API boundaries. They're weaker at adversarial edge cases, enforcing invariants, state transitions under load, and remembering decisions made hundreds of thousands of tokens ago.

I've spent much of the past month coding with agents and building a set of open source tools called CrossCheck [a] to push development further. What follows is an overview of my philosophy and implementation.

74,424	+4,315,952/-1,209,193	47416	27	2756.4
MESSAGES	LINES	FILES	DAYS	MSGS/DAY

This is a governance problem, not a prompting problem

Most advice about coding with AI focuses on prompts: be more explicit, ask for tests, review carefully.

That's behavioral control. But behavioral control isn't deterministic and decays as context grows.

So the question shouldn't be "how do I prompt better?" but how do I design a repo where the compliant path is the path of least resistance?

This means converting norms into invariants.

Not "please run tests." Instead, require a test script to pass or git refuses to proceed.

Not "NEVER commit secrets." Instead, block secret patterns at commit time.

Not "don't merge your own code." Instead, set git branch protections to make it impossible.

Instructions are advisory. constraints are real.

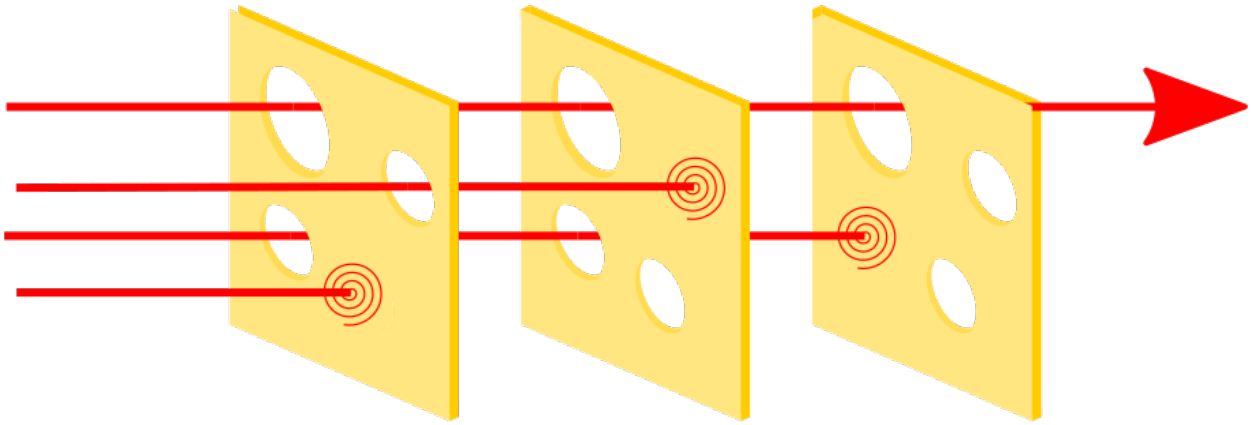
Uncorrelated layers reduce failures

If Claude writes the code, writes the tests, reviews the diff, and evaluates the security scan output, you do not have four layers. You have one reasoning process with the same blind spots wearing four hats.

If a model misunderstands a concurrency primitive, that misunderstanding propagates through implementation, test design, and review commentary. Everything agrees. Everything is wrong.

Just like investing, a couple of uncorrelated checks is much more valuable than many correlated ones.

Aviation's "swiss cheese model of accident causation"



In the early jet age, planes had the nasty tendency to get lost and fly into mountains. The fix wasn't "train better pilots." It was layered, uncorrelated safety systems:

Pilots crosscheck independent instruments. Critical phases of flight operate under “sterile cockpit” rules where nothing unrelated is discussed. Aircraft carry multiple navigation systems: GPS, inertial reference, radio beacons that all validate each other.

Most importantly, modern aircraft connect navigation to a terrain database. If the projected flight path intersects terrain, the cockpit doesn’t politely suggest a correction. It shouts:

“Terrain… pull up.”

After incidents, black boxes reconstruct what happened; even near-misses feed back into procedures.

The result: planes are ~150x safer than driving and crashes are much “weirder” results of many errors.

Governance layers for AI code

AI development needs the same architecture as aviation safety: multiple layers, each with different strengths and weaknesses. I’ve found a few things to dramatically decrease my error rate:

1. **Execution-level deny list.** Destructive commands and bypass flags are permission-blocked. The agent literally cannot execute rm, sudo, –no-verify, etc. We mitigate -rm with a garbage folder the agent moves files to and the user empties occasionally.
2. **Git hooks.** The enforcement boundary: secrets, format, tests, protected paths.
3. **Tests.** Behavioral verification, invariant assertion from planning docs. Code must do what it claims under defined conditions and adversarially test against itself. I’ve also tried enforcing a CI but have been blowing through my GitHub account’s quota :/
4. **Multi-model review.** A model from a different lab reviews the code. Different training data, different reward tuning, different strengths and weaknesses.
5. **Server-side branch protection.** Identity-enforced approval. The agent account that pushes code cannot approve its own PR. Squash-only merges.
6. **Give humans a place to write documents the AI can’t edit.** I’ve found AI has a bad tendency to change documents that I write. I now have a section of my repos dedicated to human produced text that the AI can use as a “golden reference” and should never change. This folder is protected by a git hook.

A bug that slips past tests may not slip past a second model trained on different distributions. A secret missed by one pattern matcher may not bypass a deny list. A sloppy review cannot bypass server-side merge rules.

Git hooks as zero-trust infrastructure

```
● Update(TROUBLESHOOTING.md)
  └ Removed 13 lines

    **Exact error message:**

183 -⛔ Pre-push hook failed
184 -
    ❌

187 -
188 -Files with conflicts:
189 -  src/auth.ts:45: [conflict start marker]
190 -  src/api.py:128: [conflict separator]
191 -
192 -Actions:
193 -  1. Resolve merge conflicts
194 -  2. Remove conflict markers (start, separator, end)
195 -  3. Test the merged code
196 -  4. Commit the resolution
197 -  5. Retry push

    **Resolution:**
```

AI agents regress to the path of least resistance (commit directly to main with no tests). Git hooks convert norms into invariants. They run automatically at commit, push, and merge boundaries. If a change violates policy, the repository itself refuses to move forward. Hooks enforce procedure, secrets scanning, format checks, and merge policy at commit and push boundaries while reminding the AI what it should be doing.

For solo developers, the permissions aren't really designed to support vetting the code "you" submit so I opened a second GitHub account just for my agents. This ensures the agent account that pushes code can't approve its own PRs and codeowners adds another gate.

The result: agents are treated like software engineers at a well permissioned company. They have total freedom on feature branches and a very scoped blast radius from others' contributions, git history, and production systems.

AI reviews AI code

If AI writes your code, the first reviewer should not be you. It should be another model. This matters because a second model reduces correlated error, and frees you to focus on higher-leverage decisions.

Every three PRs in my setup trigger a git hook enforced repo-wide assessment for bugs and security regressions.

```
● Codex is actively analyzing the repo. Let me wait for it to
  complete and provide its assessment:

● Task Output b5a7d99
  Initiate Codex comprehensive repo assessment
     Waiting for task (esc to give additional instructions)

✳ Musing… (3m 28s · ↓ 474 tokens)
  └ □ Deduplicate documentation (93 docs, 17 duplicate grou…
    □ Progressive Ruff quality cleanup (2380 issues)
    ✔ Fix malformed CI YAML in ci-hybrid.yml
    ✔ Resolve CI workflow conflicts and duplication
    ✔ Fix integration test regressions (13 failed, 17 error…
    ✔ Remove deprecated blog/feed architecture paths
    ✔ Clean repo hygiene - remove non-product artifacts
    ✔ Add run-full-tests label to PR #180
    ✔ Fix documentation timestamp compliance (64 docs missi…
    ✔ Document breaking change for check-blogs removal
     … +5 completed
```

Only scale dependably correct code

Once you have a system that writes good code, it's time to create ways for the agents to run in a development loop without your involvement. Some best practices include:

```
✳ Removing archived .venv directory… (1h 4m 29s · ↓ 3.9k tokens · thought for 5s)
  └ ✔ Fix test_podcast_pipeline.py priority constraint violation
    ■ Remove archived .venv directory from docs
    ✔ Consolidate USER_GUIDE.md (docs/ vs docs/user/)
    ✔ Consolidate TROUBLESHOOTING.md (docs/ vs docs/user/)
    ✔ Consolidate CLI_REFERENCE.md (docs/ vs docs/user/)
    ✔ Consolidate CONFIGURATION.md (docs/ vs docs/user/)
    ✔ Consolidate CONTENT_SOURCES.md (docs/ vs docs/features/)
    ✔ Submit PR for Phase 2 docs consolidation
    ✔ Audit feed-related code and config files
    ✔ Document feed pipeline consolidation plan
     … +12 completed
```

1. Give agents tools like CLIs and MCPs. Configure once, let the agent interact programmatically. As with GitHub, I create a separate account for my agents with scoped permissions.

2. Protect the model's context so it remains cohesive longer. Use CLAUDE.md but don't make it too long.
3. Parallelize with git worktrees.

```
● ✅ PR #36 MERGED! Counter: 8

  Waiting for Railway to deploy, then running final E2E test:

  Bash(echo "Waiting 45s for Railway deployment..." && sleep 45 &&
      echo "Testing if HTTPS fix deployed..." && curl -s -X POST
      https://playlistmint-production.up.railwa…)
  └ Waiting 45s for Railway deployment...
    (22s · timeout 1m)
    ctrl+b to run in background

✽ Synthesizing… (9m 52s · ↓ 3.8k tokens · thought for 2s)
```

Protect the model's context window

Context is a budget. Every rule, every file read, every test dump consumes tokens. As context grows, the model gets slower and starts forgetting what you told it. Heavy exploration should be delegated to subagents that return summaries.

Stale context is worse than no context. CLAUDE.md is loaded into every session so every line needs to earn its place. Mine is <100 lines and it's supported by more detailed rules that load via hook-based triggers rather than sitting in context hoping to be remembered 200k tokens later.

Parallelize agents with git worktrees

Parallelizing development is another multiplier on your output while coding with AI. Git worktrees let you run isolated branches each with their own agents and context windows. Spin up 3-5 features in parallel, each in its own terminal tab, each with its own agent.

I combine this with a task queue: structured files the agent processes autonomously in priority order. Create branch, write code and tests, submit PR.

Numbers 001-099 are human-created tasks. 100+ are agent-created when the agent discovers follow-up work during execution. Define five tasks, review five branches of PRs an hour later.

The key: every parallel path still routes through the same structural gates. Throughput increases without sacrificing enforcement.

A warning: I tried doing this before I implemented quality gates / multi-modal checks and it took me a few days to resolve the mess :/

What's next

```
✻ Fermenting… (1h 18m 15s · ↓ 28.4k tokens)
```

Making adoption easy

This isn't free. Setup is heavy. Token usage increases. Trivial scripts become slower. For prototypes, this is overkill. For long-lived systems, I think it's pretty essential.

To make it easier to get started, I built my tools into an open source system called CrossCheck (link below). Let me know what you think :)

What I'm still trying to solve

Long-term memory. CLAUDE.md is static. it doesn't know what the agent did yesterday or which tasks are blocked. the missing piece is dynamic state that persists across sessions: task graphs, dependency tracking, learned patterns.

UI loops. Backend code has clean, autonomous, easy to verify loops. UI doesn't. You need to see the component, evaluate it aesthetically, and decide if the layout works. Screenshot-based feedback loops exist but AI's visual judgment is unreliable. I'm urgently looking for a solution to this one. Until then all my apps will be CLIs and simple landing pages :/

Should I add a third model? Right now Claude writes and Codex reviews, but the merge-to-main step only has branch protection. Should I add Gemini? Gemini and Kimi? Where does this stop being useful?

Safer CLI access. CLIs enable loops, but many CLIs and MCPs don't have a great way to permission access between agents and humans. We need more approval gates for destructive operations and a better way to distinguish between agents and humans in a way that doesn't involve buying two seats for every piece of software.

Better agent permissions. We need more granularity on what parts of a git tree an agent should be able to access. I have strong main branch protections enforced on main but more nuance is needed between the branches different agents are individually working on (eg agent1 should be able to have ~unlimited permissions on branch-agent1 but agent2 running in parallel shouldn't have any permissions there).

How I'm thinking about myself as a developer

I've always seen myself as a builder more than an engineer, and I haven't had the identity crisis some of my friends are going through. The role of human judgment is still central. It's just shifted: less typing, more governing. Less "write this function," more "design this system."

That's empowering. People without specialized engineering backgrounds or the resources to hire specialists can now build things that used to be gated by both. You don't need to be a great engineer. You need to be a great governor of systems that create software. I've never been more excited to build.

Links

a. github.com/sburl/CrossCheck

Spencer Burleigh · spencerburleigh.com/crosscheck February 13, 2026