Agent Teams

Part IV of *The End of Software Engineering and the Rise of Agentic Engineering*

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com

September 2026

Abstract

Practitioners who outgrow one agent face three questions: when several agents are the right answer, how to arrange them, and what the arrangement costs. We define seven objects operationally: a team, a team contract between two agents, a shared board, an ownership rule, a serialized shared resource, an integration review, and a referee. Four orchestration patterns, fan-out with fan-in, the pipeline, the supervisor tree, and directed-acyclic-graph execution, are treated as compositions with stated guarantees and failure handling. Each pattern has a named ancestor in distributed systems published between 1973 and 2014. Two production code bases supply the orchestration evidence: an Elixir system with supervision trees and a staged pipeline, and a Rust control plane with a layered graph executor, worker leases, and a database-backed team roster. The Rust system admits two nodes for concurrent execution only when their declared resource sets are non-empty and disjoint, a rule a companion formal series also derives. The cost evidence for teams is stronger than the benefit evidence. One vendor engineering post reports a 90.2 percent improvement over a single-agent baseline on an internal evaluation and, in the same post, about fifteen times the tokens of a chat interaction. Against it, a fixed three-phase pipeline resolved 32.00 percent of SWE-bench Lite at seventy cents an instance, a taxonomy documents fourteen failure modes across seven frameworks, and a first-party decomposition experiment resolved zero of ten instances at eight-billion-parameter scale. The main limitation is that the positive evidence is one vendor-internal figure, not yet independently replicated.

1 Introduction

A practitioner who has automated one task with one agent soon meets a task that resists it. The task overflows one context window, or splits into parts that could run at the same time, or needs a check the agent doing the work should not perform on itself. The obvious remedy is a second agent. Much less obvious is when that remedy is right, how the agents should be arranged, what they must promise each other, and what the arrangement costs.

Research question. RQ4: how do multiple agents coordinate, communicate, and check each other, and which orchestration patterns hold up?

This paper is Part IV of a five-part series. Part I establishes the central claim that software engineering is being redefined as agentic engineering. The unit of work is moving from hand-authored code to specifications, contracts, orchestrated agent activity, and verification of machine-produced

artifacts. Agent teams make that change concrete: engineering judgment defines the contracts, ownership, evidence, and acceptance authority under which several agents can act as one system.

A team sits one level above the loop of Part III: several such loops have to fit together. The failure modes Part III catalogues, among them self-review, stub artifacts, skipped stages, unverified success claims, and silent fallbacks, still occur inside every member of a team. Only the modes that need two agents before they can happen are new here.

Part I defines agent and autonomy level; Part II the contract, the orchestrator, the sandbox, verification, and the composition operators; Part III the loop and the gate; Part V memory. Those definitions are cited by number below and not restated.

Contributions. Seven definitions name the objects a practitioner needs to reason about a team. Each is decidable by someone with access to a running system, and each is strict enough to exclude things commonly called by the same word. A crowd of agents working in parallel on unrelated tasks is not a team. A rule that lives in a prompt and is honoured by good behaviour is not an ownership rule. A component that records two disagreeing verdicts without deciding between them is not a referee.

The four patterns are treated as compositions, each with its guarantee and its behaviour on failure, together with the machinery that holds a composition together: the team contract as a typed boundary between two agents, the shared board, the ownership rule, serialized resources, integration review, and the referee. The orchestration evidence comes from two production code bases read at pinned commits, supplemented by vendor documentation and the multi-agent research literature.

On cost, and on the cases where a team is the wrong answer, the evidence is weakest. The costs of running a team are measured and public. The benefits are largely vendor-internal, evaluated on non-public tasks, and unreplicated.

Evidence. Two kinds of source carry this Part. Published literature and vendor documentation are cited with the class of source named at the point of citation. A randomized trial, a preprint, a vendor engineering post, and a product documentation page do not carry equal weight.

Repository evidence is cited by pinned commit and file path. Every such statement is an implementation claim established by reading source at that commit: the implementation at commit X performs Y. It is not a claim about production behaviour. No production traces were available.

Composition as the organising idea. Part II argues that agent systems are built by composing parts with explicit inputs and outputs across typed boundaries, keeping effects explicit and recorded, and treating artifacts as immutable values that flow between steps. A team is the largest composition in the series, so this is where that argument pays off or fails to.

Two of its claims can be tested here. Typed boundaries are said to catch defects that would otherwise propagate, and Section 10 finds that five of fourteen documented multi-agent failure modes are specification or interface defects. Composition is said to be safe when component footprints are disjoint, and that criterion turns up twice independently: as a numbered result in a companion formal working series, and as a production concurrency-admission check written without reference to it. Section 7 examines both.

2 Teams

Each definition below is written so that an observer with access to a running system can decide whether it holds. Objects defined in earlier Parts are used without restatement: the agent and the autonomy level from Part I (Definitions 4.2 and 4.3); the contract, the orchestrator, verification, the composition operators, the typed boundary, and the immutable artifact from Part II (Definitions 3.4, 3.7, 3.8, 3.15, 3.11 and 3.14); and the gate, the loop, and bounded iteration from Part III (Definitions 2.1, 2.2, and 2.4).

Definition 2.1 (Team)**.** A *team* is a set of two or more agents working on parts of one task, such that at least one agent's input depends on another agent's output, and such that the assignment of parts to agents is recorded in a place both agents can read. Agents running in parallel on unrelated tasks are not a team, however many of them there are, because no agent's input depends on another's output. A single agent that delegates to helpers which only return summaries to it is a team only if the helpers can read each other's assignments; otherwise it is one agent using other agents as tools.

The exclusion in the last sentence follows the distinction drawn in primary vendor documentation. A subagent runs in its own context window with a custom system prompt and restricted tool access, does its work, and returns only a summary to the agent that called it [12]. Teammates, in the same vendor's usage, share a task list, claim work, and communicate directly with each other [13]. That documentation recommends checking whether the lighter arrangement will do before setting up the heavier one, which is the definition put to practical use. If no member needs to learn what another member decided, a team buys nothing.

Definition 2.2 (Team contract)**.** A *team contract* is a contract, in the sense of Definition 3.4 of Part II, whose obligated party is another agent rather than a human. It carries everything a contract carries and adds two names: a *producer*, the agent that must supply the artifact, and a *consumer*, the agent that will receive it. Written as a tuple, a team contract is $c = (p, q, S, \chi)$ with producer p, consumer q, a shape S that the handed-over artifact must satisfy, and an acceptance check χ that the consumer runs before it begins its own work. A statement is a team contract only if χ can be decided without asking either p or q whether the handoff was good.

Definition 2.3 (Shared board)**.** A *shared board* is a single append-only record that every member of a team can read and write, used to publish state that other members depend on. Reading the board is how a member learns another member's state. A private message from one member to another is not a board, because a third member cannot learn from it; a log that only the orchestrator reads is not a board, because members cannot learn from it. Append-only means an entry, once written, is not edited in place: a correction is a new entry that supersedes an earlier one, and both remain readable.

Definition 2.4 (Ownership rule)**.** An *ownership rule* is a stated assignment of each shared object to exactly one team member permitted to change it, such that a change by a non-owner is detectable. Write it as a partial map ω from shared objects to members. The rule is in force only if the detection is mechanical: a rule that lives in a prompt and is enforced by the members' good behaviour is a convention, not an ownership rule.

Definition 2.5 (Serialized shared resource)**.** A *serialized shared resource* is a resource that more than one team member needs and that is guarded so that at most one member uses it at a time, with waiting members blocked rather than proceeding. The blocking requirement is what distinguishes serialization from a hint. A member that fails to acquire the guard and continues anyway has not been serialized, and a resource whose guard can be bypassed by a member that decides the guard does not apply to it is not serialized either.

Definition 2.6 (Integration review)**.** An *integration review* is a review performed after every team member has finished, whose object is the consistency of the members' outputs with each other rather than the correctness of any one output. It checks that shared objects are defined once, that every cross-reference resolves, and that the interfaces the members assumed of each other agree. An integration review is distinct from per-member verification in what it can fail on: it fails when two individually passing outputs disagree.

Definition 2.7 (Referee)**.** A *referee* is a party distinct from the disagreeing members that resolves a disagreement when the members' own verification procedures return conflicting verdicts, and whose decision the members treat as final for that disagreement. A party is a referee only if its decision has a defined effect on what happens next. A component that records the disagreement, or that reports both verdicts upward without deciding, is an observer, not a referee.

Definition 2.8 (Fan-out and fan-in)**.** *Fan-out and fan-in* is the parallel operator of Definition 3.15 of Part II applied at team scale: a set of members receives copies of a shared input, each produces its own artifact, and a named combining step consumes all of them. The pattern is instantiated only when four things are stated: the fan width, the bound on how many members may be in flight at once, each member's effect footprint, and the combining rule, including what the combining step does when some members fail. Completion-order independence requires the declared and actual footprints to be disjoint and the combining rule to be commutative or applied in a canonical order.

Definition 2.9 (Pipeline)**.** A *pipeline* is the sequence operator of Definition 3.15 of Part II applied at team scale: an ordered list of members in which each member's team contract names the previous member as producer and the next as consumer. The pattern is instantiated only when the failure rule for a stage is stated, because a pipeline whose stages fail silently degrades into a sequence of unrelated agents.

Definition 2.10 (Supervisor tree)**.** A *supervisor tree* is the supervise operator of Definition 3.15 of Part II applied recursively: a tree in which every non-leaf node is a supervisor holding a restart policy over its children, and every leaf is a member doing work. A supervisor holds no work of its own. The pattern is instantiated only when the restart policy names which siblings are affected by a child's failure and what the supervisor does when restarts exceed a stated intensity within a stated window.

Definition 2.11 (DAG execution)**.** *DAG execution* is the general form of which the pipeline and fan-out patterns are special cases: members are the nodes of a directed acyclic graph whose edges are team contracts, and execution proceeds by repeatedly running a set of nodes whose producers have all completed. The pattern is instantiated only when the graph is checked for cycles before any node runs and when the behaviour on a node failure is stated for the nodes downstream of it.

Notation used in the rest of the paper. A team is written $T = (A, C, B, \omega)$ with agent set $A = \{a_1, \ldots, a_n\}$, a set C of team contracts, a board B, and the ownership map ω. Following the companion formal working series, $\mathrm{supp}(x)$ denotes the set of names an object x touches, called its support or footprint. No operator symbols are introduced for the composition operators; they are named in words, as Part II names them.

3 Prior art

Every arrangement of agents analysed in this paper was designed, named, and documented for processes rather than agents, between one and five decades before large language models existed. Table 1 gives the lineage.

Agent-era pattern	Named ancestor	Year	Primary source
Isolated members exchanging messages	Actor model	1973	Hewitt, Bishop and Steiger at IJCAI [19]
Shared board	Blackboard architecture in Hearsay-II	1980	Erman, Hayes-Roth, Lesser and Reddy, *ACM Computing Surveys* [18]
Serialized shared resource, associative coordination	Tuple spaces in Linda	1985	Gelernter, *ACM TOPLAS* [23]
Supervisor tree with restart policy	OTP supervisor behaviour, let it crash	2003	Armstrong's dissertation [7] and the OTP reference [34]
DAG execution over declared dependencies	Apache Airflow	2014	The Airflow project page [4]
Checkpoint and replay after failure	Durable execution	current	Temporal documentation [43]
Branch, parallel, map, retry, catch, human callback	State-machine workflow vocabulary	current	AWS Step Functions documentation [40]

Table 1: Ancestry of the orchestration patterns. Each row names a pattern in current agent practice and the published work that introduced the same structure for processes or tasks. Years are the publication year of the primary source; the two rows marked current name live documentation for patterns whose introduction dates are not stated on the pages cited.

The lineage matters because the failure knowledge comes with it. Arrange agents in a supervisor tree and you inherit a worked answer to the question of what happens when a child keeps failing. The OTP reference specifies an intensity and a period, and states that when more than the permitted number of restarts occur inside the window, the supervisor terminates all its children and then itself [34]. The defaults are one restart within five seconds. The problem that answer solves is concrete: unbounded restarting turns a deterministic bug into an infinite loop that burns resources and fills logs.

The shared board carries the same inheritance. Hearsay-II is its direct ancestor, a shared repository in which specialist knowledge sources update a common structure rather than calling each other [18]. What transfers is architectural. The specialists need to know about the board, not about each other, so the number of interfaces grows with the number of specialists rather than with its square. A team whose members message each other directly lacks that property, and pays a different price for each new member.

The transfer is partial, because the classical patterns assume components that fail by stopping or by returning a wrong value, and that behave the same way on the same input. An agent can fail by producing a fluent, well-shaped, confidently wrong artifact, and can behave differently on identical input. Restart-and-retry answers a transient crash well and answers a systematically wrong agent not at all.

A gap remains: we found no source connecting two-phase commit, leader election, or the consistency and availability tradeoff to agent orchestration in the sources reviewed (cutoff 2026-09-01). Those are the parts of the distributed systems tradition that bear most directly on agents writing shared state.

3.1 The vendor pattern vocabulary

Alongside the ancestry there is a current vocabulary. One vendor engineering post, published in December 2024, names six arrangements: prompt chaining, routing, parallelization, orchestrator-

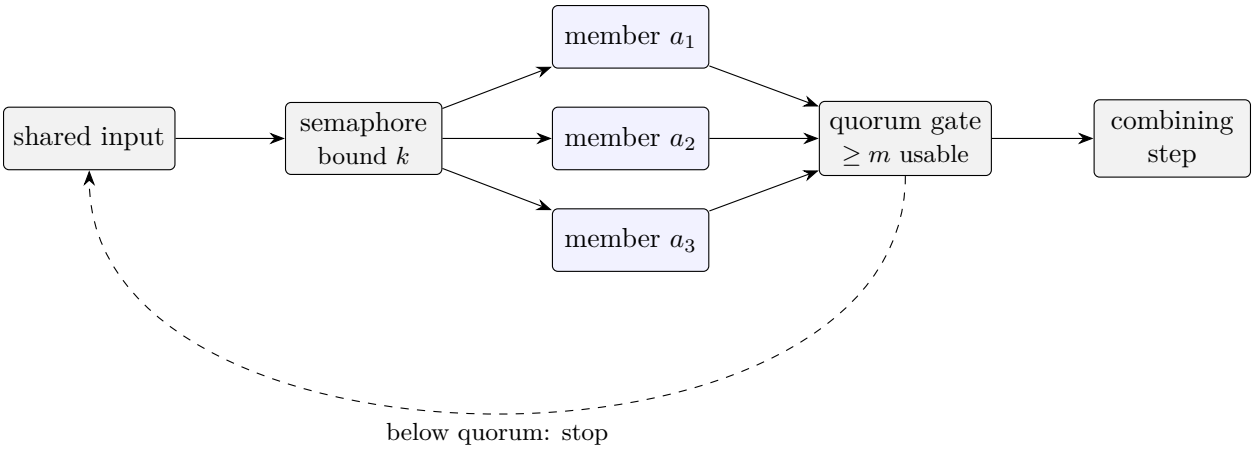

Figure 1: Fan-out and fan-in with an admission bound and a quorum gate. The semaphore bounds how many members run at once, independently of how many were fanned out. The gate turns partial success into a defined outcome by requiring a minimum number of usable results before the combining step runs.

workers, evaluator-optimizer, and autonomous agents [5]. The same post draws the distinction this series uses between workflows, in which models and tools are orchestrated through predefined code paths, and agents, which direct their own processes and tool usage. That distinction is Part I's, and Part I owns it.

Six published frameworks build from the same small set of arrangements. One executes a crew of agents sequentially, or under a manager agent that delegates and validates outcomes before proceeding [16]. A second exposes a state graph whose nodes are functions over shared state and whose edges route between them; composing nodes and edges creates looping workflows that evolve the state over time [21]. A companion library adds the hierarchical case, where a central supervisor controls communication flow and task delegation [22]. A third encodes standardized operating procedures into prompt sequences [27]. A fourth arranges agents along a chat chain across design, coding, and testing phases [11]. An early conversational framework built applications from multiple agents that converse with each other [8]. A vendor software development kit exposes handoffs, which let agents delegate to other agents, and guardrails, which validate agent inputs and outputs [31].

Read against Table 1, the six-pattern vocabulary restates older structures. Prompt chaining is a pipeline. Parallelization is fan-out and fan-in. Orchestrator-workers is a supervisor tree one level deep. Routing is a branch. Evaluator-optimizer is Part III's review-fix loop with a second agent as the reviewer. The vocabulary is useful because it is small. A practitioner learning it is learning five familiar structures under new names.

4 The four patterns as compositions

Each pattern below is given in the same form: what it takes, what it guarantees, what it does when a member fails, and the implementation evidence. The guarantees are stated as conditionals, because each holds only under a premise that is easy to leave unchecked.

4.1 Fan-out and fan-in

Requirements. A shared input, a set of members, a bound k on how many run at once, declared effect footprints for the members, and a combining rule.

Guarantees. The combined result does not depend on completion order only when member effects are actually disjoint and the combining rule is commutative or consumes results in a canonical order. Distinct output names are necessary but not sufficient: two members may still mutate the same file, database row, or remote resource. Let two members both write a key called `summary`: whichever finishes last decides what the combining step sees, and the run becomes order-dependent with no error raised. A companion formal working series proves the general form. When sibling nodes can write the same output key, the map from a validated plan to its final store does not factor through the composition structure at all, so no order-independent reading of the arrangement exists (Corollary 5.12, `cor:no-algebra`) [38].

Failure handling. Two mechanisms answer two different questions. An admission bound answers how many members may run at once. A quorum gate answers what counts as enough results.

One production control plane implements the bound as a `tokio` semaphore. Its permit count comes from the manifest's declared concurrency and defaults to the layer width, every node acquires a permit before its task is spawned, and results are collected by index so that they reassemble in declaration order rather than completion order [37, AgentHero at 1c3ad24011, `crates/dag-executor/src/lib.rs`, lines 2350 to 2410]. It implements the gate as a type carrying an optional minimum count of usable sources [37, AgentHero at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 181 to 187]. Practitioners should pair the two. Bounding concurrency without a quorum rule yields an arrangement that runs politely and then silently combines whatever happened to succeed.

The check that makes concurrency admissible. The same control plane goes further than a semaphore. Before dispatching a layer concurrently, it requires every pair of nodes in the layer to be compatible. Compatibility is decided by a function that refuses if either node is declared exclusive, admits if either is declared pure or both are read-only, and otherwise admits exactly when both nodes declare a non-empty resource set and the two sets do not overlap, an undeclared set being treated as conflicting with everything [37, AgentHero at 1c3ad24011, `crates/dag-executor/src/lib.rs`, lines 5333 to 5358 and 5485 to 5510]. That is a disjoint-footprint test, reached from operational necessity. Section 7 returns to it.

4.2 Pipeline

Requirements. An ordered list of members, a team contract between each adjacent pair, and a rule for what a failing stage does.

Guarantees. Each stage sees the previous stage's output, and no stage runs on an input the previous stage did not produce.

Failure handling. A stage will fail. The question is whether the failure is attributable. One Elixir implementation folds over the declared stages with an early-exit reduction, halting on the first failure and returning a value that carries the failing stage's name alongside the underlying reason [36, agent-os at 61cb3994da, `src/agent_os/lib/agent_os/pipeline.ex`, lines 55 to 68]. Only a run in which every stage succeeds reaches the contract check over the collected artifacts.

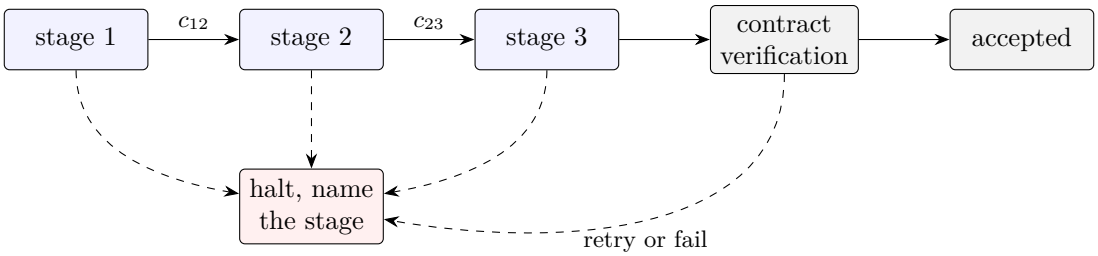

Figure 2: A pipeline with a halting failure rule and a final contract check. Each edge c_{ij} is a team contract naming stage i as producer and stage j as consumer. The first failing stage halts the run and its identity is carried in the failure value, so the caller learns which stage failed rather than only that the run did.

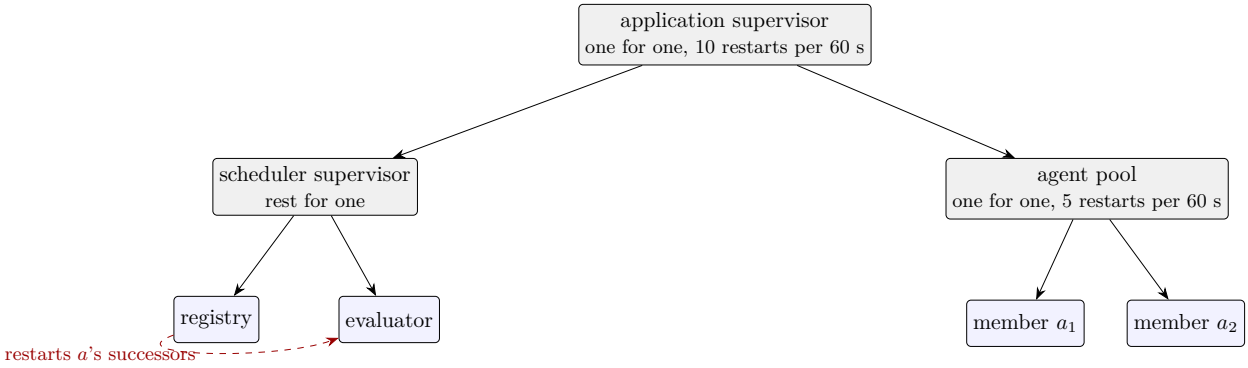

Figure 3: A supervision tree with restart strategies labelled. Children are drawn left to right in startup order, and that order is what the strategies act on. Under one for one a failing child is restarted alone. Under rest for one a failing child is restarted together with every sibling started after it, and no sibling started before it, which is the blast radius a startup dependency ordering requires: the registry is started first, so its failure restarts the evaluator, while an evaluator failure leaves the registry running. Exceeding the restart intensity within the period terminates the subtree rather than restarting forever.

That check has exactly two outcomes, success and a retry signal carrying a reason, and its declared specification admits no other [36, agent-os at 61cb3994da, same file, lines 74 to 109, and `src/agent_os/lib/agent_os/contracts/verify.ex`, line 21]. The failure value a caller sees comes from the enclosing fold over stages, which halts before the check runs, not from the check itself. The retry signal is the detail worth copying. A check with only pass and fail cannot express the common case in which the artifacts are wrong in a way the pipeline could repair.

Pipelines compared with teams. A three-phase localize, repair, and validate pipeline with no agentic loop resolved 32.00 percent of SWE-bench Lite at seventy cents per instance [2]. When a task decomposes cleanly, a fixed sequence is cheaper, more predictable, and easier to debug than a team. The burden of proof sits with whoever proposes the team.

4.3 Supervisor tree

Requirements. A tree of supervisors over members, a restart strategy at each supervisor, and an intensity and period at each supervisor.

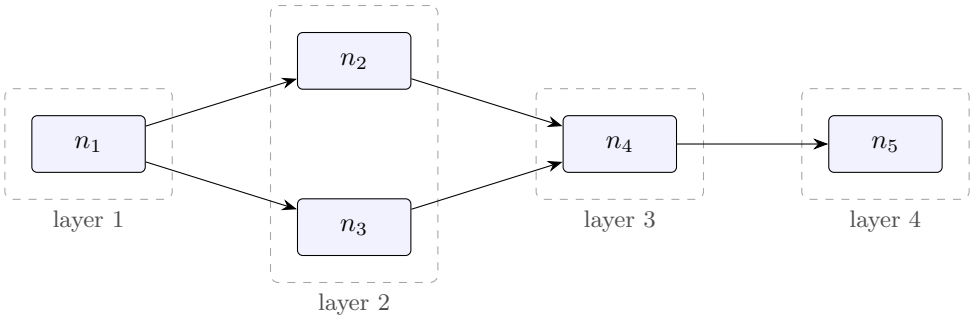

Figure 4: Layered execution of a directed acyclic graph. A layer is the set of nodes whose producers have all completed. Nodes inside a layer are candidates for concurrent dispatch; the transition between layers is a barrier. If at any point the set of ready nodes is empty while nodes remain, the graph contains a cycle and the run is rejected before any node executes.

Guarantees. A failed member does not silently become a success, and a member that fails persistently causes a bounded amount of restarting before the failure is propagated upward as a definite failure. Without an intensity bound, supervision converts a permanent fault into an unbounded retry loop.

Failure handling. The OTP reference names four strategies: `one_for_one`, `one_for_all`, `rest_for_one`, and `simple_one_for_one`. It makes a supervisor responsible for starting, stopping, and monitoring its child processes, and gives it an intensity and a period governing when it terminates all children and then itself [34].

One Elixir agent system uses two strategies at two levels. Its scheduler application starts a registry, an agent registry, an evaluator, a scheduler, and a pool supervisor under `rest_for_one` with a maximum of ten restarts in sixty seconds [36, agent-os at 61cb3994da, `src/agent_scheduler/lib/agent_scheduler.ex`, lines 42 to 57]. That is the right strategy when later children depend on earlier ones. Its agent pool is a dynamic supervisor using `one_for_one` with a maximum of five restarts in sixty seconds, on the stated reasoning that pooled agents are independent and share no state [36, agent-os at 61cb3994da, `src/agent_scheduler/lib/agent_scheduler/supervisor.ex`, lines 15 to 24 and 176 to 183]. Both bounds are far more permissive than the OTP default of one restart in five seconds. A reader can weigh that choice because it is written down.

A documentation drift finding. The same repository's top-level application module carries a comment describing a supervision tree over four subsystems started in dependency order: memory, then tools, then scheduler, then planner. The `start/2` function at the same commit starts three children, none of them those four, under `one_for_one` [36, agent-os at 61cb3994da, `src/agent_os/lib/agent_os/application.ex`, lines 1 to 27]. We report this rather than correct it in passing, because it is the class of defect an integration review exists to catch: a document and a program that each pass their own checks and disagree with each other.

4.4 DAG execution

Requirements. A set of member nodes, a set of edges standing for team contracts, and a rule for what happens downstream of a failed node.

Guarantees. Cycle freedom, checked before any node runs, and the property that a node begins only after every producer it declares has completed.

Failure handling. The layering algorithm carries the cycle check as a structural consequence rather than as a separate pass. In one implementation the executor repeatedly collects the nodes with no remaining dependencies, and if that set is empty while nodes remain it returns a cycle error [37, AgentHero at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 1027 to 1083]. Ready nodes are sorted by their declaration index before dispatch, which makes the layer contents deterministic even though the execution inside a layer is concurrent.

The node taxonomy. The same runtime types its nodes into fifteen kinds, among them an agent call, a synthesizer, a verifier, a gate, a loop, a branch, a map, a nested graph call, and a human approval [37, AgentHero at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 109 to 125]. Those kinds map closely onto the five composition operators of Definition 3.15 of Part II. Edges carry sequence, a layer carries parallel, the branch and map kinds carry branch, the loop kind carries bounded loop through its declared maximum rounds and continue key [37, AgentHero at 1c3ad24011, same file, lines 188 to 196], and a node's retry policy carries supervise. Only the approval kind has no counterpart in the operator list, and it is where a human enters a composition of agents.

The static graph. The graph is static. The repository's own design documentation states that it is compiled ahead of time rather than recursively planned, and that the agent may not mutate or repair the graph structure at run time [37, AgentHero at 1c3ad24011, `docs/agenthero_platform_restructuring_plan.md`, section 1]. Choosing DAG execution therefore means choosing predictability over adaptivity.

5 Team contracts and typed handoffs

A team is held together by what its members promise each other. Definition 3.4 of Part II defines a contract as a written, machine-readable statement of what an agent must produce and how the result will be checked, authored before the agent runs and evaluated after it stops. A statement qualifies only if a program can decide whether it was satisfied without asking the model that produced the work.

Definition 2.2 adds a producer and a consumer. The addition looks small and changes the engineering considerably. The party who must be prevented from marking their own work is now another agent, and no human sits in the path to notice.

A team contract in practice looks like the following. The syntax is illustrative rather than any one system's.

```
contract: localize_to_repair
  producer: localizer
  consumer: repairer
  shape:
    type: object
    additionalProperties: false
    required: [files, ranked_symbols, confidence]
    properties:
      files:          { type: array, items: { type: string }, maxItems: 20 }
```

```
    ranked_symbols: { type: array, items: { type: string } }
    confidence:     { type: number, minimum: 0, maximum: 1 }
acceptance:
  - shape_valid
  - every_file_exists_at_base_commit
  - files_nonempty
on_reject: one corrective retry, then fail the node
```

The shape is closed. Setting `additionalProperties` to false stops a producer smuggling an extra field past the boundary. One control plane applies the same discipline to its wire types, marking its team protocol and coordination structures so that unknown fields are rejected on deserialization rather than ignored [37, AgentHero at 1c3ad24011, `crates/agent-runtime/src/team_protocol.rs`, lines 20 to 60, and `crates/dag-runtime/src/lib.rs`, lines 404 to 420]. An agent that starts adding helpful commentary to a structured handoff then fails loudly at the boundary instead of quietly at the consumer.

The acceptance check does not rely on the producer. Two of the three checks in the listing are decidable against the file system rather than against the producer's own account of its work. Definition 3.8 of Part II makes the underlying rule: asking a model whether its own output is correct is not independent verification. At team scale the rule becomes a proposition about evidence classes.

Proposition 5.1 (Producer self-assessment is not independent evidence)**.** *Let $c = (p, q, S, \chi)$ be a candidate team contract. If χ consumes a self-assessment produced by p, that value does not constitute an evidence class independent of p. An otherwise independent check remains valid if it also records the self-assessment, but the assessment cannot replace evaluating the artifact against S.*

Argument. The artifact and self-assessment share a producer and therefore may share a failure cause. Evaluating S on the artifact through a distinct procedure supplies an independent check; merely accepting the producer's report does not. Adding the report as diagnostic context neither creates nor destroys the independence of that distinct check. The conclusion concerns evidence classes, not whether self-review can sometimes find defects. □

Identity from outside the artifact. The same control plane carries an execution context alongside every team task. It holds the team, member, and task identities, the team and task revisions, a lease token, a monotonic fence token, and the attempt number. The scheduler supplies that structure outside the application-controlled task payload, so a model may report the fields back but cannot choose or alter any of them [37, AgentHero at 1c3ad24011, `crates/agent-runtime/src/team_protocol.rs`, lines 16 to 39]. The design rule generalizes. Anything the team's bookkeeping depends on must live where the agent cannot write it, or the bookkeeping is advisory.

The failure this prevents. A vendor's software development kit exposes handoffs and guardrails as separate primitives, guardrails being validation of agent inputs and outputs [31]. Separating them is right. A handoff without a guardrail is how one agent's mistaken interpretation becomes the next agent's premise. A coding-agent vendor arguing against multi-agent architectures states the failure directly: actions carry implicit decisions, and conflicting decisions carry bad results [15].

A typed handoff makes the artifact explicit and leaves the decision behind it implicit. That is the limit of the technique, and Section 10 returns to the vendor's argument.

6 The shared board and the ownership rule

Members of a team learn about each other in one of two ways. They message each other, or they read a shared record. The two cost differently. Under direct messaging, telling everyone something takes one message per recipient, which is what one vendor's team documentation says: to reach everyone, send one message per recipient [13]. With a board, publishing is one write regardless of how many members read it.

6.1 Purpose of a board

A board is where a member publishes state that other members depend on. One vendor's team consists of a lead, teammates, a shared task list, and a mailbox. Each agent's mailbox is a file on disk and the task list is a directory of task records. Tasks have three states and can declare dependencies, and a pending task with unresolved dependencies cannot be claimed until those dependencies complete [13].

A production control plane holds the same three objects as database tables. Its module header describes them as a PostgreSQL-authoritative roster, task, and mailbox contract [37, AgentHero at 1c3ad24011, `crates/orchestrator/src/agent_teams.rs`, line 1]. Its public interface amounts to an inventory of what a board must support: create a team, provision and bind a member, drain the team, create, update, claim, heartbeat, complete and fail a task, enqueue, claim and acknowledge a message, and reconcile [37, AgentHero at 1c3ad24011, same file, functions at lines 1413, 1647, 1793, 1922, 2019, 2149, 2256, 2337, 2392, 2489, 2556, 2700, 2787 and 2861].

The append-only requirement in Definition 2.3 exists for attribution. It is the team-scale form of the immutable artifact of Definition 3.14 of Part II. A member who can edit an entry in place leaves the board recording the current state but not who put it there, and a disagreement about what was decided becomes unresolvable. The same control plane preserves attribution by incrementing a revision on every canonical team transition and by typing each transition's authenticated origin as an agent bound to a logical member, an authenticated operator, or a system component [37, AgentHero at 1c3ad24011, same file, lines 16 to 35].

6.2 Ownership and a worked conflict

A conflict that needs no exotic conditions shows why an ownership rule (Definition 2.4) has to be mechanical.

The storage model decides whether the failure is possible at all, so state it first. Assume what both implementations examined here assume: members share one checkout on one machine and write to the working tree directly. That is the default in every system cited. One vendor's team feature runs teammates as separate sessions in the same project directory and warns about this case [13]. The layered executor gives a node its own scratch working directory only when a container sandbox is declared, and its default sandbox policy is none, meaning the host [37, AgentHero at 1c3ad24011, `crates/agent-runtime/src/types.rs`, lines 100 to 118]. The checkout is a repository, so version control is present, but no member is on its own branch.

Isolation is available as a product decision rather than a default. One vendor's cloud agents run in isolated virtual machines with full development environments instead of on the local machine [17]. That removes the shared working tree, and with it both the overwrite and the shared view.

A team of three is assigned a refactor. Member a_1 owns the request parsing module, a_2 owns the response serialization module, and a_3 owns the tests. All three need to change one shared file, the type definitions, because the refactor changes a struct that all three modules use. Nobody said who owns the type definitions.

Without an ownership rule, the sequence runs as follows. Member a_1 reads the type file, adds a field, and writes the file. Member a_2 read the same file before a_1's write, adds a different field, and writes the file, discarding a_1's change. Both members' own verification passes, because each compiled against the file as it existed when that member read it. Member a_3 writes tests against whichever version it happened to read. Nothing has failed. Three individually passing outputs have produced a repository state that no member intended, and the first signal is a compile error in continuous integration that names none of the three as the cause. One vendor's documentation states the risk in one sentence and gives the same remedy this section gives: two teammates editing the same file leads to overwrites, so break the work so each teammate owns a different set of files [13].

Branching. A practitioner's first instinct is that version control should have caught this. It catches part of it. Giving each member its own branch or worktree turns the silent overwrite into a merge conflict at integration time, which is a real improvement, because the loss becomes visible and attributable.

Three things survive branching. A textual merge succeeds whenever the two members edited different lines of the same file, which is exactly what happens when each adds its own field, so no conflict is raised and the semantic incompatibility passes into the merged result. Someone must resolve the conflicts that are raised, and if that someone is an agent, it is an agent making a decision no member's contract covers. And isolation costs the team the thing it was assembled for: a member on its own branch cannot see what the others decided, which is the context-starvation failure of Table 2 arriving by another route.

Branch isolation and an ownership rule are complementary. The ownership rule is the one that prevents rather than detects.

An ownership rule assigns the shared object. Writing $\omega(\texttt{types.rs}) = a_1$ makes a_1 the only member permitted to change it. Members a_2 and a_3 request the change through the board rather than making it, and the board's record of that request is what an integration review later reads. The rule earns its name only when the detection is mechanical, which is where the engineering lies.

The same control plane makes it mechanical on the task-claiming path. A member claiming a task must hold a named capability. The claim then selects a ready task for the member's role whose required capabilities are a subset of the member's capabilities and whose write scopes are a subset of the member's write scopes, using a row lock that skips rows another transaction already holds, and increments a monotonic fence token and a revision on the task [37, AgentHero at 1c3ad24011, `crates/orchestrator/src/agent_teams.rs`, lines 2256 to 2300].

The subset condition on write scopes is the ownership rule made checkable. A member cannot claim work that would require writing outside the scopes it was provisioned with, and the check runs in the database at claim time rather than in a prompt at generation time. The manifest constrains what scopes a role may be given at all, requiring every declared worker role to carry a named work capability and a set of write scopes validated for safety [37, AgentHero at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 387 to 420 and 909 to 969].

The vendor implementation reaches the same requirement by another route: task claiming uses file locking to prevent race conditions when multiple teammates try to claim the same task at once [13]. Two independent systems putting a lock on the claim path is good evidence that the claim path is where an ownership rule has to live.

Proposition 6.1 (Attribution from ownership and an append-only board)**.** *Let $T = (A, C, B, \omega)$ be a team in which every write to a shared object o with $\omega(o)$ defined is recorded on the board B as an entry carrying the writing member's identity, and in which B is append-only. Then a write to*

o by a member other than $\omega(o)$ is detectable by a check that reads only B and ω, and the check is decidable in time linear in the number of entries mentioning o.

Argument. Scan the entries of B mentioning o and compare each entry's writer identity against $\omega(o)$. Append-only guarantees that an entry present at write time is present at check time, so no violating write can be removed from the record after the fact. The linear bound is the scan. The proposition does not claim that the violating write is prevented, only that it is detectable. Prevention requires the claim-time enforcement described above, and detection after the fact is what a board buys on its own. □

6.3 Memory shared between agents

A board is not memory. The two differ in the scope of their persistence. A board holds the state of one task while a team works on it and is torn down when the team is. In one vendor's implementation the team configuration directory is removed when the session ends, while the task list directory persists locally under the same retention settings that govern session transcripts [13].

Persistent memory is stored state that outlives a session and is retrieved selectively into a later session's context. It is a different object with different design problems, and Part V owns it and defines it. Everything this Part says about a board concerns within-task coordination. A reader who wants to know how a team's findings survive into the next task should read Part V.

7 Functional thinking at team scale

Definition 3.16 of Part II names functional thinking as the discipline of designing an agent system as a composition of pure and effectful steps joined by typed boundaries, and argues that the skill agentic engineering demands is designing and checking compositions rather than hand-writing steps. A team is the largest composition the series treats, and at team scale that skill reduces to one rule.

7.1 A transferable rule

A companion formal working series, described below, yields one rule a practitioner can apply without any of its machinery. In its vocabulary an object carries a support, the finite set of names it mentions, and combination is defined only when two objects' supports are disjoint (Definition 7.1, `def:named-composition`). Restricted to that case, combination is order-independent and grouping-independent (Proposition 7.2, `prop:named-monoid`). The comparison map produced by combining two components is an isomorphism, meaning genuinely no interference, exactly when their footprints do not overlap (Theorem 7.7, `thm:strength-support`) [39].

Before combining two agents, list the names each one touches. Disjoint lists mean the combination is safe. Overlapping lists mean the overlap is where the bug will be.

The same criterion turns up independently in production code. The concurrency-admission function of Section 4.1 admits a pair of nodes for concurrent execution exactly when neither is exclusive and either one is pure, both are read only, or both declare non-empty resource sets that do not overlap, an undeclared set being treated as conflicting [37, AgentHero at 1c3ad24011, `crates/dag-executor/src/lib.rs`, lines 5485 to 5510]. That is footprint disjointness, with the pure and read-only cases as the two situations in which a footprint counts as empty and an undeclared footprint treated as unbounded rather than empty. The code cites no formal result and was written for operational reasons. A criterion reached twice, once by derivation and once by necessity, is worth a practitioner's adoption.

7.2 Limits

The companion series states three limits on what its structure can promise.

The cited formal result establishes a necessary condition at the modeled store boundary: output-key collisions destroy order independence (Corollary 5.12, `cor:no-algebra`) [38]. Output-key disjointness alone does not establish independence in a running system, whose siblings may share other effects. That is the formal counterpart of the worked conflict in Section 6.2, and it makes the conflict structural rather than an implementation oversight.

Passing validation does not mean the arrangement is wired correctly. The same work records that manifest validation in the system it examines contains no check that a declared input has a real supplier, so a broken wiring passes validation silently (Corollary 5.8, `cor:not-recogniser`) [38]. A schema check on each handoff does not tell a practitioner that the handoffs connect.

A structural ceiling bounds what any composition formalism of this kind can express. A node with two or more outputs whose values are correlated cannot be represented, because the formalism forces the outputs to be independent functions of the inputs (Theorem 7.2, `thm:multi-output`) [38]. No later version fixes that. It bounds the class of arrangements any such analysis covers. Agents routinely produce correlated outputs, a patch and the test meant to exercise it among them, so the bound bites.

7.3 Boundaries in the failure taxonomy

The main empirical support for the compositional argument comes from outside the formal work. A preprint analysing multi-agent systems developed a taxonomy of fourteen failure modes in three categories from 150 hand-annotated traces, with inter-annotator agreement of $\kappa = 0.88$, released alongside a dataset of more than 1,600 annotated traces across seven multi-agent frameworks [26]. Its three categories are specification and system design issues, inter-agent misalignment, and task verification.

Five of the fourteen modes are boundary defects in the sense this series uses. Failing to follow a task specification and failing to follow a role specification are literally specification-adherence failures between components. Being unaware of termination conditions is an undefined termination contract. Failing to ask for clarification is an interface so under-specified that clarification is impossible. Withholding information is an unenforced information contract. Each of the five is addressed by making the boundary explicit and checked, which is what Section 5 describes.

The other nine do not reduce to boundaries, and this Part does not pretend otherwise. They include reasoning failures inside a member, premature termination, and verification that is present but shallow. The split is taxonomic, not a frequency estimate. Five of fourteen named failure-mode categories are interface problems that composition discipline addresses; the trace data reported here do not show that those categories account for one-third of observed failures. A discipline that addresses five categories in a documented taxonomy is useful and is not a solution.

Remark 7.1 (What may not be claimed). The empirical question of whether typed boundaries help or hurt model performance is open and contested. One study reports a decline in reasoning ability under format restrictions; a rebuttal argues the comparison used non-comparable prompts between conditions and reports the opposite result under matched prompts, and its author sells structured-generation tooling. Neither this Part nor Part II may present typed boundaries as a measured improvement. The argument here rests on the failure taxonomy, which does not depend on the format question, and on the design case, and it is presented as a discipline argued from cases rather than an established effect.

8 Serialized resources

Some things a team needs cannot be duplicated: a rate-limited API key, a build directory, a database migration lock, a single reviewer process, a physical device. Definition 2.5 requires a member that cannot acquire the guard to wait rather than proceed. That requirement is stricter than it looks, because an agent denied a resource fails naturally by working around it.

Three mechanisms appear in the evidence. They are complementary, not alternatives.

Admission bounds. A counted semaphore limits how many members touch a resource class at once. In the layered executor this is the manifest's declared concurrency, and a permit is acquired before each node's task is spawned [37, AgentHero at 1c3ad24011, `crates/dag-executor/src/lib.rs`, lines 2357 to 2366]. An admission bound answers how many, and answers nothing about which.

Exclusive claims. A claim that at most one member can hold. In the database-backed board this is a row-level lock that skips rows already locked, so a second claimant takes a different task rather than blocking on the first [37, AgentHero at 1c3ad24011, `crates/orchestrator/src/agent_teams.rs`, lines 2274 to 2287]. In the file-backed board it is file locking on the task record [13]. An exclusive claim answers which member, and answers nothing about what happens if that member dies holding it.

Leases and fences. A claim with an expiry, renewed by a heartbeat, so that a member which stops responding loses the claim instead of holding it forever. The same control plane leases an application run for sixty seconds and renews it every fifteen, with an assertion in its own test suite that the lease is at least four times the heartbeat interval [37, AgentHero at 1c3ad24011, `crates/orchestrator/src/app_runs.rs`, lines 22 to 23 and 6955 to 6956]. A separate ten-second heartbeat tracks worker liveness [37, AgentHero at 1c3ad24011, `crates/orchestrator/src/scheduler.rs`, line 358]. Task claims carry a lease token and a monotonic fence token incremented on every claim [37, AgentHero at 1c3ad24011, `crates/orchestrator/src/agent_teams.rs`, lines 2296 to 2300].

The fence is what makes the lease safe. A member whose lease expired while it was paused wakes, tries to record completion, presents a stale fence, and is rejected. Without a fence, a lease only shortens the window in which two members believe they own the same work.

The four-to-one ratio between lease and heartbeat matters because getting it wrong produces a confusing bug. Set the lease too close to the heartbeat interval and an ordinary garbage-collection pause or a slow network call revokes a healthy member's claim. The team then develops intermittent duplicated work that looks like a model failure and is not.

This machinery comes from durable execution systems. One describes a workflow execution as a durable, reliable, and scalable function execution that resumes after failure by replaying against a recorded event history and validating generated commands against it [43]. A state-machine workflow service supplies the adjacent vocabulary of choice, parallel, map, retry, catch, and a callback pattern in which the workflow waits for an external token before continuing [40]. Agent orchestration reuses both. A practitioner is better off reading their documentation than deriving the same mechanisms again.

9 Integration review and the referee

Per-member verification and integration review fail on different things. Per-member verification asks whether member a_i's output satisfies its own contract. Integration review asks whether the

outputs of $a_1, \ldots, a_n$ are consistent with each other, and it can fail when every member has passed. That is its distinguishing property. The worked conflict in Section 6.2 is such a case: three passing members and a broken repository.

9.1 Integration review

Three classes of check apply, each cheap and each catching a real class of defect.

Definitional consistency asks whether each shared object is defined exactly once, by whichever member owns it. Reference resolution asks whether every reference one member makes to another member's output names something that exists. Interface agreement asks whether the shape each consumer assumed matches the shape its producer emits. All three are decidable without re-running any member, which is what keeps the review cheap enough to be worth doing.

Two node kinds in the layered executor sit at this position: a synthesizer, which consumes several members' outputs and produces a combined artifact, and a verifier, which checks an artifact against declared criteria [37, AgentHero at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 109 to 125]. The distinction between them is the distinction this section draws. A synthesizer combines and can therefore paper over a disagreement; a verifier checks and can therefore fail on one. A team whose only cross-member step is a synthesizer has combination without integration review.

9.2 The referee

Integration review resolves inconsistencies between artifacts. It does not resolve disagreements between verdicts. When two reviewing members examine the same artifact and one passes it while the other fails it, an integration review has nothing to compare, because both verdicts are internally consistent. Definition 2.7 exists for that case.

Evidence that the role is real and commonly absent comes from a system documenting its own gap. One control plane's remediation document lists the gaps its authors consider open. Under a heading naming the absence of an escalation policy, it states that escalating to a higher-tier model when two specialists disagree is a natural pattern at scale, that the current behaviour is the deterministic output of a single meta-reviewer call, and that a tie-breaker agent role together with a specialist disagreement detector would let the framework express ensembles [37, AgentHero at 1c3ad24011, `docs/agent-harness-foundation-remediation.md`, item S3]. The same document places escalation support in its tier-two backlog rather than in shipped work.

That is a first-party negative finding, which is why it carries weight. This system has specialist reviewers, a verifier ladder, and a meta-reviewer, and its authors thought carefully enough about the architecture to write a remediation document. It still has no defined outcome when two of its reviewers disagree. A practitioner assembling a team of mutually checking agents should assume they lack the role too, until they have built it.

Proposition 9.1 (Disagreement requires a decision rule)**.** *Let T be a team containing members a_i and a_j whose verification procedures both take the same artifact x as input and each return a verdict in $\{pass, fail\}$. If there exists an artifact x on which the two procedures return different verdicts, then the team's behaviour is determined only if its specification states a decision rule. A deterministic rule may resolve the disagreement without appointing a referee; if the rule leaves discretion, the specification must designate a referee in the sense of Definition 2.7.*

Argument. The team's next action after verification must be determined from the two verdicts. A rule such as "any failure rejects" or a fixed priority order supplies that function. Such a rule is a decision function, not a party, and therefore not a referee under Definition 2.7. If no rule is stated,

implementation details such as arrival order decide the outcome. If the stated rule escalates for judgment, the party making that judgment is the referee. □

The cheap remedy is often a deterministic policy. A referee is needed only when the policy deliberately leaves a discretionary decision, such as escalation to a human or a separate adjudicating model.

10 Failure handling across agents

Every mode in Part III's catalogue of single-loop failures still occurs inside every member of a team. Table 2 lists the modes that need two agents before they can occur.

Unbounded delegation. A vendor engineering post reports that early versions of its research agents spawned fifty subagents for simple queries and scoured the web endlessly for nonexistent sources [6]. Two independent systems bound the same behaviour by declaration. One requires a coordination declaration to name a maximum member count of at most sixty-four, a maximum task count of at most 4,096, and a maximum coordination depth that version one requires to be exactly one, and rejects any manifest violating these [37, AgentHero at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 951 to 969]. The other states as a limitation that teammates cannot spawn their own teammates and that only the lead can manage the team [13].

Both make depth one. Production coordination is one level deep, and a design that assumes recursive teams of teams is ahead of what either system supports.

Task-status lag. The vendor documentation lists it as a limitation in plain terms. Teammates sometimes fail to mark tasks as completed, which blocks dependent tasks, and the suggested response is to check whether the work is actually done and either update the status manually or tell the lead to prompt the teammate [13]. The dependency mechanism is only as reliable as a member's willingness to update a record about itself. That is a self-report, and Part III's rule about evidence before assertion applies to it.

Context starvation. Decomposing a coding task across agents means each agent acts on a partial view. Actions carry implicit decisions, and conflicting implicit decisions produce bad results. The recommended response is to share context and full agent traces rather than individual messages [15].

Nothing in the evidence assembled here refutes that argument. The typed handoffs of Section 5 make artifacts explicit and leave the decisions behind them implicit. The board of Section 6 lets a member publish a decision but does not oblige it to notice which of its actions encoded one.

The detection cell for this row in Table 2 carries a specific meaning, and the meaning matters for the rest of the series. Mechanical here means a procedure that decides the same way on the same input without consulting a model, which is what Definition 3.8 of Part II requires of verification. A model asked whether a consumer had enough of the producer's context is not a mechanical detector. It is probabilistic, its judgment comes from the same class of system whose failure it is being asked to find, and a negative answer from it is not evidence in any of Part II's evidence classes. Such an evaluator makes a useful sampling instrument for finding candidate failures. It cannot be the check a gate is built on. Reporting it as detection would be exactly the substitution this series argues against.

Cross-agent mode	Symptom	Detection	Remedy	Source
Conflicting implicit decisions	Two members produce individually valid but mutually incompatible outputs	An integration review, or a compile or test failure that names no member	An ownership rule with claim-time enforcement, and a shared board	[15], and category (ii) of [26]
Handoff shape violation	The consumer improvises around a malformed input, or crashes	An acceptance check at the boundary before the consumer runs	A closed schema that rejects unknown fields, with one corrective retry before failing	[31] and [37]
Silent wiring break	A declared input has no supplier, and validation passes anyway	A supplier-existence check across the whole graph	Add the supplier check, which is absent by default	Corollary 5.8 of [38]
Output-key collision	Two siblings write one key, so the result depends on completion order	A static key-disjointness check over each layer	Rename the key, or serialize the two members	Corollary 5.12 of [38], and [37]
Reviewer deadlock	Two verifiers disagree and no next action is defined	Compare the verdicts rather than combining them	A referee, or a stated policy naming which verdict decides	Item S3 of [37]
Unbounded delegation	A member spawns many helpers for a simple request	Member count and depth checked against a declared bound	Declared maximums for members, tasks, and depth, with nesting forbidden	[6], [37], and [13]
Stalled claim	A member holds work it is no longer doing	Lease expiry checked against a heartbeat interval	A lease with a monotonic fence token, reclaimed on expiry	[37] and [43]
Task-status lag	A completed task stays open and blocks its dependents	Inspection of the dependency graph against the actual artifacts	Manual status correction, or a completion hook	[13]
Restart storm	A member fails and is restarted indefinitely	The restart count within a declared period	An intensity and period bound that terminates the subtree	[34] and [36]
Context starvation on handoff	The consumer lacks the producer's reasoning and redoes or contradicts it	No mechanical detection located	Share full traces, or do not decompose the task	[15]

Table 2: Failure modes that require more than one agent. Modes internal to a single agent's loop are catalogued in Part III and are not repeated here. The last row records deliberately that no mechanical detector was located, because it is the mode on which the strongest argument against multi-agent architectures is built.

11 Orchestration behaviour in two code bases

Two systems have supplied evidence throughout, both read at pinned commits.

11.1 Supervision and staged pipelines in Elixir

The first is an Elixir and OTP umbrella application with a Phoenix interface that runs agent pipelines, pinned at commit 61cb3994da. Its orchestration contribution is supervision and sequencing.

Its stage runner is the pipeline of Section 4.2: an early-exit fold over declared stages, halting on the first failure with the failing stage named, followed by a contract check over the collected artifacts that returns success or a retry signal [36, agent-os at 61cb3994da, `src/agent_os/lib/agent_os/pipeline.ex`, lines 41 to 109]. Its supervision uses two OTP strategies at two levels with different intensities, described in Section 4.3.

Its planner contains a decomposer that maps a task to a graph of subtasks and topologically sorts them into levels. The docstring describes execution levels within which subtasks run in parallel, transitions between levels as barrier synchronizations, and a level count equal to the critical path length [36, agent-os at 61cb3994da, `src/planner_engine/lib/planner_engine/decomposer.ex`, lines 1 to 30]. That is Figure 4 written out in prose, in a different language, by a different design.

The supervision here is single-node, which bounds what it demonstrates. The repository's own internal review records a single-node bottleneck for shared order-book state and notes that distributed failure modes were avoided rather than solved. Read this as evidence about process supervision within one machine, not about distributed agent teams.

11.2 Layered graphs, leases and a team roster in Rust

The second is a Rust and Tokio control plane that validates declarative graph manifests and executes them as layered dependency graphs of typed nodes, pinned at commit 1c3ad24011. It supplies most of the mechanism in this paper: layering with cycle detection, the concurrency semaphore, the pairwise concurrency-admission check, the quorum gate, the node taxonomy, the retry policy, worker leases with heartbeats, task claims with fence tokens, and a database-authoritative team roster with tasks and a mailbox.

Its declared bounds are unusually explicit and are collected in Table 3. Its retry policies are two distinct mechanisms: a per-node retry carrying a maximum attempt count and a fixed backoff in milliseconds [37, AgentHero at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 264 to 269], and a provider-level backoff for rate-limit and server errors with a maximum of four attempts and a thirty-second cap, using exponential delay with jitter unless the provider supplied a retry-after value [37, AgentHero at 1c3ad24011, `crates/llm-adapter/src/retry.rs`, lines 11 to 22].

Its own documentation records what it does not do. The graph is compiled ahead of time and the agent may not mutate or repair it at run time. The retry cursor is process-local, with no persistent scheduler checkpoint table. One review-graph component is described as still decorative, with the executor walking specialist roles manually. There is no cost ceiling per unit of work, so a pathological input can cost far above an operator's intended cap. And there is no escalation or tie-breaker policy when specialist reviewers disagree [37, AgentHero at 1c3ad24011, `docs/agent-harness-foundation-remediation.md`, items B3, S2, S3 and 4, and `docs/agenthero_platform_restructuring_plan.md`, section 1]. Weigh the positive evidence and these limitations together. They come from the same authors about the same system.

11.3 Lineage

Both repositories and the companion formal working series cited in Section 7 share one author, who also wrote this paper. They are three artifacts from one lineage, not three independent confirmations. They are used here because they are readable at pinned commits and because their authors documented their own gaps, which is what makes them useful.

The companion formal working series supplies structural arguments and nothing else. It is a five-document category-theoretic treatment of three of the same code bases, in preparation rather than published, and its value here is the disjointness criterion of Section 7 together with three limits on what composition can promise.

Three qualifications bound that value. Its framing as a sibling of this series originates in this project, because nothing in that series describes a practitioner-facing counterpart. It reports no measurement of any kind, stating plainly that it contains no experiment, no measurement, and no benchmark [39]. And its own account of its status distinguishes internal consistency checks from external validation without claiming the latter. Its results therefore appear here as arguments about structure, never as established facts, each with the limitation its authors attach to it.

12 The cost of a team

The evidence on team benefits and coordination costs differs in quality. The costs are measured and public. The benefits are largely vendor-internal and unreplicated.

12.1 Measurements

Table 3 collects every coordination cost and bound we located, with its source and what it is a cost of.

One conflation circulates and needs correcting. The vendor engineering post about a research system states, in one sentence, that agents typically use about four times more tokens than chat interactions and multi-agent systems about fifteen times more than chats [6]. The agent-teams product documentation for a different system states that token costs scale linearly with teammate count and gives no multiplier at all [13]. Citing the first for the multiplier and the second for linear scaling is correct; merging them is not.

12.2 The case for a team

The strongest single result is a 90.2 percent improvement: a multi-agent system with a lead agent coordinating parallel subagents beat a single-agent baseline by that margin on an internal research evaluation [6].

Four qualifications attach to it. The evaluation is internal and non-public, so nobody can reproduce the result. The domain is open-ended research rather than software engineering. The comparison is against a single-agent baseline from the same vendor rather than a well-tuned fixed pipeline. And the fifteen-times token multiplier comes from the same post, so the improvement is bought at a cost the source itself reports.

Published multi-agent systems for software issue resolution report a range of results. One framework reported resolving 13.94 percent of SWE-bench issues, described as an eight-fold increase over direct application of the model it built on [24]. Two others each report 28.33 percent on SWE-bench Lite [14, 25]. Training-data work on a large synthetic corpus of 50,000 task instances from 128 repositories reports 40.2 percent pass at one on SWE-bench Verified for a 32-billion-parameter

Quantity	Value	Source and scope
Token use of agents against chat interactions	about 4×	A vendor engineering post reporting its own data for a research system [6]
Token use of multi-agent systems against chat interactions	about 15×	The same sentence of the same post, about the same system [6]
Improvement over a single-agent baseline	90.2 percent	The same post, reporting an internal and non-public research evaluation [6]
Cost scaling in teammate count	linear, with no multiplier stated	Product documentation for a different system [13]
Recommended starting team size	3 to 5 teammates	The same product documentation, stated as practical guidance [13]
Maximum coordination depth	exactly 1 in version one	Manifest validation, which rejects any other value [37]
Maximum team members and maximum tasks	64 members and 4,096 tasks	The same manifest validation [37]
Nested teams	not permitted	Product documentation, as a stated limitation [13]
Provider retry attempts and backoff cap	4 attempts and a 30 s cap	The provider adapter [37]
Default agent corrective retries and timeout	2 retries and 180 s	Agent specification defaults [37]
Run lease, lease heartbeat, and worker heartbeat	60 s, 15 s, and 10 s	Scheduler constants, with the lease asserted to be at least four times the heartbeat [37]
Supervisor restart intensity, default	1 restart per 5 s	The OTP reference default [34]
Supervisor restart intensity, as configured	10 per 60 s at the application level and 5 per 60 s in the agent pool	The two configured supervisors of the Elixir system [36]

Table 3: Coordination costs and declared bounds. The two token multipliers and the 90.2 percent figure come from one vendor post about one research system and are reported together because separating them would misrepresent the source.

model [42], though that is a training result rather than a multi-agent one. A generalist agent framework reports outperforming baselines across issue resolution, repository-level generation, and fault localization without stating numeric scores in its current abstract [20].

Every SWE-bench figure in this paragraph carries a discount. An audit of the benchmark found solution leakage in 32.67 percent of successful patches and weak tests behind 31.08 percent, and filtering both dropped one system's resolution rate from 12.47 percent to 3.97 percent [41]. A separate audit of the Verified subset by its largest institutional consumer found defective tests, which reject functionally correct solutions, in at least 59.4 percent of the problems examined, and that consumer stopped reporting the metric [33]. Part I collects the audits in its benchmark section. The resolution rates above are not unbiased estimates of capability, and comparisons between them inherit the same defects. Leakage can inflate a score, while defective tests can reject correct solutions; without measuring both effects on each system, the direction of the net bias is indeterminate.

Fragility of the figures. Two of those systems, built by different groups with different architectures, report the identical headline figure of 28.33 percent on SWE-bench Lite [14, 25]. We read both primary abstracts directly, and both state it, so this is not a citation error propagating through secondary sources. Nothing in the published material settles whether the coincidence is genuine or

reflects a shared property of the split, such as a common set of resolvable instances.

Neither paper is wrong. The lesson is about the instrument. A practitioner comparing multi-agent systems on single headline percentages is using a measurement that can return identical readings for different systems, and should treat small differences between such figures as noise.

12.3 The case against a team

Four lines of evidence bear against teams, and they differ in strength.

Fixed pipelines. A three-phase localize, repair, and validate process with no agentic decision-making resolved 32.00 percent of SWE-bench Lite at seventy cents per instance [2]. That is higher than the multi-agent systems above, at a cost the paper reports, and it carries the same benchmark discount as those figures. A related analysis of published agent benchmarks argues that state-of-the-art agents are needlessly complex and costly and that the community has reached mistaken conclusions about the sources of accuracy gains [3]. Both results concern issue resolution specifically.

Context sharing. A coding-agent vendor's position piece argues that decomposing coding work across agents disperses decision-making, that context cannot be shared thoroughly enough between agents, and that conflicting implicit decisions produce bad results [15]. Section 10 records that nothing here refutes it.

The failure taxonomy. Fourteen failure modes across three categories were developed from 150 hand-annotated traces at $\kappa = 0.88$ and released with more than 1,600 traces across seven frameworks [26]. Two of the three categories, inter-agent misalignment and task verification, exist only because there is more than one agent. A team does not merely inherit single-agent failures. It adds failures of its own.

A first-party decomposition experiment. An external preprint that Part II cites for its account of the layer surrounding the model compared direct prompting against a three-stage decomposition and against a compiled graph form of the same decomposition, on ten SWE-bench Lite instances with an eight-billion-parameter locally-hosted model. All three conditions resolved zero instances. A rerun against a second eight-billion-parameter model with a format-correction retry loop resolved zero of thirty submissions and recovered no patches through retry [10].

The author scopes that result carefully, and the scoping must travel with it. At one evaluated instance against zero, the experiment is under-powered to discriminate between architectures. The binding constraint it identifies is the model's ability to emit cleanly applying diffs, not the decomposition. And the author states that task-resolution gains require a substantially stronger model, outside that paper's local-only scope. With those qualifications, it remains the only first-party controlled comparison of a decomposition against direct prompting that we located, and it found no benefit.

12.4 Vendor retreat from multi-agent products

Table 4 records a pattern that is easy to miss because each event is announced as progress.

The same industry assessment places two adjacent techniques in its caution ring: codebase cognitive debt, defined as the growing gap between a system's implementation and a team's shared understanding of how and why it works, and coding throughput as a measure of productivity, with the observation that organizations often measure success using superficial indicators such as lines of

System	Date	What changed	Source
An early multi-agent orchestration framework	2024 to 2025	Described by its own repository as an educational framework exploring ergonomic, lightweight multi-agent orchestration, and now stated to be replaced by a production-ready software development kit	The repository [32]
A conversational multi-agent framework	January 2025	A complete redesign of the library, stated as improving code quality, robustness, generality and scalability	A vendor blog post [9]
The same framework's community fork	November 2024 onward	A governance fork that later diverged further, with version one stated not to be a drop-in upgrade and the original preserved separately	The repository [1]
The same framework's vendor line	October 2025	Converged with a separate product's enterprise-ready foundations into a unified, commercial-grade framework, which is an implicit statement that the earlier line was not enterprise-ready	A vendor blog post [30]
An agent teams feature	2026	Shipped experimental and disabled by default behind an environment variable, with nine stated limitations including no nested teams and no session resumption for in-process teammates	Product documentation [13]
Coding agent swarms	2026	Placed in the caution ring of an industry technology assessment, defined there as applying dozens to hundreds of agents with composition and size determined dynamically	An industry technology radar [44]

Table 4: Publicly documented retreats, rewrites, and hedges in multi-agent products. Each row is an announcement made by the vendor or maintainer of the system named. The pattern is not that multi-agent architectures failed. It is that every vendor line in this table has been rebuilt, relabelled, or gated within roughly two years of shipping.

code generated or the number of pull requests [44]. Both bear on teams directly, because a team multiplies output volume faster than it multiplies anyone's understanding of that output.

12.5 A decision rule

Use a team when the work decomposes into parts a single member can hold entirely, when members need to learn what other members decided, and when the result is worth an order-of-magnitude rise in token cost. Research, review from several independent angles, and investigation of competing hypotheses all fit, and are the cases the product documentation itself recommends [13].

Use a fixed pipeline when the decomposition is known in advance. A pipeline is cheaper, more predictable, and easier to attribute failures in than a team.

Use a single agent when the task needs one coherent view of the whole problem. That is the case the context-sharing argument identifies [15], and one vendor says the same about its own system: domains requiring all agents to share the same context, or involving many dependencies between agents, are a poor fit [6].

13 Current team boundaries

Four task classes show where the agentic engineer should retain a single agent, a fixed workflow, or direct authorship.

Real changes in large, mature code bases the engineer knows well. A randomized controlled trial gave sixteen experienced open-source maintainers 246 real issues in their own repositories, randomizing each issue to allow or disallow AI tooling. Allowing AI increased completion time by 19 percent. The same developers had forecast a 24 percent reduction beforehand and, after finishing, still estimated a 20 percent reduction [29]. Measurement and perception pointed opposite ways. The other randomized trial in the literature points the opposite direction: a completion assistant made the 35 freelancers who completed a single standardized greenfield task, an HTTP server written from scratch, 55.8 percent faster [35]. The two trials differ in population, task class, and tool, and Section 5 of Part I sets them side by side; the slowdown is a finding about the task class named here, not about assisted programming in general.

That is the clearest example we found of a class in which human effort moved without falling: high-context maintenance work in a familiar mature repository with high contribution standards. The correct control-layer decision is to preserve one coherent context and avoid coordination overhead.

Work no member can hold for long enough. Team size is not the only bound that matters. So is how long a member can usefully run before its output stops being trustworthy. A measurement programme tracking the length of task an agent can complete reports that the frontier time horizon has been doubling approximately every seven months since 2019, with the caveat that the trend may have accelerated in 2024 [28].

That bound is rising, not absent, and it constrains team design directly. A task whose smallest sensible unit exceeds what one member can hold does not become tractable by adding members, because every member faces the same bound. The vendor guidance to size tasks as self-contained units producing a clear deliverable, and to keep roughly five or six tasks per teammate, is the practical form of the same constraint [13].

Work with sequential dependencies or shared files. Theory and vendor guidance agree here. Product documentation states that for sequential tasks, same-file edits, or work with many dependencies, a single session or subagents are more effective [13]. The worked conflict of Section 6.2 explains why. A team's advantage is parallelism, and dependencies remove the parallelism while leaving the coordination cost in place.

Work whose decomposition is already known. A practitioner who can write down the stages in advance should use a fixed pipeline, which dominates on every axis measured here. The concrete instance is 32.00 percent on SWE-bench Lite at seventy cents an instance [2]. Run-time decision-making about which member acts next is worth paying for only when the decomposition genuinely cannot be fixed in advance.

Documented failures and reverts. Table 4 records this Part's documented-failure evidence: six publicly announced retreats, rewrites, or hedges by the vendors and maintainers of multi-agent systems, including a framework its own repository describes as educational and now replaced.

On the narrower question, we found no documented case of an organization that adopted agent teams broadly and then withdrew or restricted them for measured quality or cost reasons, in the sources reviewed (cutoff 2026-09-01). That absence is not evidence that no such reversion occurred, and it is not support for teams.

14 Limitations

The following would weaken this Part's claims.

Benefit evidence. One measured improvement figure carries the positive case, it is from a vendor about its own system on a non-public evaluation in a non-coding domain, and it has not been independently reproduced [6]. If a public replication found no improvement, the positive case in Section 12 would have essentially nothing left, and the cost figures would stand unopposed.

Repository lineage. Both code bases and the companion formal working series share one author, who wrote this paper (Section 11.3). If an independent system were found to implement coordination depth greater than one in production, the claim in Section 10 that production coordination is one level deep would need revision, since it currently rests on two systems from a narrow slice of the field.

Implementation claims. Every repository statement here was established by reading source at a pinned commit. None was established by running the systems and observing what they did. A lease constant in a file is evidence that the implementation declares that lease, not evidence that leases behaved correctly under load.

A task class where effort did not fall. Named in Section 13: real issues in large mature repositories the developer has worked on for years, where a randomized trial measured a 19 percent increase in completion time [29], against the 55.8 percent speed-up the other randomized trial measured for freelancers on a greenfield task [35]. The different results separate redefinition from uniform speed improvement. In both populations, engineering work is reorganized around directing, reviewing, and accepting machine-produced changes. The task class determines whether that organization saves time and how much human control it requires.

Open questions. The missing bridge from two-phase commit, leader election, and the consistency and availability tradeoff to agent orchestration, noted in Section 3, remains open. So does the identical 28.33 percent figure reported by two different multi-agent systems, which both primary abstracts state.

Light formalization. The three propositions in this Part are arguments, not proofs, and none formalizes team behaviour as a whole. A companion formal working series attempts more and reports a structural ceiling on what any such formalization can express, because correlated multiple outputs from one node fall outside it [38]. More formal weight is not available: the companion series is in preparation rather than published (Section 11.3).

15 Conclusion

A set of agents becomes a team when one member's input depends on another's output and the assignment is written where both can read it. Everything else in this Part follows from taking that dependency seriously.

Four arrangements cover the field, and all four are old. Fan-out with fan-in guarantees order-independence only when effect footprints are disjoint and the combining rule is commutative or canonically ordered. A pipeline guarantees that each stage sees the previous stage's output, and deserves to be a first choice rather than a fallback. A supervisor tree guarantees that failure propagates instead of looping forever, provided a restart intensity is declared. Directed-acyclic-graph execution generalizes the first two, buying predictability by giving up run-time adaptation.

Four mechanisms hold an arrangement together. A team contract is a typed boundary whose acceptance check cannot rely solely on the producer's self-assessment. A shared board makes state readable without one message per recipient, and its append-only property is what makes writes attributable. An ownership rule turns a convention into a check, and belongs on the claim path where a lock and a scope-subset test can enforce it. A deterministic policy can resolve disagreeing verdicts; a separate referee is needed when the policy leaves a discretionary decision.

The measured multipliers are large and public. The measured benefit is a single figure from a vendor about its own system on a non-public evaluation, reported in the same post as the multiplier. A fixed pipeline outperformed contemporary agent scaffolds on issue resolution at a cost its authors report. Six vendor lines have been rebuilt, relabelled, or gated within about two years. Both production systems examined here cap coordination depth at one.

None of that says teams do not work. It says the burden of proof sits with the team. This Part recommends the discipline that makes discharging that burden possible: name the members, write the contracts between them, assign every shared object an owner, serialize what cannot be duplicated, review the outputs against each other rather than one at a time, and decide in advance which policy or referee breaks a tie. A practitioner who does those five things has a team whose failures are attributable. That is the precondition for finding out whether the team was worth it.

References

[1] AG2 project maintainers. AG2 (formerly AutoGen): The Open-Source AgentOS. Repository. `https://github.com/ag2ai/ag2` (accessed 1 September 2026).

[2] Xia, C. S., Deng, Y., Dunn, S., Zhang, L. Agentless: Demystifying LLM-based Software Engineering Agents. Preprint, arXiv:2407.01489, 1 July 2024, revised 29 October 2024. `https://arxiv.org/abs/2407.01489`

[3] Kapoor, S., Stroebl, B., Siegel, Z. S., Nadgir, N., Narayanan, A. AI Agents That Matter. Preprint, arXiv:2407.01502, 1 July 2024. https://arxiv.org/abs/2407.01502

[4] Apache Airflow project. Project. Documentation for Airflow 3.3.1. https://airflow.apache.org/docs/apache-airflow/stable/project.html (accessed 1 September 2026).

[5] Anthropic. Building Effective AI Agents. Engineering blog, 19 December 2024. https://www.anthropic.com/engineering/building-effective-agents

[6] Anthropic. How we built our multi-agent research system. Engineering blog, 13 June 2025. https://www.anthropic.com/engineering/built-multi-agent-research-system

[7] Armstrong, J. Making reliable distributed systems in the presence of software errors. PhD thesis, Royal Institute of Technology (KTH), Stockholm, December 2003. Print source; no working primary URL located.

[8] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., Wang, C. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. Preprint, arXiv:2308.08155, 16 August 2023, revised 3 October 2023. https://arxiv.org/abs/2308.08155

[9] Microsoft Research. AutoGen v0.4: Reimagining the foundation of agentic AI for scale, extensibility, and robustness. Vendor blog, 14 January 2025. https://www.microsoft.com/en-us/research/blog/autogen-v0-4-reimagining-the-foundation-of-agentic-ai-for-scale-extensibility-and-robustness/

[10] Banu, B. Harness Engineering as Categorical Architecture. Preprint, arXiv:2605.12239 [cs.PL], 12 May 2026. Sections 6.4 and 6.4.1 cited. https://arxiv.org/abs/2605.12239

[11] Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., Sun, M. ChatDev: Communicative Agents for Software Development. Association for Computational Linguistics, 2024. Preprint arXiv:2307.07924, 16 July 2023, revised 5 June 2024. https://arxiv.org/abs/2307.07924

[12] Anthropic. Create custom subagents. Claude Code product documentation. https://code.claude.com/docs/en/sub-agents (accessed 1 September 2026).

[13] Anthropic. Orchestrate teams of Claude Code sessions. Claude Code product documentation, describing the feature as of version 2.1.178. https://code.claude.com/docs/en/agent-teams (accessed 1 September 2026).

[14] Chen, D., Lin, S., Zeng, M., Zan, D., Wang, J.-G., Cheshkov, A., Sun, J., Yu, H., Dong, G., Aliev, A., Wang, J., Cheng, X., Liang, G., Ma, Y., Bian, P., Xie, T., Wang, Q. CodeR: Issue Resolving with Multi-Agent and Task Graphs. Preprint, arXiv:2406.01304, 3 June 2024. https://arxiv.org/abs/2406.01304

[15] Yan, W. Don't Build Multi-Agents. Cognition, 12 June 2025. https://cognition.com/blog/dont-build-multi-agents

[16] CrewAI. Crews. Product documentation. https://docs.crewai.com/en/concepts/crews (accessed 1 September 2026).

[17] Cursor. Cloud Agents. Product documentation. https://cursor.com/docs/cloud-agent (accessed 1 September 2026).

[18] Erman, L. D., Hayes-Roth, F., Lesser, V. R., Reddy, D. R. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys* 12(2), 1980, pp. 213-253. DOI 10.1145/356810.356816

[19] Hewitt, C., Bishop, P., Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Stanford, 1973. Print source.

[20] Phan, H. N., Nguyen, T. N., Nguyen, P. X., Bui, N. D. Q. HyperAgent: Generalist Software Engineering Agents to Solve Coding Tasks at Scale. Preprint, arXiv:2409.16299, 9 September 2024, revised 5 September 2025. https://arxiv.org/abs/2409.16299

[21] LangChain. Graph API overview. LangGraph documentation. https://docs.langchain.com/oss/python/langgraph/graph-api (accessed 1 September 2026).

[22] LangChain. LangGraph Multi-Agent Supervisor. Repository. https://github.com/langchain-ai/langgraph-supervisor-py (accessed 1 September 2026).

[23] Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7(1), 1985, pp. 80-112. DOI 10.1145/2363.2433

[24] Tao, W., Zhou, Y., Wang, Y., Zhang, W., Zhang, H., Cheng, Y. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. Preprint, arXiv:2403.17927, 26 March 2024. https://arxiv.org/abs/2403.17927

[25] Arora, D., Sonwane, A., Wadhwa, N., Mehrotra, A., Utpala, S., Bairi, R., Kanade, A., Natarajan, N. MASAI: Modular Architecture for Software-engineering AI Agents. Preprint, arXiv:2406.11638, 17 June 2024. https://arxiv.org/abs/2406.11638

[26] Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., Zaharia, M., Gonzalez, J. E., Stoica, I. Why Do Multi-Agent LLM Systems Fail? Preprint, arXiv:2503.13657, 17 March 2025, revised 26 October 2025. https://arxiv.org/abs/2503.13657

[27] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., Schmidhuber, J. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. Preprint, arXiv:2308.00352, 1 August 2023, revised 1 November 2024. https://arxiv.org/abs/2308.00352

[28] Kwa, T., West, B., Becker, J., Deng, A., Garcia, K., Hasin, M., Jawhar, S., Kinniment, M., Rush, N., Von Arx, S., Bloom, R., Broadley, T., Du, H., Goodrich, B., Jurkovic, N., Miles, L. H., Nix, S., Lin, T., Painter, C., Parikh, N., Rein, D., Sato, L. J. K., Wijk, H., Ziegler, D. M., Barnes, E., Chan, L. Measuring AI Ability to Complete Long Software Tasks. Preprint, arXiv:2503.14499, 18 March 2025. https://arxiv.org/abs/2503.14499

[29] Becker, J., Rush, N., Barnes, E., Rein, D. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. METR research report; preprint, arXiv:2507.09089, 12 July 2025. https://arxiv.org/abs/2507.09089

[30] Microsoft. Introducing Microsoft Agent Framework. Azure blog, 1 October 2025. https://azure.microsoft.com/en-us/blog/introducing-microsoft-agent-framework/

[31] OpenAI. OpenAI Agents SDK. Product documentation. https://openai.github.io/openai-agents-python/ (accessed 1 September 2026).

[32] OpenAI. Swarm. Repository. https://github.com/openai/swarm (accessed 1 September 2026).

[33] OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. Vendor limitations statement; exact publication date not verified. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/

[34] Ericsson AB. Supervisor Behaviour. Erlang System Documentation, version 29.0.6. https://www.erlang.org/doc/system/sup_princ.html (accessed 1 September 2026).

[35] Peng, S., Kalliamvakou, E., Cihon, P., Demirer, M. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. Preprint, arXiv:2302.06590, 13 February 2023. Randomized controlled trial. https://arxiv.org/abs/2302.06590

[36] Long, M. agent-os. Private repository, read at commit 61cb3994da (24 March 2026), accessed 1 September 2026. Claims are implementation-level statements established by source inspection at that commit.

[37] Long, M. AgentHero platform. Private repository, read at commit 1c3ad24011 (1 September 2026), accessed 1 September 2026. Claims are implementation-level statements established by source inspection at that commit.

[38] Long, M. Skills as operad operations: DAG composition in AgentHero. In: the companion formal working series (2026), in preparation. Read at commit e48408972b, `papers/latex/skills-operad.tex`, accessed 1 September 2026. Results cited by number and label.

[39] Long, M. A modular account of the agent harness. In: the companion formal working series (2026), in preparation. Read at commit e48408972b, `papers/latex/synthesis.tex` and `team/integration-report.md`, accessed 1 September 2026. Results cited by number and label.

[40] Amazon Web Services. What is Step Functions? AWS Step Functions Developer Guide. `https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html` (accessed 1 September 2026).

[41] Aleithan, R., Xue, H., Mohajer, M. M., Nnorom, E., Uddin, G., Wang, S. SWE-Bench+: Enhanced Coding Benchmark for LLMs. Preprint, arXiv:2410.06992, 9 October 2024. `https://arxiv.org/abs/2410.06992`

[42] Yang, J., Lieret, K., Jimenez, C. E., Wettig, A., Khandpur, K., Zhang, Y., Hui, B., Press, O., Schmidt, L., Yang, D. SWE-smith: Scaling Data for Software Engineering Agents. Preprint, arXiv:2504.21798, 30 April 2025, revised 21 May 2025. `https://arxiv.org/abs/2504.21798`

[43] Temporal Technologies. Temporal Workflow Execution overview. Product documentation. `https://docs.temporal.io/workflow-execution` (accessed 1 September 2026).

[44] Thoughtworks. Technology Radar, techniques. Entries for coding agent swarms, codebase cognitive debt, and coding throughput as a measure of productivity, all in the caution ring at the access date. `https://www.thoughtworks.com/radar/techniques` (accessed 1 September 2026).