What Changes When Coding Agents Are Infinitely Parallel

Tom Rochette <tom.rochette@coreteks.org>

February 19, 2026 — cc8289f8

1 Prompt

Imagine you can run hundreds/thousands of coding agents in parallel.
How would you go about using them?

2 Answer

This is one of those questions where the interesting answer isn't "do everything faster" but rather **what fundamentally changes when parallelism is cheap**.

The core shift: from sequential exploration to parallel search

When you have one agent, you think carefully before acting — it's expensive to go down dead ends. With thousands, the calculus flips: *exploration becomes cheap, convergence becomes the hard problem.* Your job shifts from "what should I try?" to "how do I synthesize thousands of results?"

High-leverage use patterns

Speculative execution on decisions — At every architectural fork ("should this be a queue-based system or polling?"), you don't decide, you branch. Two fleets of agents build both, and you evaluate the result. This is huge for situations where you genuinely don't know which approach is better until you've tried it.

Mutation testing at scale — Spin up hundreds of agents making targeted changes to a codebase, each trying a different hypothesis about why a bug exists or how to optimize something. Like fuzzing, but semantically directed.

Full-stack consistency checking — Have agents simultaneously hold the contract between every pair of services in your system, constantly verifying that implementations match specs, that error handling is symmetric, that naming is consistent. Things that fall through the cracks in sequential review.

Competitive benchmarking of approaches — For something like your image pull latency problem, you could have 50 agents implement 50 different caching strategies against the same test harness, and just… pick the winner. No reading papers and reasoning about tradeoffs — empirically settle it.

Living documentation — Agents continuously reconcile docs, comments, and code. Every PR triggers agents that check for documentation drift, update runbooks, and surface inconsistencies.

The harder problems this creates

Synthesis is the bottleneck. If 1000 agents each produce a PR, you're back to being the serial bottleneck reviewing them. You need meta-agents whose job is to evaluate and rank the output of other agents, with clear scoring functions.

State and conflict. Agents working in parallel on the same codebase will conflict. You probably want them working in isolated sandboxes (branches, ephemeral clusters, test environments) with a merge/tournament layer on top.

Task decomposition quality matters more, not less. Bad task specs mean 1000 agents going confidently in the wrong direction simultaneously. The skill of writing tight, evaluable task specs becomes enormously valuable.

Evaluation functions become load-bearing. "Did the agent succeed?" needs a concrete, automated answer — you can't eyeball 1000 outputs. This pushes you toward test-driven development in a serious way.

What this implies for ML infra specifically

For your work — Kubernetes optimization, quota systems, image distribution — the interesting application is probably **parallel experimentation on real infrastructure parameters**. Spin up ephemeral clusters, have agents try different Dragonfly configurations or pod scheduling strategies, measure actual latency, tear them down. The thing that makes infrastructure hard is that you normally can't afford to run 50 experiments simultaneously. With cheap parallel agents managing the scaffolding, that constraint disappears.

The practical limit ends up being **compute and money, not ideas** — which is genuinely a different world than the one most engineering workflows were designed for.