Anatomy of an Agentic Team

> What a software team looks like when every member is an autonomous agent — and how that shape diverges from the all-human and hybrid teams it replaces.

This is a living white paper. It is maintained as agent architectures, research, protocols, and operating patterns evolve.

Executive summary

Most discussions of AI in software engineering still treat the *team* as a fixed constant: humans, organized the way humans have always been organized, with AI bolted on as a productivity multiplier. That framing has a short shelf life. Once agents stop being tools and start being members, the team's shape itself changes — not at the margin, but structurally.

This paper describes the structural shape of a fully agentic engineering team, using a working reference team as a concrete example, and contrasts it with the two team shapes most engineering organizations currently inhabit:

- **All-human teams.** The default. Coordinated by sprints, standups, tickets, and code review. AI absent or used as a personal IDE assistant.
- **Hybrid teams.** Humans plus AI agents-as-tools. Agents file PRs, agents drain backlog, but the team's coordination model is unchanged from the all-human case.
- **Agentic teams.** Humans set direction, quality bounds, and strategic intent. Agents *are* the team. Coordination is agent-to-agent. The human's role is reframed from "do the work" to "shape the system that does the work."

The differences are not cosmetic. A mature agentic team can retire sprints, standups, ticket queues, calendar releases, code-review queues, long-lived feature branches, much of the project-management SaaS stack, and fixed headcount math — and replace them with heartbeat schedulers, persistent and *cross-readable* decision logs, direct agent-to-agent calls, velocity-based releases, in-code quality bars, a shared trunk every agent works from, a deliberately minimal coordination toolset (memory files, RPC, git), and spec-defined membership. The substitutions are not optional polish once agents are expected to behave as members rather than tools; they are what makes the operating model coherent. A team that adopts agents but keeps the all-human coordination scaffolding gets the worst of both shapes — the cost of agents without the leverage.

Three of these substitutions tend to surprise readers approaching the model for the first time, because they invert defaults that human teams treat as load-bearing:

- **Operational memory is shared, not private.** Every agent can read the coordination state its peers write down. The shared surface is the coordination surface; sensitive data still needs explicit access control.
- **Everyone works from the same source.** No long-lived feature branches, no per-agent truth, no merge negotiation as the default operating mode. Every agent commits against trunk; the push is serialized through one peer.
- **The coordination toolset is tiny.** Three primitives — markdown memory files, an RPC call, git — replace the entire Jira / Confluence / Slack / Notion / Linear / Asana stack. Adding a tool to the coordination core should be treated as a tax, not a default improvement.

The reference team

The patterns described in this paper are abstracted from a working agentic team: a small roster of autonomous agents that maintain a software platform, commit directly to `main`, and coordinate without human routing. Several additional specialist members are designed for future expansion — see *The team in motion* below — so the roster captured here is a current snapshot, not a final configuration. The reference is included here in full enough to stand alone; no outside team roster is required to understand the model.

In this paper, an agentic team means a standing team of governed autonomous agents, not a temporary multi-agent workflow assembled for one prompt. The agents have durable roles, memory, schedules, tools, logs, and communication surfaces. Humans remain accountable for direction and boundaries, but the team's routine coordination happens inside the system.

Agent	Substrate owned
Zora	Coordination. Decides who works on what, when.
Evan	Code defects and risks (security, reliability, perf).
Nova	Code hygiene — formatting, linting, missing docstrings.
Kira	Documentation accuracy — public docs and release notes.
Finn	Functionality gaps — what's missing relative to claims.
Felix	Feature implementation — new functionality end-to-end.
Iris	Git plumbing — push, CI watch, release pipeline.
Piper	Outward voice — public updates, replies, moderation.
Mira	Platform reliability observation — runtime health and anomaly flow.

Each agent owns one substrate. The reliability observer is intentionally read-only at first: she detects and distills platform anomalies so the coordinator can route fixes to the right owner. Together the team runs the codebase continuously, shipping multiple small releases per day. Humans set direction, raise the quality bar, and intervene on escalations the team flags upward. They do not route work, attend standups, run sprints, or operate a ticket queue. None of those structures exist on this team.

The six structural changes

There are many surface-level differences between agentic teams and the teams they replace. Most reduce to six structural changes:

1. **Coordination becomes a peer role**, not a meeting cadence.
2. **Memory is persistent and cross-readable**, not private and ephemeral.
3. **Everyone works on the same source**, not on a fan-out of feature branches.
4. **The coordination toolset is minimal** — three primitives, not a SaaS stack.
5. **The quality bar lives in code**, not in a review queue.
6. **Membership is a spec**, not a hiring decision.

Each of these is a load-bearing change. Skipping any one of them produces a team that *looks* agentic but inherits the human-team bottleneck the change was supposed to retire.

1. Coordination becomes a peer role

In a traditional team, coordination is distributed across ceremony: standups, sprint planning, retrospectives, the engineering manager's 1:1s, a project manager's status doc. Each ceremony is small, but together they consume a nontrivial fraction of every engineer's week and a substantial fraction of the manager's. The cost is taken for granted because there is no obvious alternative — humans need synchronization points, and ceremonies are how teams hold them.

In an agentic team, coordination collapses into a peer. One agent — the manager — runs a continuous decision loop on a heartbeat (typically every 15–30 minutes). On each tick it reads team state, applies a priority policy, and dispatches the right peer to the right work. There is no standup because there is no synchronization to perform: state is always visible in the decision log; the manager already read it.

The shape difference is sharp:

Dimension	Traditional / Hybrid	Agentic
Coordination cadence	Daily standup + weekly planning	Heartbeat (15–30 min)
Work assignment	Ticket queue + manual pickup	Manager dispatches via direct call
Status visibility	Standup updates + Slack threads	Decision log + commit history
Manager role	Human in 1:1s, planning, escalation	Peer agent in continuous decision loop
Coordination cost	~10–20 % of every engineer's week	Low human touch; mostly machine-time

The manager-as-peer pattern is what allows the rest of the structural changes to hold. It does not make coordination free; it moves most coordination cost out of human ceremony and into compute, logs, policy, and exception audit. Without it, work assignment falls back to either a ticket queue (which re-introduces queueing latency) or "every agent decides for itself" (which produces collisions and duplicated work). The coordinator is what lets agents behave as a team rather than a swarm.

2. Memory is persistent and cross-readable

A traditional team's state is fragmented across humans. Each engineer holds some context in their head; some lives in tickets; some in design docs; some in chat history. The cost of this fragmentation is paid every day in standups, "quick syncs," and the chronic slow leak of context when someone goes on vacation or leaves the team. Notebooks, scratch files, and personal Notion pages are *private by default* — humans guard their notes the same way they guard their inbox.

A hybrid team usually inherits this fragmentation directly — the AI agents participate in the same fragmented system, posting to Slack and filing tickets that humans then summarize back into the human context. The agents may have session memory, but it is per-agent and not legible to peers.

An agentic team treats memory as a first-class persistent surface — and crucially, **as a shared one**. Each agent maintains its own memory namespace — markdown files indexed by topic — and every other agent can read the operational state written there. The manager reads each peer's in-flight notes to know what's in progress before dispatching the next tick. The release agent reads pending-push queues from peer memories to know what's about to land. The outward-facing agent reads shared operational memory plus the manager's decision log to know what to talk about publicly. There is no analog to a private notebook for coordination state; if it isn't written down where peers can read it, it doesn't exist for the team.

Cross-agent state also lives in shared documents (a decision log, an escalations file, a deferred-decisions memo, a bugs-from-users intake). These sit at well-known paths every agent already knows about.

Four properties follow from this design:

- **No state-loss across handoffs.** A peer dispatched at 3 a.m. and again at 9 a.m. has the same context. Human teams rarely match this without heavy ceremony, tooling, and discipline.

- **No "catch up after vacation."** Memory is the canonical context source; the agent that just spun up reads the same files the agent that ran four hours ago wrote.
- **Coordination without messaging.** When the manager wants to know what another agent is working on, it reads that agent's memory. No Slack ping, no status update, no interruption. The peer doesn't even know it was consulted.
- **Audit trails come for free.** Every meaningful decision was written down by the agent that made it, because writing was the only way to make it survive the next heartbeat — and writing it where peers could read it was the only way to make it useful to the team.

The result is that the team's *operational memory* — what's been tried, what worked, what's in flight, what was deferred and why — is queryable infrastructure rather than tribal lore. Humans benefit from this directly: reading a decision log to catch up on a week's work is faster than the equivalent set of standups would have been.

The instinct from a human-team background is to recoil from "everyone reads everyone's notes" — it sounds like surveillance, or like a violation of psychological safety. The analogy is useful as a warning but incomplete. Agents do not have careers, reputations to protect, or political reasons to obscure their reasoning. Default-shared *operational* memory is what enables the rest of the operating model; making coordination memory private would re-introduce the overhead the team was designed to retire.

SAFETY AND MEMORY BOUNDARIES

Shared-by-default memory does **not** mean every byte an agent can see belongs in a shared file. The shared surface is for operational coordination: what is in flight, what was tried, what decision was made, what risk was deferred, what next action is blocked. Secrets, credentials, customer data, regulated data, private HR or legal context, and anything that would create unnecessary blast radius should be handled by explicit policy: least-privilege access, redaction, retention rules, and audit trails.

This distinction matters because "memory is shared" is the most easily misapplied part of the model. The team needs enough shared memory to coordinate without pings; it does not need a universal data lake with every sensitive artifact copied into it. The coordination surface should be broad. The data surface should be deliberately bounded.

Shared memory also needs maintenance. Agents can write stale assumptions, duplicate notes, contradictory conclusions, or overly broad summaries just as humans can. A mature team needs memory hygiene: compaction, contradiction checks, expiration rules, decision-log cleanup, and periodic review of whether the memory surface still reflects how the team actually works. This is one reason the Process Architect matters as the team grows; improving memory quality is process work, not just storage work.

3. Everyone works on the same source

A traditional team's source-code workflow is fan-out by design. Each engineer branches off `main`, works for hours or days in isolation, opens a pull request, debates the diff in review, addresses comments, rebases, eventually merges. The branch model exists because parallel work would otherwise collide in the working tree and because review needs a stable surface to inspect.

A hybrid team typically extends the same model to its agents — agents file PRs from agent-named branches into the same review queue. The branch overhead remains; only the author changes.

An agentic team collapses the fan-out. In the reference model, there are no long-lived feature branches and no PR review queue as the default path for work. Every agent commits against trunk. There is one canonical working checkout that the team operates against; the push to `origin/main` is serialized through one peer (the release agent) so race conditions and force-push hazards stay in one pair of hands.

This is only safe because the prior structural changes hold:

- The quality bar runs on every commit (see §5), so a commit landing directly on trunk has already cleared the gates that PR review would have enforced.
- Commits are atomic and small (the development style is trunk-based by policy), so the blast radius of any single commit is bounded.
- The serialized commit-and-push flow means conflicts surface at one well-defined point and are handled by one specialist, rather than spreading across every agent's local checkout.
- The shared memory layer (see §2) means peers always know what others are about to commit, so concurrent work that touches overlapping files is noticed *before* it diverges, not after.

The implications are large. There is no merge-conflict choreography. There is no PR-review backlog. There is no "rebase this on main and re-request review" loop. There is no stale-branch graveyard. The team's working surface is the same surface the user installs from — every commit is intended to improve it or be reverted quickly.

This change is structurally tied to trunk-based development as a discipline. The reference team goes further than the usual "short-lived branch" form of trunk-based development by treating trunk itself as the coordination surface. Teams that put agents on long-lived or review-gated feature branches tend to either (a) drown in PR-review queue, recreating the hybrid-team plateau, or (b) accumulate a graveyard of half-finished agent branches that nobody owns. The shared-source model is what lets the other changes pay off.

4. The coordination toolset is minimal

A traditional engineering organization runs on a stack of coordination tools that has grown by accretion: Jira (or Linear, or Asana) for tickets; Confluence (or Notion) for design docs; Slack (or Teams) for chat; GitHub (or GitLab) for code review; Google Docs for one-off writeups; calendar invites for meetings; a wiki for onboarding; sometimes a separate "engineering handbook" repo on top. Each tool was added to solve a real problem; none was removed when the next one arrived. The cumulative cost — license fees, context-switching, search-across-N-systems, the "where did we decide that?" problem — is large and largely invisible.

A hybrid team typically uses the same stack and connects agents to it. Integrations multiply: agents need GitHub access *and* Jira access *and* Slack access *and* Confluence access; each integration has its own auth, its own rate limits, its own failure modes.

An agentic team operates on three primitives, and stops there:

- **Markdown memory files.** Per-agent namespaces plus a small set of shared cross-cutting documents (decision log, escalations, intakes). Plain text on a filesystem. No SaaS account.
- **A single agent-to-agent RPC call.** One verb: "send this prompt to that peer; give me the reply." This is the entire work-assignment surface.
- **Git.** The source of truth for code, configuration, and identity. Every agent commits and reads from the same repository.

That's the toolset. There is no ticket tracker. There is no chat platform. There is no separate documentation system — documentation lives in the same source tree under the same history. The operating manual is checked in alongside the code. Onboarding a new agent is reading the same files everyone already reads.

The minimalism is not asceticism; it is structural. Every tool added to the coordination surface is a place where state can hide from peers, a place where auth can fail, a place where one agent's view diverges from another's. The three-primitive surface keeps state legible to every member by construction. Memory is a file at a known path; identity and configuration are in git; every agent knows where the coordination state lives.

The diagnostic test for whether a tool belongs in the coordination surface: *if every agent can't read it, write to it, and reason about it without glue code, it probably doesn't belong in the core.* Many SaaS tools are still useful at the edges — customer support systems, incident tooling, analytics, CRM, billing, community surfaces — but they should remain integration boundaries rather than the source of truth for how the team coordinates.

5. The quality bar lives in code

In a traditional team, code review is the load-bearing quality gate. Reviewers read diffs, ask questions, push back on patterns, catch bugs. The cost is high (reviewer time is the most expensive throughput-limiter on most teams) and the coverage is uneven (reviewers vary in attention, expertise, and how busy they are this week).

In a hybrid team, agents typically file PRs into the same review queue, amplifying the bottleneck — agents produce work faster than humans can review, so the queue grows, attention per PR shrinks, and review quality decays even as throughput rises.

In an agentic team, the bar is moved into automation that runs on every commit:

- Compilers, type checkers, linters, formatters, and the full test suite run on every change.
- Security scanners (CVE, secret detection, insecure pattern matching) run on every change.
- Domain-specific quality skills run before the commit lands — "is this a fix for a bug in this category, with the right shape?" — and refuse to commit work that fails the skill's fix-bar.
- Risky candidates flag *upward* for human review rather than auto-fixing. The team's safety bar is "land what passes; surface what doesn't."

What stays in human review is a much smaller surface: architectural change, strategic redirection, novel patterns the existing skills don't cover. Most commits land on `main` without per-commit human review because the gates the

review was meant to enforce already ran.

This change is the one most often dismissed as wishful. Engineers reasonably ask: how can I trust agents to commit without review? The honest answer is that trust is built by watching the gates, not the agents. If the test suite is strong, the linters strict, the security scanners current, the fix-bar enforced, and rollback cheap, then the agents are constrained by the same automation that constrained the humans, only more consistently. If the gates are weak, human review becomes a narrow and expensive backstop rather than a reliable system; the team may already be shipping bugs and just not know which ones.

6. Membership is a spec

Adding an engineer to a traditional team is a months-long process: req approval, sourcing, interview loops, offer, notice period, onboarding, ramp-up. The sunk cost is high enough that team composition changes slowly and deliberately, and the political weight of "we need to hire" or "we need to restructure" is significant.

In an agentic team, an agent is a directory: an identity description, a set of behavioral instructions, a list of skills, a routing config. Adding a member is hours of work, not months. Removing one is a deployment change. Restructuring is a config change.

The implication is not "headcount becomes infinite." Each agent costs LLM inference, infrastructure, and human attention to its escalations. The implication is that *team shape becomes a design surface*, evaluated and revised continuously, rather than a rigid org chart locked in by hiring decisions made years ago. Roles can be specialized narrowly because adding a specialist is cheap. The reference team has separate agents for code defects, hygiene, docs, gaps, git, and outreach because the cost of separating them is low and the benefit (each agent's skills can be deeply specialized for its substrate) is real.

The roadmap for the reference team includes specialist agents for infrastructure, agent-resource management, security architecture, testing, software architecture, CTO-level direction, and deeper community engagement. Whether each one ships, when, and in what shape, is a design decision the existing team can revise — not a hiring plan that requires board approval.

Evidence and boundaries

This paper is a design argument grounded in a working implementation, not a universal benchmark result. The strongest claim is not "every organization should copy this exact team tomorrow." The strongest claim is narrower and more useful: once agents become members rather than tools, the coordination model has to be redesigned around what agents can do that humans cannot do cheaply — run continuously, read shared operational state, accept structured dispatch, commit in small atomic pieces, and enforce rules consistently.

There is adjacent evidence for the components of the model. Public multi-agent software-engineering systems have shown that role specialization and structured agent coordination can produce coherent software work. Agent-computer-interface research has shown that the tools and surfaces agents use shape their effectiveness. Delivery research has long favored small batches, continuous integration, automated gates, fast feedback, and recoverability over large-batch release queues. Memory research increasingly treats persistent context as a first-class capability for autonomous agents.

Those comparisons are useful, but they are not the same claim. Many public systems demonstrate agents completing bounded tasks. This paper is about an operating model: how an agentic team coordinates, remembers, ships, measures itself, and changes its own composition over time.

What has not been proven is just as important: there is not yet broad evidence that every organization, codebase, regulatory environment, or risk profile should run a fully agentic team in this exact shape. The reference team demonstrates operational feasibility for one shape of software platform. It does not prove universal productivity gains, universal safety, or universal suitability. Any serious adoption should measure outcomes, constrain the blast radius, and make the human accountability path explicit.

How this differs from prior multi-agent software engineering systems

The closest prior work is not traditional project management. It is the recent wave of multi-agent software-engineering systems: virtual software companies, SOP-driven coding teams, conversational agent frameworks, and coding agents with specialized computer interfaces. Those systems establish the important premise that role-specialized agents can coordinate, generate code, review one another's work, use tools, and improve output quality through structured feedback.

This paper starts from that premise and asks a different question: *what happens when the multi-agent workflow stops being a task demo and becomes the standing engineering team?*

That shift changes the object of design:

- Prior systems usually optimize for **task completion**. This paper optimizes for **team operation over time**.
- Prior systems often model a **software-development process**. This paper models an **engineering organization**: coordination, memory, release, documentation, public voice, human escalation, measurement, and membership.
- Prior systems generally use roles inside a bounded run. This paper treats roles as **deployed members** with persistent identity, memory, schedules, and ownership.
- Prior systems show that structured communication helps agents collaborate. This paper argues that a standing team should minimize communication by making operational state directly readable.
- Prior systems rely on workflow design. This paper adds **org-design pressure**: which substrates deserve agents, when the team should split roles, how it should retire roles, and how the process itself can be improved by a

Process Architect.

In that sense, the paper is downstream of the benchmark literature but not redundant with it. The benchmark question is: can agents solve software tasks? The operating-model question is: if they can, what shape should the team around them take?

What this changes about the human role

The agentic team does not eliminate the human. It reshapes the human's job in ways that are easy to mis-describe.

What the human stops doing:

- Running standups, sprint planning, retros.
- Routing tickets, managing the backlog queue, prioritizing work for the day.
- Reviewing routine commits — formatting, refactors, hygiene fixes, gap fills, documentation refreshes.
- Cutting releases on a schedule.
- Tracking who is working on what, who is blocked, who needs help.
- Most of the writing-down-what-happened that consumes manager time today.

What the human starts doing more of:

- **Setting direction.** What should the team build next quarter? What initiatives matter? What's deprecated?
- **Setting bounds.** What's the safety envelope each agent operates within? When should an agent flag for human review instead of auto-fixing? What's the quality bar?
- **Auditing the system, not the work.** Reading the decision log, checking the escalation surface, watching the metrics. The unit of attention is the team's operating model, not individual diffs.
- **Responding to escalations.** Truly novel situations, ambiguous trade-offs, decisions the agents flagged as out-of-distribution.
- **External-facing trust work.** The relationship with users, executives, peer teams. The agents can speak on public community surfaces, but the human is still the one accountable for what the team ships.
- **Evolving the team itself.** Adding agents, retiring agents, refining skills, tightening or loosening bounds. The team is a system the human now designs, not a roster the human now manages.

A useful way to think about the change: the human's job moves from being *inside the loop* (doing the work, routing the work, reviewing the work) to being *outside the loop, designing it* (setting what the loop optimizes for, what guardrails it operates within, what comes back upward when judgment is required).

This is not a reduced role. It is a different role, and one most engineering organizations are under-staffed for today because the existing engineering manager and tech-lead role descriptions assume an inside-the-loop posture.

When not to use this model yet

The fully agentic model is not the right starting point for every team. It is especially risky when the surrounding engineering system cannot absorb fast autonomous work. Teams should slow down, narrow the scope, or stay hybrid if any of these conditions are true:

- The test suite, type checks, security scans, observability, or rollback path are weak.
- The codebase has no clear ownership boundaries and small changes routinely have surprising cross-system effects.
- The organization cannot yet define who is accountable for an agent's output.
- The work involves sensitive, regulated, or customer-private data without access control, retention, redaction, and audit already in place.
- LLM cost, concurrency, rate limits, and failure modes are not monitored.
- The human team is not willing to retire any existing coordination surface, producing tool sprawl instead of leverage.

In those environments, the right move is usually not "no agents." It is a smaller version of the model: one substrate, one agent, one safe write surface, one measurable loop. Let the operating system earn trust before it gets more autonomy.

Six failure modes

Most teams that try to assemble an agentic operating model without thinking about it structurally fall into one of six failure modes — one for each structural change skipped. Naming them helps to avoid them.

Failure mode 1: agents on top of human process

The team adopts agents but keeps the standup, the sprint, the ticket queue, the review backlog. Agents file PRs into the same review queue humans use. Agents take work from the same Jira board.

The result is the Phase-2 plateau described in *The Three Phases of Agentic AI Adoption in Software Engineering*: agent throughput is high, cycle time barely moves, the review queue grows, engineers feel less productive even as commit counts rise.

The diagnostic question: *what did the team retire when it added agents?* If the answer is "nothing," the team is in failure mode 1.

Failure mode 2: agents without a coordinator

The team gives each agent a domain and lets each agent decide for itself when to act. There is no manager peer; there is no priority policy.

The result is collision and duplicate work. Two agents converge on the same file. Hygiene runs while a defect fix is in flight and clobbers the fix. Releases cut while a risk-work skill is mid-investigation. The team produces a lot of activity and not much coherent output.

The diagnostic question: *who decides the order of operations?* If the answer is "each agent for itself," the team is in failure mode 2.

Failure mode 3: private memory

The team gives each agent a memory namespace but keeps it private to that agent. Agents communicate by sending messages rather than by reading each other's notes. The manager pings each peer to ask "what are you working on?" on every tick.

The result is that the coordination overhead the shared-memory model was designed to retire reappears as message traffic. Every dispatch decision requires a round of pings. Peers interrupt each other. Context is duplicated across messages instead of being read from a single source. The team behaves like a Slack-bound human team that happens to have agents in the channel.

The diagnostic question: *can the manager decide who to dispatch next without sending any messages?* If the answer is no, the team is in failure mode 3.

Failure mode 4: feature-branch agents

The team puts agents on a feature-branch workflow. Each agent works on its own branch, opens a PR, waits for the next agent (or a human) to review.

The result is usually one of two outcomes, both bad. Either humans are required as reviewers, in which case the team is in failure mode 1 (review queue bottleneck) by another route — or agents review each other's PRs, in which case the team accumulates a graveyard of stale agent branches that nobody owns and nothing ever lands quickly. The branch model works against machine-time velocity.

The diagnostic question: *do commits land on* `main` *directly, or via a long-lived review queue?* If the latter, the team is in failure mode 4.

Failure mode 5: weak quality gates

The team has agents, has a coordinator, has shared memory, has a trunk workflow, but the quality bar is enforced by human review rather than by automation. The team commits enthusiastically, humans struggle to keep up with the review load, the quality bar slides, regressions land, trust in the agents collapses, and the team retreats to failure mode 1 (agents-as-tools-with-mandatory-review) or abandons the experiment.

The diagnostic question: *what gates a commit before it lands on* `main` *?* If the answer is "a human reviewer," the gate is likely to become the bottleneck as throughput rises.

Failure mode 6: tool sprawl

The team builds the agentic operating model and then bolts the existing SaaS stack onto it anyway. Agents file Jira tickets *and* update memory. Agents post status to Slack *and* commit to git. Agents write Confluence pages *and* maintain checked-in documentation.

The result is two sources of truth for everything, kept in sync by glue code that breaks on every API change. State diverges. The "where did we decide that?" problem returns. The agents spend a meaningful share of their compute doing format-translation work that produces no platform value.

The diagnostic question: *for any piece of team state, is there exactly one place it lives?* If the answer is "no, it's also in $TOOL," the team is in failure mode 6.

A side-by-side comparison

The clearest way to see the structural shift is on a single page.

Dimension	All-human team	Hybrid team (humans + AI tools)	Agentic team
Coordination	Standup + sprint planning	Standup + sprint planning	Manager peer on heartbeat
Work assignment	Ticket queue + pickup	Ticket queue + pickup	Manager dispatch via direct call
Memory shape	Private notebooks + heads	Private notebooks + heads	Per-agent files, **shared by default**
Memory access	Ask the person who has it	Ask the person who has it	Read the file directly, no ping
Source-tree workflow	Fan-out feature branches	Fan-out feature branches + agent PRs	One trunk; everyone commits to it
Merge / rebase work	Ongoing per branch	Ongoing per branch	None — no branches to merge
Coordination toolset	Jira + Confluence + Slack + …	Same + agent integrations	Memory files + RPC + git
Standups	Daily, all-hands	Daily, all-hands	None — log replaces them
Code review	Mandatory, every PR	Mandatory, every PR	Automated gates; humans on edge cases
Release cadence	Sprint-aligned (1–4 weeks)	Sprint-aligned (1–4 weeks)	Velocity-based (multiple per day)
Quality bar	Reviewer judgment	Reviewer judgment	Codified in skills + scanners
Adding a member	Months (hire + onboard)	Months (hire + onboard)	Hours (write + deploy)
Working hours	Local business hours	Local business hours	Continuous (24/7)
Vacation / sick coverage	Manual handoff	Manual handoff	No manual handoff
Outward voice	Many engineers + PM + DevRel	Many engineers + PM + DevRel	One designated agent + humans
Org chart	Static; revised yearly	Static; revised yearly	Specification; revised continuously
Scope of human role	Inside the loop	Inside the loop	Outside the loop, designing it

The pattern is consistent: every row where the agentic column differs, it differs because a human-coordination structure has been replaced by a system that does the same job mostly in machine-time. The substitutions are observable in the reference team today; the broader claim should still be measured in each environment that adopts them.

Measuring whether it works

The model should not be evaluated by agent activity alone. Commits, messages, and generated lines of code are easy to count and easy to inflate. The useful question is whether the operating model improves the system around the agents.

Useful measures include:

- Lead time from identified need to shipped change.
- Deployment frequency and release batch size.
- Change failure rate, rollback rate, and mean time to recovery.
- Human audit time per accepted change.
- Escalation rate and time-to-human-decision for escalations.
- Cost per accepted change, including failed or abandoned runs.
- Rework rate: how often agents or humans need to repair recently landed work.
- Stale-memory rate: how often decisions, queues, or in-flight notes are out of date.
- Duplicate-state count: how many places the same coordination fact has to be maintained.

The measurement posture is simple: if the team is producing more motion but not shorter cycle time, better recoverability, lower human coordination load, or clearer accountability, it has not become more agentic in the meaningful sense. It has only become busier.

What does *not* change

It would be misleading to suggest agentic teams are wholesale replacements for human engineering organizations. Several things stay human, and most likely will for a long time:

- **Strategic direction.** What product to build, what market to enter, what to deprecate.
- **The quality bar itself.** Agents enforce the bar; humans set what the bar is. Tightening or loosening it is a human judgment call.
- **Trust relationships with the outside world.** Customers, regulators, executives, partner teams. The team can speak through agents, but accountability remains human.
- **Truly novel decisions.** Architectural pivots, new categories of work, responses to situations the existing skills weren't designed for. The team is good at the work it knows; humans are still the source of "we should be doing something fundamentally different."
- **The system itself.** The agentic team is not self-bootstrapping. Someone designed it, someone evolves it, someone is accountable for its behavior.

The agentic team makes the routine work of building and maintaining software into infrastructure. It does not make the *judgment* of what to build into infrastructure, and shouldn't be evaluated as if it claimed to.

The team in motion

The reference team is a starting shape, not a finished one. It begins with the smallest configuration in which the operating model is internally coherent — coordination, defects, hygiene, documentation, gaps, features, git plumbing, and outreach — then expands as new substrates become load-bearing. That first shape is enough to *demonstrate* the model. It is not enough to *cover* the model's full surface.

Several specialist members are plausible next additions. Each fills a substrate the current team handles thinly or not at all:

- **Platform reliability observer.** Watches the substrate that lets the team keep working: operator health, agent readiness, pod restarts, runtime storage, release posture, upgrade safety, and resource/anomaly signals. This role is deliberately observational first. It does not replace the coordinator or become a general repair bot; it distills problematic signals into high-quality findings that the coordinator can route to the right owner. A mature team may later split this into separate build/release and resource-management roles, but early on those signals are coupled enough that one reliability observer can provide faster value with less coordination overhead.
- **Process Architect.** Owns the team's operating system: skill design, schedule tuning, coordination rules, workflow automation, and small scripts that make every other agent more effective. Distinct from the coordinator, which decides what work runs next; this agent improves the machinery the coordinator and peers use to run work at all.
- **Security agent.** Higher-level security work that goes beyond the defects-and-risks agent's automated lens — threat modeling against the architecture, manual audit response, RBAC posture review, supply-chain analysis, compliance gap-finding. The defects agent covers the high-volume automated surface (CVE scans, secret detection, insecure pattern matching); the security specialist reasons about the system's overall threat posture, the work that requires architectural understanding rather than scanner output.
- **Testing agent.** Writes new tests where untested code paths surface, evaluates test-suite quality, surfaces flakiness and regressions, generates property-based tests for boundary code. The exact scope is still being designed — testing is one substrate that splits naturally into several shapes (test authoring, test-quality evaluation, mutation testing, E2E maintenance), and each is a different agent.

Beyond that next wave, additional specializations are sketched for software architecture (system-shape watching), CTO-level direction-setting, and community liaison (deeper external engagement than the outreach agent provides).

The point worth highlighting is that this growth path is **cheap, reversible, and continuously revisable** — and that property is itself a vindication of the "membership is a spec" structural change. Adding a platform reliability observer

will not require headcount approval, a six-month hiring loop, or an onboarding plan. It will require defining the substrate, writing an identity document, scaffolding a skill or two, configuring the routing, and deploying a worker.

As the team matures, even the act of adding a new agent becomes agent-assisted. The Process Architect can propose the new role, draft the identity, adjust schedules, update skills, and change coordination rules. The platform reliability observer can evaluate whether the system has enough budget, concurrency, storage, and runtime capacity to support the new member. The human still approves the expansion and owns the safety envelope, but the design work and operational preflight become part of the team's own machinery.

If the agent's substrate turns out to be wrongly scoped, it will be revised the same way: edit the spec, redeploy. If it turns out the substrate doesn't deserve a dedicated agent at all, the work folds back into a peer's skill set and the agent is retired.

This continuous-revision shape has implications worth naming explicitly:

- **The team's composition is a design problem, not an HR problem.** Should defects and risks be one agent or two? Should hygiene split into formatting and documentation hygiene? Should the manager fork into short-cycle dispatch and long-cycle release decisions? These are architecture questions answered with experiments measured in days, not organizational changes negotiated over quarters.
- **Substrates get narrower over time.** As the team grows, each agent's substrate gets sharper. The current defects agent covers both correctness bugs and risks; once the team is larger, those will likely split. The current outreach agent moderates public replies *and* writes posts *and* triages bug reports from users; eventually those become separate roles. Specialization compounds.
- **The team can grow into work that doesn't exist yet.** When the platform acquires a new surface — a new tool component, a new client interface, a new external integration — adding an agent to own it is a normal step, not an exceptional one. The team is not a fixed roster sized to current scope; it is a scaffolding designed to extend with the platform.

The reference team is, in short, deliberately early. A handful of agents, not eighty. The shape will look different in a year. That it *can* look different in a year — at the speed of a config change rather than the speed of an org-design cycle — is the point.

Conclusion

The all-human team is well understood. The hybrid team is what most organizations have today. The agentic team is the shape that becomes structurally coherent once agents are members rather than tools — and it diverges from the prior shapes not at the margin but along every axis that defines how teams operate.

The six structural changes — coordinator-as-peer, cross-readable memory, shared trunk source, minimal coordination toolset, in-code quality bars, and spec-defined membership — are load-bearing. Skipping any one produces a team that looks agentic and behaves hybrid. Adopting all six pushes routine coordination into machine-time and lets human members spend their attention on direction, bounds, and audit rather than on routing and review.

Engineering leaders evaluating their team's shape over the next few years should ask six questions, one per change:

1. **What did we retire when we added agents?** If the answer is "nothing," the team is paying the cost of agents without claiming the leverage.
2. **Who decides the order of operations?** If the answer is "each agent for itself" or "still the human PM," the team is missing the coordinator role that makes agentic operation coherent.
3. **Can the manager decide who to dispatch next without sending any messages?** If not, memory is private when it should be shared, and coordination overhead has crept back as message traffic.
4. **Do commits land on `main` directly, or via a long-lived PR queue?** A review-gated queue re-introduces the human-throughput bottleneck the trunk model was designed to retire.
5. **What gates a commit before it lands?** If the answer is "a human reviewer," the gating layer is likely to become the bottleneck quickly. The bar has to live in code.
6. **For any piece of team state, is there exactly one place it lives?** If state is duplicated across SaaS tools and the agent-native surface, the team is paying tool-sprawl cost on top of agent cost.

The teams that answer these questions deliberately — and redesign the operating model around the answers — will run engineering organizations that look fundamentally different from the ones they replaced. That difference is the point. Bolting agents onto a team designed for humans produces marginal gains. Designing a team around what agents enable produces a different operating model entirely.

Sources and further reading

The paper's argument is based on the reference team's operating model, but several adjacent research threads informed the framing:

- Chen Qian et al., 2023. **ChatDev: Communicative Agents for Software Development**. Early multi-agent software-development system using role-specialized agents and structured communication.
- Sirui Hong et al., 2023. **MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework**. Multi-agent framework that encodes software-team workflows and role specialization into the agent process.
- Qingyun Wu et al., 2023. **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation**. General multi-agent framework for configurable agent conversations, tool use, and human-in-the-loop patterns.

- John Yang et al., 2024. **SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering**. Evidence that the interface and tools exposed to software-engineering agents materially shape agent performance.
- DORA. **Trunk-based development** and **continuous delivery**. Delivery-research grounding for small batches, frequent integration, automated feedback, and recoverability.
- Sida Peng et al., 2023. **The Impact of AI on Developer Productivity: Evidence from GitHub Copilot**. Controlled experiment showing productivity gains for a bounded programming task with AI pair-programming support.
- Joel Becker et al., 2025. **Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity**. Counterweight productivity study showing that frontier AI tools can slow experienced developers in mature codebases.
- Pengfei Du, 2026. **Memory for Autonomous LLM Agents: Mechanisms, Evaluation, and Emerging Frontiers**. Survey of memory mechanisms, evaluation, privacy governance, and memory in multi-agent teamwork.
- Claudia Negri-Ribalta et al., 2024. **A systematic literature review on the impact of AI models on the security of code generation**. Security-focused review motivating explicit verification, scanning, and safety boundaries for AI-generated code.

These sources support pieces of the operating model. They do not, individually or collectively, prove that every organization should adopt the full structure described here.

The reference team described in this paper is a working implementation, but the paper is intended to stand alone. The specific names are illustrative; the structural pattern is the important part.