



Beyond the Model: What Distinguishes Effective Agentic Development

NEW RESEARCH FROM SPAN

03

Executive summary

05

Introduction

07

Methodology

09

Results

10

Result 01
Clearer prompts, cheaper code

12

Result 02
Ready environments, more autonomous output

14

Result 03
Quality stewardship up front, fewer review cycles later

16

Recommendations

22

Conclusion

24

Appendix

01.

Executive summary



Executive summary

The conversation about AI coding effectiveness tends to focus on model selection. This study examines the variables on the other side of the interaction: how clearly developers specify work, how well the development environment supports agent autonomy, and how actively developers steward AI-generated changes toward production.

We analyzed complete agent trajectories across 103 engineering teams and compared three Span evaluation scores with downstream measures of cost, autonomy, and rework.

Across the dataset:

- A one-point increase in **prompt clarity** was associated with approximately 27% **lower token cost** per merged AI-authored line.
- A one-point increase in **environment readiness** was associated with approximately 88% **higher turn yield**, measured as merged AI-authored lines per human turn.
- A one-point increase in **quality stewardship** was associated with approximately 39% **fewer review cycles** per 1,000 AI-authored lines merged.

These observational findings do not establish causality. They do suggest that some of the most important levers for effective agentic development are measurable, trainable, and within the control of engineering teams.

02 .

Introduction



What are the leading indicators of agent effectiveness?

AI coding agents are now writing a meaningful share of production code. But not all agent sessions are created equal. Some trajectories glide: the agent receives a task, works largely on its own, and ships code that survives review. Others grind through repeated clarifications, broken tooling, half-verified changes, and a code review process that redoes the work the agent was supposed to save.

The difference between those two experiences is expensive and hard to measure before the bill arrives.

The mainstream discourse offers a ready explanation for the gap: the model. Benchmark leaderboards, model release threads, and “which model should my team use?” debates dominate the conversation, as if selecting the right frontier model were the primary variable worth tuning. Far less attention goes to the other side of the interaction: how clearly the human specifies the task, and how ready the codebase and tooling are for the agent.

Our data suggests that focus is a mistake. Within our corpus, trajectories running on the same models diverged dramatically in cost, autonomy, and rework, with those differences strongly associated with properties of the prompt, the environment, and the human’s engagement.

This report is about measuring those neglected variables.

We define an **effective agent trajectory** as one that:

- Produces low-cost code, measured by token spend per line of accepted AI-authored code.
- Produces more merged AI-authored code per human turn, a measure of autonomy we call turn yield.
- Produces little rework, surviving code review with minimal back-and-forth.

Why these three? Because they map directly to two of the most expensive inputs in AI-assisted development: **human attention** and **tokens**. Every clarification, handoff, and review cycle amplifies the rate limiter of human bandwidth. And every retry, dead-end exploration, and regeneration compounds the cost of the output. A trajectory that minimizes both is one that delivers on the promise of agentic development.

But all three are **lagging** metrics. You learn the cost per line after the tokens are spent. You count the review back-and-forths after the PR has churned. What we wanted were leading indicators: signals visible inside the trajectory itself, early enough to act on.

03 .

Methodology



Agent eval scores

To find signals related to effectiveness, we developed a set of evals that examine the full interaction history between human and agent: the initial prompt, the environment the agent operated in, the tool calls it made, and how the human engaged throughout the trajectory.

The three categories that emerged as most predictive:

What you improve	What it plainly means
Prompt clarity	The task is specified clearly
Environment readiness	The repo and tooling let the agent work and self-check
Quality stewardship	The user steered the code toward standards, and steered the change toward a safe rollout

Each category is scored by an agent eval that assesses observable properties of the trajectory. Did the prompt state acceptance criteria, or did the agent have to infer them? Could the agent run the test suite, or did tooling failures force it to fly blind? Did the developer steer the output toward local standards, production safety, and legibility for reviewers and future maintainers?

Dataset

We analyzed agent trajectories from **103 engineering teams**, across 12 weeks of real-world development activity from May through July 2026. Each trajectory includes the complete interaction record: the initial prompt, every human and agent turn, tool calls and their results, the resulting diff, and the linked code review history through merge or abandonment.

We then compared the eval scores with downstream measures of token efficiency, agent autonomy, and review rework. Because these are observational relationships rather than results from a randomized experiment, we treat the scores as practically useful leading indicators, not evidence of causality.

04.

Results

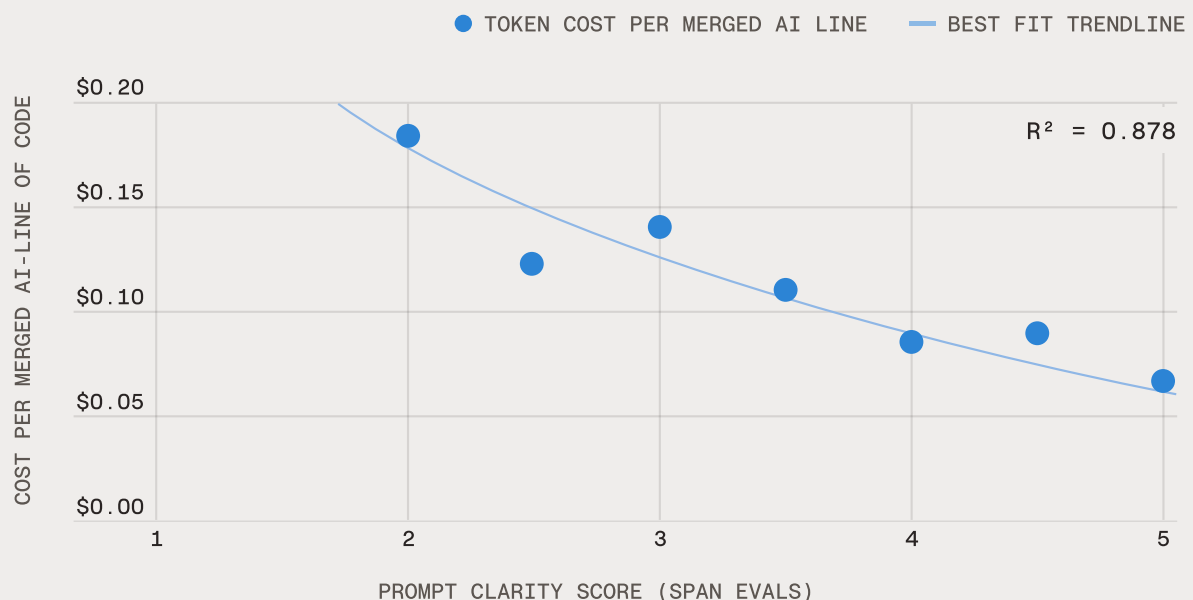


Clearer prompts, cheaper code

As prompt clarity increases, cost per AI-authored line decreases.

Trajectories that scored high on prompt clarity — explicit scope, stated constraints, concrete acceptance criteria — consumed substantially fewer tokens per line of code that ultimately shipped. The mechanism is visible in the traces: vague prompts trigger exploratory loops, wrong-direction implementations, and regeneration cycles. A clear prompt collapses the search space, and the token meter reflects it.

The impact of prompt clarity on token cost per merged AI line of code



How to read this chart

Token spend per merged AI-authored line

Total token spend during the trajectory, divided by the net lines of AI-authored code that survived to merge.

Prompt clarity score (scored from 1 to 5)

Span eval of how clearly the developer defined the task, including its scope, constraints, relevant context, and acceptance criteria.

Our analysis found a very strong relationship between Span's prompt clarity score and cost per merged AI-generated line, with $R^2 = 0.878$. Every one-point increase in prompt clarity was associated with an approximately 27% reduction in cost per merged AI-generated line.

To make this tangible, the median merged PR in the dataset included 50 lines of AI-generated code. At a prompt clarity score of 3.0, our logarithmic model estimates that PR would cost \$6.43 in token spend. At a score of 4.0, the estimated cost falls to \$4.72.

At scale, improving developers' ability to prompt clearly can materially improve token efficiency. Later in the report, we outline practical ways teams can improve prompt clarity.

Ready environments, more autonomous output

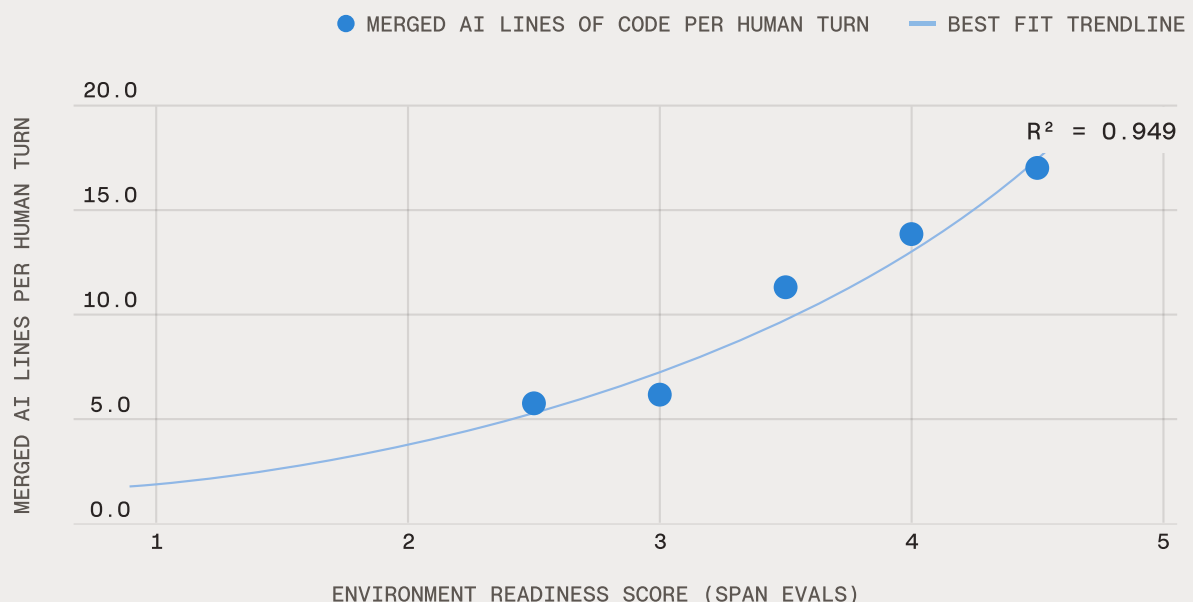
As environment readiness improves, agents produce more merged AI-authored code for each human turn.

Trajectories with the highest environment-readiness scores produced markedly more accepted output before requiring additional human input, a measure we call **turn yield**. When the repository builds cleanly, tests run out of the box, and the agent can verify its own work, it can make more progress before returning to the developer.

The bottom of the distribution tells the opposite story: the agent hits a broken lint configuration or flaky test harness, cannot distinguish its own errors from problems in the environment, and repeatedly escalates to the human.

Autonomy, it turns out, is less a property of the model and more a property of the workspace you hand it.

The impact of environment readiness on turn yield



How to read this chart

Turn yield: Merged AI lines per human turn

A measure of how much AI-authored code survives to merge for each human turn. Higher turn yield indicates greater autonomy.

Environment readiness score (scored from 1 to 5)

Span eval of how well the environment enables the agent to work and verify its own output through clean builds, reliable tests and checks, and clear signals to distinguish code failures from environment issues.

Our analysis found a very strong relationship between Span's environment readiness score and turn yield, with $R^2 = 0.949$. Every one-point increase in environment readiness was associated with approximately 88% more merged AI-authored lines per human turn.

Across environment readiness scores of 2.5 to 4.5, estimated turn yield ranged from roughly 5 to 17 merged AI lines per human turn. In more ready environments, agents produced substantially more merged output before requiring additional developer input.

The better the environment supports the agent in acting, checking, and correcting its own work, the more progress it can make before needing additional developer attention.

Note: Environment readiness scores of 5.0 were excluded from this analysis because the dataset did not contain enough trajectories at that score to support a reliable estimate.

Quality stewardship up front, fewer review cycles later

As quality stewardship improves, code review back-and-forths decrease.

Trajectories that scored high on stewardship did more than produce cleaner code.

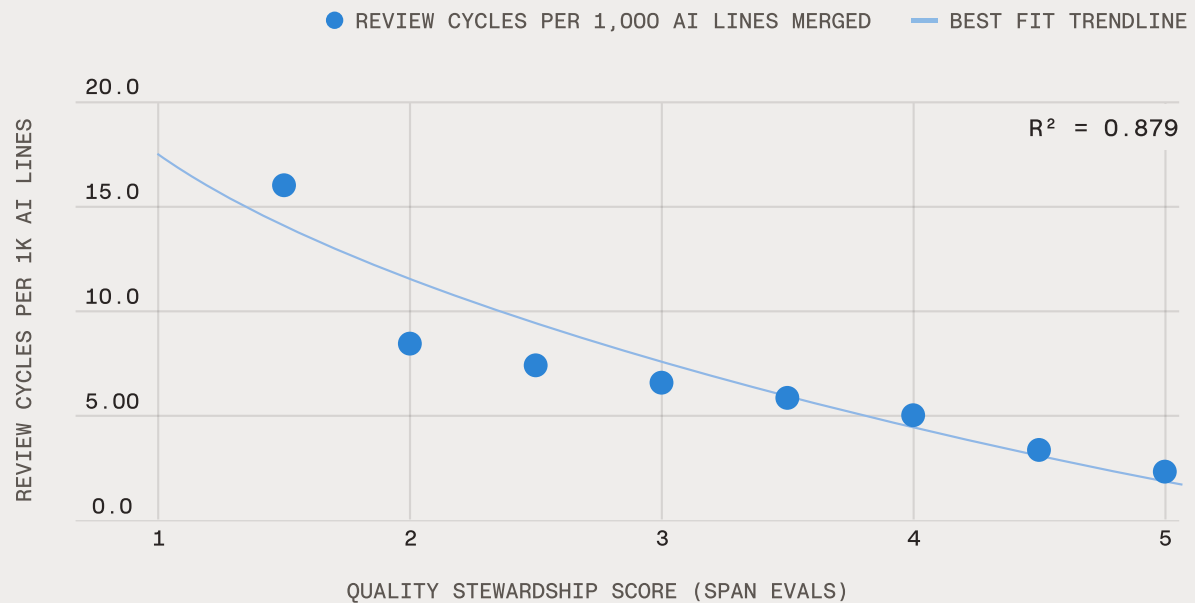
Developers pushed drafts toward the repository's local conventions, made failure handling and production constraints explicit, and considered blast radius and rollout safety before opening the PR. They also left behind the context others needed to evaluate and maintain the change, including tradeoffs, shortcuts, technical debt, deprecation decisions, and documentation where warranted.

These changes required less interpretation during review. Reviewers spent fewer cycles uncovering missing context, asking how edge cases were handled, or reconstructing why a decision had been made.

The mechanism makes intuitive sense when you consider what review comments contain: much of the churn is this work arriving late. Naming and structure nits. "This doesn't match how we do it in this module." "What happens when this is null?" "How do we roll this back?" "Where's the logging?"

Low-stewardship trajectories pushed that work downstream, where it involved more people and more review cycles. The code might have been functionally complete, but reviewers still had to surface hidden debt, missing rationale, safety gaps, or future maintenance concerns.

The impact of quality stewardship on number of review cycles



How to read this chart

Review cycles per 1,000 AI lines merged

The average number of review rounds, including requested-changes cycles and substantive comment threads, across 1,000 lines of AI-authored code merged.

Quality stewardship score

Span eval of how well the developer steered the change toward code quality, local conventions, clear failure handling, rollout safety, and the context reviewers and future maintainers need to understand it.

Span's quality stewardship score was strongly associated with the number of review cycles across 1,000 lines of AI-generated code merged, with $R^2 = 0.879$. Every one-point increase in quality stewardship was associated with approximately 39% fewer review cycles.

At a quality stewardship score of 3.0, our model estimates approximately seven review cycles per 1,000 lines of AI-generated code merged, across contributing PRs. At a score of 4.0, that estimate falls to roughly four or five cycles.

Strong stewardship does not eliminate the work required to make a change safe, maintainable, and understandable. It moves more of that work into the original trajectory, before it becomes a reviewer's problem.

04.

Recommendations



A practical field guide to more effective agentic development

The findings point to a practical conclusion: many of the levers for cheaper, more autonomous, lower-rework AI development are within the control of engineering teams.

The recommendations below translate each leading indicator into concrete practices teams can start applying now or build toward over time:

- **Improve prompt clarity** to reduce wasted exploration and token spend.
- **Improve environment readiness** so agents can act, verify, and correct their work while producing more accepted output per human turn.
- **Improve quality stewardship** to move judgment, safety, and maintainability work earlier, before it becomes review churn.

These are not rigid requirements or a single prescribed workflow. They are practical ways to improve the conditions around agentic development, starting with the changes most within reach for your team.

Improving prompt clarity

Prompt clarity is the most immediate lever. It does not require new infrastructure or tooling, only a more disciplined way of defining work before implementation begins.

The highest-scoring prompts in our corpus shared a recognizable shape: they read less like an open-ended chat message and more like a well-written ticket or specification.

REC 1.1

State acceptance criteria explicitly. Say what "done" means: which behavior should change, which tests should pass, which cases must keep working. The single biggest token sink we observed was agents inferring, and mis-inferring, the definition of done.

REC 1.2

Scope the change, including the negative space. Name the files, modules, or layers in scope, and just as importantly what's out of scope ("don't touch the public API," "no dependency changes"). Unscoped prompts invite exploratory wandering that you pay for by the token.

RECOMMENDATIONS

REC 1.3

Provide pointers, not archaeology assignments. Reference the relevant issue, design doc, error message, or exemplar file ("follow the pattern in payments/refunds.ts"). Every fact the agent has to rediscover from the codebase is token spend you could have avoided with one sentence.

REC 1.4

State constraints and conventions up front. Language version, framework idioms, error-handling style, performance or security requirements. Retrofitting conventions after generation is a regeneration cycle.

REC 1.5

Adopt lightweight prompt templates for recurring task types. A team-level template for "bug fix," "refactor," or "new endpoint" (context → goal → acceptance criteria → constraints → out of scope) turns clarity from individual skill into team habit — and makes prompt quality reviewable, the same way commit-message conventions did for VCS hygiene.

REC 1.5

Adopt lightweight prompt templates for recurring task types. A team-level template for "bug fix," "refactor," or "new endpoint" (context → goal → acceptance criteria → constraints → out of scope) turns clarity from individual skill into team habit — and makes prompt quality reviewable, the same way commit-message conventions did for VCS hygiene.

REC 1.6

Front-load the ambiguity resolution. If the task has open questions, resolve them with the agent (or a colleague) before delegating implementation. A two-minute clarification chat is cheaper than a wrong implementation.

Improving environment readiness

Environment readiness usually requires more team-level investment, but many improvements are straightforward: reliable setup commands, runnable tests, fast feedback loops, and clear repository guidance.

The trajectories with the highest turn yield ran in repositories where agents could act, observe, and correct their own work. These environments allowed agents to produce more merged AI-authored code before requiring additional input from a developer.



RECOMMENDATIONS

REC 2.1

Make setup one command. The agent (like a new hire) should be able to install dependencies, build, and run tests with a single documented command. Every undocumented environment quirk becomes a human handoff.

REC 2.2

Guarantee a fast, runnable test suite. Self-checking is the engine of autonomy: an agent that can run tests can catch and fix its own mistakes; an agent that cannot must ask you. Prioritize fixing flaky tests and broken harnesses — in our traces, a flaky suite was worse than none, because the agent couldn't distinguish its errors from the environment's.

REC 2.3

Expose fast feedback loops beyond tests. Type checkers, linters, formatters, and build steps that run quickly and produce machine-readable errors let the agent iterate cheaply. Slow or noisy feedback pushes verification onto the human.

REC 2.4

Write agent-facing repo documentation. Maintain a concise contributor guide at the repo root (e.g., an [AGENTS.md](#)/[CLAUDE.md](#)-style file) covering how to build, how to test, architectural conventions, and known sharp edges. This is the cheapest environment-readiness investment we know of.

REC 2.5

Reduce hidden state and permissions friction. Seed data, fixtures, service stubs, and credentials for local runs should be scripted or mocked. Tasks that require a human to "just run this one thing manually" guarantee at least one handoff per task.

REC 2.6

Keep the codebase legible. Consistent structure, meaningful naming, and colocated tests aren't just human niceties — they're the retrieval surface the agent navigates. Refactoring toward legibility pays out on every subsequent agent task.



Improving quality stewardship

Quality stewardship is about treating verification, production readiness, and future maintainability as part of the original task rather than the reviewer's problem.

The strongest trajectories did not stop when the code worked. They pushed the change toward local standards, surfaced risks and tradeoffs, verified the result, and left behind enough context for others to understand and maintain it.

REC 3.1

Demand explicit failure handling in the code itself. Have the agent handle error returns, null/empty inputs, and boundary conditions in the implementation, not just in tests.

REC 3.2

Review the diff before opening the PR. A read of the agent's diff — checking scope creep, deleted tests, and suspicious rewrites — catches the majority of what otherwise becomes review churn. Don't outsource the first review to your colleagues.

REC 3.3

Steer toward reusable conventions in the repo. Point the agent at adjacent files and ask it to match their shape — local naming patterns, file layout, which helpers to use. Rules files help set a baseline, but the unstated local conventions are the ones agents miss and reviewers catch.

REC 3.4

Surface blast radius before merge, not after incident. For production-touching changes, ask up front: what does this reach, at what scale, and how do we undo it? Push for rails — feature gating, reversibility, idempotency, correct migration ordering — as part of the change.

REC 3.5

Treat production readiness as part of the change. Honor external constraints (auth, PII and secrets handling, compliance regimes, accessibility) and ship observability — logs, metrics, alerts — with the new code path, not after the first bug surfaces.

RECOMMENDATIONS

REC 3.6

Make the agent prove its work. End tasks with an explicit verification step: run the suite, exercise the changed path, demonstrate the failing case now passes. "Write tests and run them" should be part of the prompt template, not an afterthought.

REC 3.7

Ask the agent to self-critique. A cheap, high-yield step: before finishing, have the agent list edge cases it may have missed, assumptions it made, and anything it couldn't verify. This converts unknown unknowns into an explicit checklist.

05 .

Conclusion



CONCLUSION

None of this is an argument that models do not matter. Better models raise the ceiling on what an agent trajectory can achieve.

But model selection is a decision a team makes a handful of times a year. Prompt clarity, environment readiness, and quality stewardship shape every task. The industry conversation remains heavily weighted toward the former; our data suggests the compounding returns may live in the latter.

These factors are trainable, toolable, and measurable. Because they can be scored from the trajectory itself, they also create the possibility of acting before downstream costs land: identifying an underspecified task before tokens are spent, an environment issue before it forces another human turn, or an understewarded change before the PR is opened.

That is the practical opportunity ahead: not just more capable agents, but engineering systems and operating practices that help teams get more from them.



06 . Appendix



How Span's agent evals were developed

We developed the trajectory evals in four stages:

Qualitative trace review. A panel of experienced engineers manually reviewed a stratified sample of internally developed trajectories drawn from both the best and worst tails of each downstream metric. They annotated, in free text, what distinguished the smooth trajectories from the expensive ones. This surfaced a longlist of candidate trajectory properties.

Rubric construction. We clustered the annotations into candidate dimensions and wrote explicit scoring rubrics for each — concrete, observable criteria (e.g., "the prompt states acceptance criteria," "the agent successfully executed the test suite at least once") rather than holistic impressions. Dimensions that couldn't be operationalized into observable criteria were dropped.

Automated scoring and calibration. We implemented each rubric as an automated eval that scores a full trajectory. To validate the automated scores, we compared them against blind human ratings on a held-out annotation set, iterating on rubric wording until inter-rater agreement between the eval and human annotators reached [agreement statistic, e.g., Cohen's κ / Spearman ρ threshold].

Correlation analysis. We then scored the full corpus and measured the relationship between each eval dimension and each downstream metric, controlling for confounds such as task complexity [and, e.g., team, repository, model version — list actual controls]. Dimensions that showed weak or unstable correlations were cut; the three categories reported above are the ones that survived as robust leading indicators.

We emphasize that these are correlational findings from observational data. We did not randomize prompt quality or environment state across matched tasks, so we can't fully rule out confounding (for instance, teams that write clear prompts may also differ in other disciplined ways). The consistency of these relationships across teams and task types supports their practical use as leading indicators, even where causal effects have not yet been established.