ANANSI LABS

INTELLIGENCE IN EVERY THREAD

A CONCEPTUAL WORKING PAPER

The Agentic *Conway* Effect

A THEORY OF SOURCE-CODE DISPERSION UNDER MASSIVELY PARALLEL AI DEVELOPMENT

What happens to source code when a hundred machines write it at once, and whether the boundaries they carve will earn their keep.

ARMOND E. SINCLAIR, PhD

Theory · Propositions · A 1,040-run program

Intelligence in every thread

IN THIS ISSUE

KEYWORDS: coding agents · multi-agent software engineering · Conway's law · source-code topology · agent orchestration

ABSTRACT

The Agentic Conway Effect is a simple bet with conditions attached: when many AI agents truly write code at the same time, and splitting the code saves more coordination than the new boundaries cost, the same functionality will spread across more, smaller, clearly connected files.

The idea extends Conway's law from human org charts to machine workforces: how many agents write at once should shape how the code is organized. The evidence so far supports the ingredients, but not the simple claim that more agents always mean more files. Tightly coupled code, a central agent that merges everything, starved compute budgets, or rewards for consolidation can weaken or reverse the effect.

This paper turns that bet into a formal trade-off between coordination cost and interface cost, derives testable predictions, and lays out a staged experiment from one to 100 agents. It also predicts a lasting after-effect: boundaries created under heavy parallelism may stick around after the crowd is gone.

The central question is when software reorganizes around its machine writers, *and whether those new boundaries earn their cost.*

THE THESIS, IN ONE LINE

How many contributors write at once is an architectural force. Code is no longer just something machines run; it is the surface on which machine attention gets divided.

01

FEATURE

The Question

Most research asks how agents should divide tasks around a fixed repository.
This paper asks what the repository does back.

The rapid improvement of coding agents has made a once-theoretical question practical: what happens when dozens or hundreds of them are assigned to the same product? Research so far mostly treats the codebase as fixed scenery. The workforce scales around it, dividing tasks, merging changes, checking outputs.

But architecture has never been neutral to the organization that produces it. Human teams split systems into modules, services, and ownership zones partly to keep coordination manageable. Conway argued in 1968 that systems end up mirroring the communication structure of whoever builds them. If machines become a real share of the workforce, their constraints should push on architecture the same way.

And machine constraints are different. Agents read a limited window of code at a time. They get narrow assignments. They are safest working in isolated branches. Their work is tracked through logs and accepted by automated tests.

One dense file that mixes validation, execution, metrics, storage, and display may be perfectly readable to a senior developer. For twenty agents editing at once, it is a traffic jam. Split it into five modules and each one can be assigned, understood, tested, and traced on its own. The behavior is unchanged. The source has become more *dispersed*.

Whether that dispersion is adaptation or damage is the open question. Code tuned for machine parallelism may not look like code tuned for human understanding: better isolation and throughput on one side; too many interfaces, pass-through modules, and scattered logic on the other.

> *"When agents genuinely write in parallel, and the coordination they avoid is worth more than the boundaries they add, software will reorganize into more, smaller, clearly connected units that agents can work on alone, even when the product itself does not grow."*
>
> THE CENTRAL THESIS

Foundations

CONWAY'S LAW. Interfaces appear where groups have to talk; architecture is partly a record of who needed to coordinate with whom. Decades of evidence back this up: distributed organizations divide work along architectural lines (Herbsleb & Grinter, 1999); product structure and org structure match (MacCormack et al., 2012); org metrics predict which components fail (Nagappan et al., 2008). But the mirroring is not universal. Colfer and Baldwin (2016) found it much weaker in open-source projects. That caveat matters here: the effect should be strongest when working separately is the cheapest way forward, and weakest when contributors can safely share dense code.

INFORMATION HIDING. Parnas (1972) argued a module should hide a design decision that is likely to change, not just slice up execution steps. Stable public contracts then let the hidden parts evolve on their own. For massive parallelism, those contracts are what let agents avoid talking. So the theory predicts pressure toward *more boundaries, not necessarily better ones*. A split can carve out a real design decision, or it can exist purely so agents stay out of each other's way. Sometimes those are the same split. Sometimes not.

SOCIO-TECHNICAL CONGRUENCE. Work goes faster when the coordination that happens matches the coordination the code requires (Cataldo et al., 2006, 2008). Fifty agents in one dense component need to coordinate with almost everyone. One fix is better communication. The other is needing less of it: split the component. Restructuring the code becomes a coordination tool.

THE MICROSERVICES PRECEDENT. Independently owned services cut release coordination, until teams overdo it and get nanoservices and distributed monoliths: more boundaries, no real independence (Soldani et al., 2018). Splitting helps parallel work only until the cost of the seams takes over.

WHY AGENTS ARE DIFFERENT

Human communication is expensive but flexible. Agent communication is cheap to send and easy to misread: many messages, mismatched assumptions. When talk cannot be trusted, working independently is worth more.

OWNERSHIP CUTS BOTH WAYS

In Windows Vista and 7, components touched by many minor contributors failed more; components with a strong owner failed less (Bird et al., 2011). Narrow ownership zones cut simultaneous edits, but crowds inside a zone add risk.

The Evidence So Far

The ingredients are documented. The causal claim is not, and the counterevidence is instructive.

Bounded ownership scales. Self-Organized Agents assigns agents to components so output can grow while each agent's share stays small (Ishibashi & Nishimura, 2024). That is half the mechanism. The missing test: keep the product the same, add more agents, and watch whether the units multiply and shrink. No published experiment does this.

Coordination hurts at n = 2. In CooperBench, pairs of capable agents succeeded about 30 percent less often than agents working alone, mostly through broken promises and wrong assumptions (Khatua et al., 2026). Across 716 replayed pairs of overlapping pull requests, merge conflicts hit 41.7 percent between different agent models versus 19.8 percent within the same model (Xu et al., 2026). The friction concentrates exactly where agents share files.

A suggestive process signal. Across 132 multi-agent runs, Agile-style processes produced almost three times as many files as Waterfall with only about 22 percent more code (Ha et al., 2025). That points the same direction as dispersion, but the study varied process, not headcount, and its data cannot yet separate real code from process paperwork.

The counterexample. TheBotCompany's self-organizing system produced *flatter* code than average human submissions: 4 files where humans used 18; 7 versus 11; 6 versus 20 (Lyu et al., 2026). Multi-agent systems clearly can consolidate code instead. So the theory has to say when it applies, or it is simply wrong.

TABLE 1 · MEDIAN ARTIFACT SCALE BY MULTI-AGENT PROCESS (HA ET AL., 2025)

Process	Median files	Median LOC	LOC per file (ratio of medians)
Agile	37	578	15.6
Waterfall	13	475	36.5
V-model	19	934	49.2

A motivating signal, not confirmation: file counts include process artifacts, and LOC-per-file values are ratios of group medians.

The Effect, Defined

Take a repository that delivers a fixed, verified scope R in F production files and L lines of code. The number of agents you assign, A, is the dial the experiment turns; the number actually writing useful, mergeable code at the same moment, C, is what the system really receives. File-level density is simply $D = L / F$: a description of shape, not a quality score.

The deeper object is the **work surface**: a piece of the system that owns one job, exposes a clear contract, and can be tested on its own. Work surfaces and files are not the same thing. Several surfaces can live in one file, and splitting a file in half creates no new surface at all, so the analysis tracks both and the mapping between them.

A unit is **agent-addressable** when an agent can change it correctly without reading the whole repository. The experiment measures this directly: probe tasks with hidden answer keys, scored under a fixed evaluator and a fixed context budget. Documentation length and self-written tests do not count as proof.

Source dispersion is simply how many real units deliver a fixed amount of accepted functionality. It is not the same as good modularity. A system can be highly dispersed and badly organized at the same time.

NOT BLOAT

Three 200-line files versus fifteen 40-line files: same 600 lines, same verified behavior. The second is more dispersed, not larger. What grows is imports, declarations, contracts: the cost of getting around.

NOT AUTOMATICALLY GOOD

Good splitting keeps related logic together, contains failures, and lets parts be replaced independently. Bad splitting produces wrapper files, repeated conversions, and logic smeared across modules: overhead dressed up as modularity.

> *The worst outcome is procedural confetti: every piece simple, held together by glue, with no overall shape.*

The Model

The model prices a trade: every boundary you add costs interfaces, and every boundary you skip costs coordination among C(C−1)/2 pairs of agents who can collide.

$$J(N \mid C, Q) = \varkappa(N^{\eta} - 1) + \lambda\,[C(C-1)/2]\,N^{-\varrho} \qquad \text{EQ. 1}$$

$\varkappa, \eta$ · what each boundary costs, and how fast that cost grows: contracts, adapters, interface tests, navigation.

λ, ϱ · what each colliding pair of writers costs, and how much relief each new unit buys.

BUT DIVISIBILITY IS FINITE

$$1 \le N \le N_{max}(Q) \qquad \text{EQ. 2–3}$$

Some systems only divide so far. Only units that can be assigned, tested, and changed on their own count toward N. Splitting files without splitting responsibilities raises the file count and nothing else.

THE OPTIMUM RISES WITH CONCURRENCY

$$N_0(C) = [\lambda\varrho C(C-1)/2\varkappa\eta]^{1/(\eta+\varrho)} \qquad \text{EQ. 4}$$

In the simplest case, the ideal number of units grows roughly in step with concurrency. More simultaneous writers always push the ideal partition finer, until the system simply cannot divide further.

Throughput follows an inverted U. Adding real units speeds delivery up to the optimum, then slows it down as interface costs pile up. If the system hits its divisibility ceiling first, performance flattens instead of falling: the coordination pressure never gets relieved.

And boundaries dig in. Every boundary accumulates dependents, tests, deployment hooks, and ownership records. Removing one costs effort in proportion to all of that. So when the agent crowd shrinks, boundaries whose removal costs more than it saves simply stay: the architecture remembers its busiest era.

Eight Propositions

Eight ways the theory sticks its neck out, and what would break each one.

P1 · BOUNDED POPULATION EFFECT

Assign more agents and, as long as they actually write in parallel, units multiply and shrink, up to the divisibility ceiling. Only counts if quality, scope, and size hold steady.

P2 · MEDIATION BY DECOMPOSITION

More agents means more parallel work packets created before their homes exist; more packets become distinct units. Proving this requires watching the timeline, not just the end state.

P3 · ISOLATION AMPLIFIES

Isolated workspaces and branch-and-merge workflows strengthen the effect: they reward code that can be built and merged without talking to anyone.

P4 · TELEMETRY DECIDES

Tooling that tracks blame and enables rollback pushes toward more units. Tooling that lets agents see each other coming makes shared files safe and weakens the effect.

P5 · INTERFACE SUBSTITUTION

With the product held fixed, dispersion grows the connective tissue: contracts, adapters, interface tests, even if file counts barely move.

P6 · INVERTED-U THROUGHPUT

Speed rises as the code approaches its ideal partition and falls past it. No downhill side appears if the system runs out of ways to divide first.

P7 · CONSOLIDATION MODERATES

A periodic cleanup pass, by agents or humans, trims needless fragments without giving up parallel speed, as long as contracts and parallel capacity survive the cleanup.

P8 · ARCHITECTURAL HYSTERESIS

After the crowd leaves, code built at high concurrency keeps more units than code that never scaled. The boundaries that stick are the ones that accumulated structure around themselves.

The Experiment

The design is simple to state: give identical specifications to teams of **1, 5, 25, and 100 agents**, across new and existing codebases in Python, TypeScript, and Go. Same models, same neutral prompts, same acceptance tests. Only agents allowed to write production code count.

Randomizing team size answers the practical question: **what happens when you decide to assign more agents.** How much parallelism actually materializes is measured from telemetry, but it cannot prove cause on its own, because architecture limits parallelism while parallelism reshapes architecture. The experiment treats that loop as something to describe, not something to lean on.

Two compute setups rule out starving the big teams: one fixed total budget with a guaranteed floor per agent, one fixed budget per agent, both locked in before randomization. Every run stays in its assigned condition; when agents break things, that is a result, not a reason to toss the run. Each phase expands only after passing cost, schedule, and quality gates that never look at the code structure itself.

TABLE 4 · STAGED RUN BUDGET

Phase	Scale
Viability & variance pilot	72 runs
Study 1 · population × compute	288 runs
Study 2 · isolation × telemetry	192 runs
Study 3 · build × maintenance × consolidation	128 trajectories
Full pre-expansion program	**680**
Optional Study 1 expansion	+360 runs
Expanded program total	**1,040**

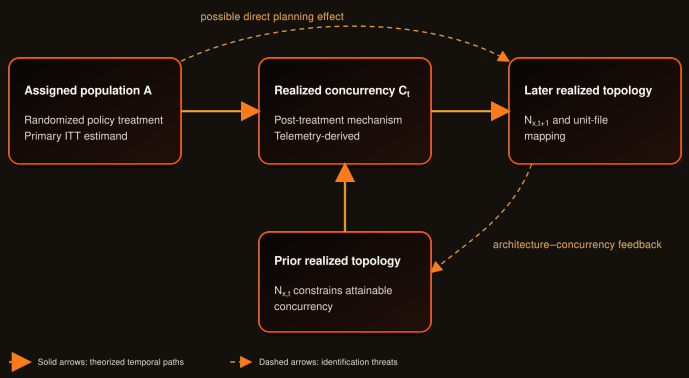

FIGURE · IDENTIFICATION STRUCTURE. *Team size is randomized, so its effect can be trusted. Realized parallelism and code structure shape each other over time, and planning can skip straight from headcount to architecture, which is why the middle of this diagram is measured rather than trusted.*

The Stakes

For engineering practice, agentic development adds a third thing to design for, next to changeability and human understanding: can a bounded worker pick up a unit, change it, test it, and hand it back? That means explicit rules, stable interfaces, local tests, and traceable dependencies. It is not a license for one-file-per-function design; responsibility matters more than size. And because splitting moves complexity into the seams, teams should watch the seams: wrapper files, interface sprawl, features smeared across modules. Boundaries that stop earning their cost should be retired on purpose.

For the theory, the rival explanations are the point: bigger projects, bigger budgets, language habits, model quirks, bossy orchestrators, framework boilerplate, and scope creep could each fake the signature. The experiment is built to shut each one out: fixed functionality, two compute setups, neutral prompts, requirement tracing, and careful separation of real code from scaffolding.

And the strongest disproof is named in advance: if adding agents genuinely raises parallel writing and the code's shape still does not move, the theory is dead. A theory that names its own kill shot is a theory worth testing.

"The goal is not maximum fragmentation or maximum agents. It is boundaries that earn their keep, without giving up system coherence or human understanding."

FROM THE CONCLUSION

Sinclair, A. E. (2026). *The Agentic Conway Effect: A theory of source-code dispersion under massively parallel AI development.* Conceptual working paper, Anansi Labs.

THEORY SYNTHESIS · RESEARCH PROPOSITIONS · CONTROLLED EMPIRICAL PROGRAM