The Anatomy of Agentic Engineering

Part II 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

Software is increasingly produced by directing programs that write, run, and repair code, but the vocabulary for what is built has not settled. This paper asks what the components of an agentic engineering system are, how they layer, and what a practitioner does with them. It defines nine primitives so that an observer or a program can check whether a system has each one: tool, context window, skill, contract, sandbox, harness, orchestrator, verification, and evidence class. Each is located in source at pinned commits in four first-party code bases. Every primitive is either a step or a boundary between steps, seven further definitions make that reading precise, and five composition operators cover the arrangements that appear in practice. A six-layer reference architecture is proposed rather than reported as consensus. Verification excludes any procedure that consults the producing agent, because a frontier model recognized its own output with 73.5% accuracy and that recognition correlated with self-preference. Effective context is smaller than advertised context: 11 of 13 models claiming at least 128K context fell below half their short-context baseline at 32K. The case studies expose the control work that the new discipline requires. Six of the twelve tools one harness declares are stubs, its plan assesses it at 42% complete, and its contract verifier passes any rule it does not recognize. Engineers must therefore verify implementations rather than infer capability from declared architecture. The result is an operational vocabulary grounded in code-level instances; comparative measurement of the composition operators remains future empirical work.

1 Introduction

A developer in 2026 opens a terminal, states an intention in a sentence, and watches a program read files, edit them, run a test suite, read the failures, and edit again. Something is being engineered. It is less clear what.

The parts have names. The names are doing too much work. "Agent" carries so many incompatible readings across subfields that it has been argued to impede research communication, and a redefinition has been proposed [13]. "Harness" arrived later and is vaguer still.

The cost of that vagueness is practical. When a practitioner says "my agent is slow," the sentence could be about the model, about how the context was assembled, about which tools were exposed, or about a permission prompt. Those are four different problems with four different fixes.

This paper is Part II of a five-part series. Part I defines the terms used here and not redefined: agent (its Definition 4.2), autonomy level (4.3), unit of work (4.5), replacement in the operational sense (4.6), and agentic engineering (4.7).

Part I establishes the series' central claim: 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. Engineering judgment becomes the control layer that defines objectives, constrains effects, composes the system, and decides whether its evidence is sufficient.

Research question. RQ2. What are the components of an agentic engineering system, how do they layer, and what does a practitioner actually do with them?

An agentic engineering system has nine primitives: tool, context window, skill, contract, sandbox, harness, orchestrator, verification, and evidence class. Each is defined here so that an observer or a program could check whether a given system has it, and each is then located in running code at a pinned commit.

Those primitives occupy six layers: model, harness, tools, state, orchestration, and governance. The layering is a proposal rather than a consensus, because no settled alternative exists to adopt instead. The vendor pattern vocabulary names patterns rather than layers [3]. The closest academic work is a research roadmap [24]. Two agent-component surveys decompose an agent rather than a deployment [33, 52].

Every one of the nine primitives is either a step or a boundary between steps. Tools are steps that change something outside the model, with a declared schema on the way in. A contract is a boundary between a human and an agent. A sandbox makes a step's reach nameable. Verification happens at a boundary.

That observation turns a parts list into a design vocabulary, and it is why this series argues that the shift rewards functional thinking. Agent systems get built by composing parts with explicit inputs and outputs across typed boundaries, keeping effects explicit and recorded, treating artifacts as immutable values, and combining behaviour from a small set of operators. Five operators cover the arrangements that appear in practice.

Scope. This Part does not describe control flow. The plan, act, and verify cycle is Part III's loop, its Definition 2.2, with the gate at 2.1 and bounded iteration at 2.4. Those definitions are cited forward and not restated. Behaviour across several agents belongs to Part IV. Memory and governed data belong to Part V; this Part names the state layer and stops. The illustrative workflow covers roles, artifacts, and handoffs only.

Evidence discipline. Every repository claim here was established by reading source at a pinned commit, and takes the form "the implementation at commit X contains Y." Nothing here was executed. External claims carry their source, and where sources disagree both sides appear in the same passage.

2 Motivation

Three vocabularies exist for this material and none of them is a stack.

The vendor pattern vocabulary separates workflows, where models and tools run through predefined code paths, from agents, which direct their own processes and tool use at run time. It then names five workflow patterns: prompt chaining, routing, parallelization, orchestrator-workers, and evaluator-optimizer, with fully autonomous agents held apart as a separate case [3]. Section 5 uses all five.

What the pattern vocabulary cannot do is locate anything. Patterns are shapes of control, not positions in a system. Knowing that a design is an evaluator-optimizer says nothing about where the sandbox is or who assembled the context.

The academic agent vocabulary decomposes an agent into planning, memory, perception, and action, across a survey of 124 papers [33], with a companion survey covering the same ground [52]. That decomposition describes the inside of an agent. A practitioner deploying one needs to know what is outside it: which tool registry it draws from, what its writes can reach, who decided it should run at all.

The closest work to this Part's subject proposes the distinction between software engineering for humans and software engineering for agents as the load-bearing one, and presents itself as a set of foundational pillars and a research roadmap rather than a completed architecture [24]. A roadmap is the right form for that paper and the wrong form for a practitioner who has to decide, today, where to put a permission check.

One external source does supply a decomposition of the harness layer, and this Part uses it. Banu's preprint argues that the properties practitioners want a harness to guarantee are attributes of the harness's structure rather than of the model. It adopts the four-pillar decomposition of Zhou et al., cited there, of memory, skills, protocols, and harness engineering, and maps each pillar onto a component of its own architecture triple [11]. The fourth pillar is named harness engineering, not orchestration. Orchestration is what that pillar subsumes.

Two of that preprint's other commitments matter here. It defines a structural guarantee as a certificate triple: a theorem statement, a symbol map, and a mechanically replayable derivation. And it treats the deployment map as a parameter rather than a fixture. It states plainly that functor laws alone are insufficient for the guarantees it wants, which is why it adds identity and replay checks.

Its empirical content is small and partly negative. The certificate-preservation study covers five target frameworks, uses structural verification only, has no baseline, and rests on a single reference implementation. Its own stated limitations include that all validation uses one code base, and that its certificates cover structural invariants but not behavioural properties, with "the agent never hallucinates" named as out of scope [11]. Its task-resolution results at small model scale are negative; they belong to Part IV, where multi-agent decomposition is the subject.

The gap these three vocabularies leave is a layered account of the deployed system, with each layer's job stated and each primitive placed in it. This Part proposes one. The alternative is that every team invents its own and the terms drift further apart.

3 Definitions

Sixteen objects are defined here: nine primitives, and seven terms that carry the compositional reading. None uses the term it defines, and none depends on a term defined only by reference back to it.

One word in the second group does double duty. A *step* is any unit of work with declared inputs and an output. Definitions 3.10 and 3.11 use it in that plain sense, because they describe the joins between steps and do not depend on what a step contains. Definitions 3.12 and 3.13 then split steps by what each touches. The order is a presentation choice, not a dependency.

On stipulation. This field has no canonical coining event. Research on the provenance of "agentic engineering" found no single origin. Several parties converged on similar phrasing between roughly mid-2025 and mid-2026, and a widely repeated attribution of the coinage to a named individual in

February 2026 is unconfirmed; this paper does not assert it [49]. "Agent" itself has been argued to be diluted past utility [13].

Several definitions below are therefore this series' own stipulations rather than reported consensus, including all seven in Section 3.2. Each says so where it appears.

3.1 The primitives

Definition 3.1 (Tool)**.** A *tool* is a named, callable function exposed to a model with a declared input schema, whose invocation produces an effect or an observation outside the model. A capability is a tool only if the model can name it and the harness can execute it; a capability described only in prose is not a tool.

The test is mechanical. Read the system's tool declarations. If a capability is absent from them, the model cannot select it, whatever the prompt says about it. The declared-schema requirement is not decoration: the interoperability standard defines each tool as identified by a name and carrying metadata describing its schema, with input and output JSON Schemas [34].

Definition 3.2 (Context window)**.** The *context window* is the bounded sequence of tokens a model attends to on a single inference call. Its operational size is not its advertised size: *effective context* is the length beyond which measured task accuracy falls below a stated fraction of the same model's short-context accuracy.

The second sentence is the part that changes designs, and Section 4.1 gives the measurements behind it.

Definition 3.3 (Skill)**.** A *skill* is a named, reusable instruction set loaded into an agent's context on demand rather than at every turn, which changes how the agent performs a class of task without changing the model or the tools. A skill is distinguishable from a prompt by being addressable by name and reused across sessions.

Definition 3.4 (Contract, engineer to agent)**.** A *contract* is 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 is a contract only if a program can decide whether it was satisfied without asking the model that produced the work.

The last clause carries the definition. A checklist the agent grades itself against is a prompt.

Definition 3.5 (Sandbox)**.** A *sandbox* is an execution environment in which an agent's actions cannot reach state outside a declared boundary, where the boundary is enforced by a mechanism the agent cannot instruct. A permission prompt that the agent's own output can influence is not a sandbox.

The exclusion is empirical rather than theoretical, and Section 4.5 gives the 2025 supply-chain compromise that motivates it [53]. A boundary a caller can be told to step over is a convention.

Definition 3.6 (Harness)**.** The *harness* is the program that runs the cycle around a model: it assembles the context, exposes the tools, executes the model's chosen invocations, enforces the sandbox and the permission policy, and decides when to stop. The model is not the harness; two systems with the same model and different harnesses are different systems.

This is a stipulation. The term is recent and observer-supplied rather than vendor-coined. Section 4 relates this practitioner definition to the formal treatment in the companion formal working series (2026), in preparation, which decomposes the same object differently.

Definition 3.7 (Orchestrator)**.** An *orchestrator* is the component that decides which agent, or which unit of work, runs next, and when, given more than one candidate. An orchestrator is present only if that decision is made outside any single agent's own cycle.

The definition is a stack position, and this Part develops it no further. What an orchestrator does across several agents is Part IV's subject.

Definition 3.8 (Verification)**.** *Verification* yields evidence that a produced artifact satisfies a stated criterion. Under this series' definition, independent verification is produced by a procedure that evaluates the artifact without relying on the producing agent's self-assessment. Self-grading may reveal defects, but it is not an independent evidence class and cannot be the sole gate.

Definition 3.9 (Evidence class)**.** An *evidence class* is a named category of verification evidence with a stated pass criterion and a stated artifact that records the result. Two pieces of evidence belong to different classes when they can fail independently.

The independence clause is what makes the notion useful. A test suite and a schema check on the same artifact are different classes, because a program can pass its tests and still emit a response its consumer cannot parse. A test suite and a second run of the same test suite are not.

3.2 Composition and boundaries

The seven definitions below are stipulations by this series. We found no source that defines them operationally for agent systems. The functional-programming tradition they draw on is about programs rather than agents, so applying it here is an analogy, and this paper treats it as one. Section 5.5 gives that tradition's empirical record, which is weaker than its reputation.

Definition 3.10 (Composition)**.** A *composition* is a larger behaviour built by arranging smaller steps so that one step's output becomes another step's input across a declared boundary. An arrangement is a composition only if the boundary between two steps can be named and inspected independently of either step's internals.

Definition 3.11 (Typed boundary)**.** A *typed boundary* is an explicit statement of what values may cross between two steps, expressed so that a program can decide, before the receiving step runs, whether a given value satisfies it. A boundary is typed only if a violation is detectable without executing the receiving step.

Definition 3.12 (Pure step)**.** A *pure step* is a step whose output is determined entirely by its declared inputs and which can be re-run on the same inputs without changing anything outside itself. It is testable by running it twice and comparing both the output and the observable world.

Definition 3.13 (Effectful step)**.** An *effectful step* is a step that changes or consults something outside its declared value inputs: a model or tool call, a file write, a network request, or a deployment. It may occur anywhere in a composition, but it must sit behind an explicit boundary, and every invocation must be recorded with enough detail to identify its inputs and result.

Definition 3.14 (Immutable artifact)**.** An *immutable artifact* is a value produced by a step that is never edited in place. A correction produces a new artifact that supersedes the old one, and both remain retrievable. It is testable by asking whether the previous version can still be read after a change.

Definition 3.15 (Composition operator)**.** A *composition operator* is one of five named ways to combine steps, each defined by what it takes and what it guarantees. *Sequence* takes an ordered list of steps and guarantees each step sees the previous step's output. *Parallel fan-out and fan-in* takes steps with disjoint declared and actual effect footprints plus a combining rule that is commutative or applied in a canonical order. Under those conditions, the result does not depend on completion order. *Branch* takes a decision value and a set of cases, and guarantees exactly one case runs. *Bounded loop* takes a body, a continue condition, and a maximum count, and guarantees termination with a distinct outcome on exhausting the bound. *Supervise* takes a child and a restart policy, and guarantees a failed child does not silently become a success.

Definition 3.16 (Functional thinking)**.** *Functional thinking* is the discipline of designing an agent system as a composition of pure steps and explicitly bounded, recorded effectful steps joined by typed boundaries, producing immutable artifacts, with verification placed at the boundaries rather than inside the steps. A design exhibits functional thinking to the degree that a reader can name every boundary and say what is checked there.

The last clause gives a review question rather than a metric, for the reason Section 5.5 gives.

Remark 3.17 (On the word cycle in Definition 3.6)*.* Definition 3.6 says a harness runs a cycle around a model. That cycle belongs to Part III: its Definition 2.2 gives the loop, and its Definitions 2.1 and 2.4 give the gate and the iteration bound. This Part names the harness as the component that runs the cycle and says nothing about the cycle's shape. The two accounts are compatible by construction, because Part III builds its loop out of the operators of Definition 3.15 below.

4 The primitives in running systems

Each of the nine primitives can be located in code. Every placement below names a repository, a commit, a path, and a line range, and states where the component sits rather than how it behaves at run time.

Four code bases are used: agent-os (Elixir, commit `61cb3994da`), AgentHero (Rust, commit `1c3ad24011`), CatDB (Rust, commit `7cc9341a16`), and ContextFS (Python, commit `a93035de04`). All four share one author with this series. They are case studies, not independent confirmation, and Section 10 states what that costs.

4.1 The model layer

The model layer is the part a practitioner cannot change from inside the system. What matters about it for design is Definition 3.2: the context window is bounded, and its useful bound is smaller than its stated one.

Three independent efforts using different methods agree on this. A retrieval task built so the answer cannot be found by literal string matching found that 11 of 13 models advertising at least 128K context fell below 50% of their own short-context baseline at 32K, with one model falling from 99.3% to 69.7% [35]. A benchmark covering 17 long-context models across 13 task types reports large drops in almost every model as context length grows, and criticizes single-needle retrieval as too superficial to reveal them [25]. Accuracy also follows a U-shaped curve in the position of the relevant material, highest at the start and the end [31].

A fourth source points the same way with a conflict of interest that should be visible. A vendor technical report covering 18 frontier models finds that even a single distractor reduces performance against baseline [16]. Its author is a vector-database vendor, and therefore has a commercial interest in the claim that curation matters more than window size.

A second bound sits at the same layer, on task length rather than input length. Measuring the human-equivalent duration of tasks an agent completes at a 50% success rate, one study reports the horizon doubling every 212 days, with a 95% bootstrapped confidence interval of 171 to 249 days [29]. The measure is defined at a 50% success rate, so the horizon names the task length at which the agent fails half the time. Both halves of that result matter. The trend is why the stack above the model is worth building. The failure rate at the horizon is why the stack has to include a place for a human.

Placement. AgentHero holds the model layer behind a provider abstraction. The workspace adapter is `crates/llm-adapter`.

The runner is `crates/agent-runtime/src/runner.rs`. It declares an `AgentRunner` trait with two implementations: a direct provider-API runner and a local subprocess runner. The specification's provider field selects between them (AgentHero at `1c3ad24011`, lines 1 to 30). Putting the model behind a trait makes the last sentence of Definition 3.6 operational. The same harness runs against different models, and the difference is visible in one file.

4.2 The harness

Definition 3.6 draws a line practitioners routinely blur. Two coding assistants running the same model behave differently, and the difference lives in this layer: which files were read into the context, which tools were declared, which invocations required approval, and when the program decided to stop.

The design argument for taking the layer seriously predates the vocabulary. Work on an agent-computer interface showed that the action and observation layer offered to a model is a designed artifact rather than an accident. A purpose-built file viewer and a linting-aware editor changed task performance without changing the model [55]. That paper also describes agents as a category of end user with their own needs and abilities, which is the design premise of everything in this section.

The term is younger and vaguer than the idea. We found no single origin for "harness" in the sources we reviewed (cutoff 2026-09-01). It appears in evaluations guidance, in preprint titles, and in trade coverage, as observer vocabulary for what vendors build rather than as a vendor coinage.

Relation to the formal treatment. The companion formal working series (2026), in preparation, treats the same object as a triple of wiring, certificates, and a deployment map. It asks which of a harness's guarantees survive recompilation into a different framework.

Definition 3.6 is coarser and answers a different question. It lists the five jobs a practitioner has to find an owner for. The two accounts line up where it matters: the formal wiring component corresponds to context assembly and tool exposure, the certificate component to the permission and verification jobs, and the deployment map to the sandbox and the stopping rule.

The external preprint this Part draws on reaches the same separation from the other side, arguing that the guarantees practitioners want are attributes of harness structure rather than of the model, and adopting the four-pillar decomposition of Zhou et al., memory, skills, protocols, and harness engineering, which it maps onto its own architecture triple [11]. This Part disagrees with it on placement, not on content. What that preprint calls memory this Part puts in a separate state layer and hands to Part V.

Placement. agent-os runs its harness as a shell program inside each execution environment. `sandbox/scripts/agent-runtime.sh` assembles context, calls a local proxy for inference, executes

the chosen commands, and applies artifact checks by file type. Its iteration bound is declared at the top of the file (`MAX_ITERATIONS=10`, line 16).

AgentHero puts the same jobs behind the `AgentRunner` trait described above. Its documentation comment records a design rule: the supervisor owns the side effects, meaning cache, verifier ladder, database persistence, rendering, and publication, while agents and runners reason and execute without touching the database or opening pull requests. That comment is a first-party statement of the rule Section 5 states in general terms.

The controls a practitioner actually touches. These are documented, not inferred. One vendor's assistant exposes a plan mode that reads and proposes without editing until approved, worktrees for parallel sessions, subagents defined as Markdown files with front matter that can restrict tools and route to cheaper models, and user-defined handlers on any of 33 lifecycle events, configured in JSON settings files at four scopes [8]. A cloud coding agent triggered from issues and pull request mentions declares a hard 59-minute maximum execution time per session [22]. Another vendor's cloud agents run in isolated virtual machines, separate from the local machine [17].

Every one of those settings belongs to the harness rather than the model. A team that has not decided where each is set has not decided what its system does.

4.3 Skills

Definition 3.3 separates a skill from a prompt by two tests: it has a name, and it survives the session. The precedent is older than the current tooling. An open-ended embodied agent built a library of executable behaviours it had written itself, retrieved them by description, and composed them into longer behaviours [51]. That is the earliest clear case of skills as persistent, addressable, reusable units rather than instructions retyped each time.

Today the primitive appears as project instruction files and rules directories. One vendor documents a four-scope hierarchy, from managed policy through user and project to local, loaded from the working directory upward at launch, with files in subdirectories loaded on demand when the agent reads files there. It advises targeting under 200 lines per file, on the stated ground that longer files consume more context and reduce adherence [9]. The same page names the constraint the whole layer exists to answer: each session begins with a fresh context window.

Another vendor documents glob-scoped rule files with front matter, replacing a legacy single-file format, and gives the same reason plainly. Models do not retain memory between completions, so rules supply persistent reusable context at the prompt level [18].

Placement, and an absence. None of the four code bases implements a skill in the sense of Definition 3.3. agent-os has a tool registry and a contract specification but no named, on-demand instruction set. AgentHero carries per-role prompts inside its manifests rather than as separately addressable artifacts. The primitive is real and widely deployed in commercial harnesses; these case studies simply do not instantiate it.

4.4 Tools

Definition 3.1 has two clauses, and the second is the one teams violate. They write a paragraph describing a capability, put it in the system prompt, and are surprised when the model does not use it. If the harness cannot execute it under a name the model can emit, it is not a tool.

The surface has a dateable origin: a 2023 API update let a model return a structured request to call a named function with typed arguments [37]. Interoperability came later. The Model Context

Protocol was announced in November 2024 [7], and its specification defines a tool as identified by a name and carrying metadata describing its schema, with input and output JSON Schemas [34].

That specification is a moving target. It carries dated revision identifiers, and five existed at the time of writing: 2024-11-05, 2025-03-26, 2025-06-18, 2025-11-25, and 2026-07-28, the last being current [34]. Five revisions in twenty months means a tool declaration written against one of them is a contract with a counterparty that keeps changing. Designing tools for a model to consume also has its own guidance, distinct from designing an API for a human caller [6].

Placement. agent-os holds tools in a three-tier registry implemented as a process. The tiers are built-in tools, which are statically defined and trusted; sandbox-tier tools, which require isolated execution; and tools discovered at run time from protocol servers, namespaced to prevent collisions. Each entry carries a name, a tier, an input schema, an optional output schema, an optional validator, and an executable function (agent-os at `61cb3994da`, `src/tool_interface/lib/tool_interface/registry.ex`, lines 1 to 50).

The registry can be frozen. Once frozen, registration fails with an explicit error, so the tool set cannot change during an execution (same file, lines 70 to 90 and 136 to 137). Access runs through scoped, time-bounded, rate-limited signed capability tokens (`capability.ex`, lines 1 to 14).

AgentHero declares tools in its manifest instead. Each declaration carries an identifier, one of twelve executor kinds, an optional command, an optional timeout, optional input and output schemas, and an optional policy envelope (AgentHero at `1c3ad24011`, `crates/dag-runtime/src/lib.rs`, lines 492 to 507 and 511 to 523).

Implementation limits at this placement. Six of the twelve tools agent-os declares return fixed empty results, each carrying an explicit stub marker in the returned map: web search, web scraping, PDF parsing, spreadsheet reading, repository cloning, and file operations (`registry.ex`, lines 212, 245, 263, 299, 322, and 376). The repository's own planning document says so, listing "12 tools return empty results" as one of four gaps and assessing the harness at 42% complete against a 78% target (`plans/harness-fix-plan.md`, lines 1 to 10). The registry establishes the shape of the interface; it does not establish operational completeness.

A second gap in the same file illustrates Definition 3.11. The registry declares a tool tier type with three values, but one value is spelled differently in the type declaration than in the code that assigns it. The declared distinction is enforced nowhere. A type that nothing checks is documentation.

4.5 Sandbox and isolation

Definition 3.5 rules out the most common arrangement. A confirmation prompt is a boundary only against accident. It stops neither an agent whose input told it to pass a flag that removes the prompt, nor a supply-chain attacker who reaches the same command line.

The 2025 compromise of a widely used build tool is the concrete case. A malicious postinstall script invoked installed developer AI command-line tools with permission-bypassing flags and used them for reconnaissance and credential exfiltration. The reported blast radius included over a thousand valid repository tokens and, in a later wave, more than 5,500 private repositories made public [53]. A sandbox boundary has to be one the agent cannot address.

Placement, in two senses. agent-os uses the word for two different things, and both are legitimate. Coarsely, each pipeline stage runs in a microVM with its own kernel, mounting a read-only context directory and a read-write output directory, defaulting to 512 MB of memory and one CPU. The module comment states that all agents must run in microVMs, with no fallback to

in-process execution (agent-os at `61cb3994da`, `src/agent_os/lib/agent_os/micro_vm.ex`, lines 1 to 21). Finely, in-process tool calls run in spawned monitored processes with isolated heaps and a timeout, defaulting to 600,000 milliseconds with an absolute maximum of 1,800,000 (`src/tool_interface/lib/tool_interface/sandbox.ex`, lines 1 to 37).

Those are different guarantees. The first bounds what a stage can reach; the second bounds what a single call can consume. A design should say which one it is claiming.

AgentHero declares an isolation policy with exactly two variants, host or container, and the default is host (AgentHero at `1c3ad24011`, `crates/agent-runtime/src/types.rs`, lines 105 to 116). Its per-tool policy defaults run the same way. Network access defaults to allowed, subprocess creation defaults to allowed, and credential inheritance from the ambient host environment defaults to on (`crates/dag-runtime/src/lib.rs`, lines 588 to 592, 611 to 615, and 625 to 629).

The executor is not defenceless. A preflight check rejects unsupported host-isolation configurations before a tool runs, failing a required node and degrading an optional one (`crates/dag-executor/src/lib.rs`, lines 4245 to 4270). But the platform's own completion audit states the position without softening it: tool isolation is implemented for local trusted-operator workloads, and hardened sandboxing for hostile code is recorded as future work (`docs/agenthero-platform-completion-audit.md`, lines 41 and 53).

Table 1 collects these. Claimed isolation and default isolation are separate columns, and the default column is the one that describes most running systems.

System	Mechanism present	Default policy	First-party stated limit
agent-os, coarse	A microVM per pipeline stage, with its own kernel, mounted context and output directories, 512 MB of memory, and one CPU	All agents are required to run in a microVM, with no in-process fallback	The harness is assessed at 42% complete in the repository's own plan
agent-os, fine	An isolated, monitored process per tool call, with its own heap, killed on timeout	A 600,000 ms default timeout and a 1,800,000 ms maximum	The bound is on what a call consumes, not on what it reaches
AgentHero	Two variants, host or container	Host, with network access, subprocess creation, and credential inheritance all defaulting to permitted	The completion audit records hostile-code sandboxing as future work
Vendor cloud agents	Isolated cloud virtual machines, separate from the local machine [17]	Not stated on the documentation page	The source is documentation, not measurement

Table 1: Sandbox mechanisms, default policies, and first-party stated limits across the case studies. Mechanism and default are separate columns because they routinely differ, and the default is what a system does when nobody has decided.

4.6 Contracts

Definition 3.4 is the primitive that makes delegation reviewable. Its decisive clause is that a program, not the producing model, decides satisfaction.

The idea has wider currency. One vendor's specification-driven toolkit describes the specification as a contract for how code should behave and as the source of truth, and its repository states the discipline as defining what to build before building it [23]. Another vendor's agent framework separates handoffs, which delegate to another agent, from guardrails, which validate agent inputs and outputs and raise a tripwire [39]. Guardrails in that sense are contracts with a narrower scope.

Placement. agent-os expresses a contract as data rather than code. A named specification carries a list of stages, each with instructions, declared outputs, and an optional input dependency on an earlier stage, plus a list of required artifacts, a list of verification rules, a retry bound, and a resource block (agent-os at `61cb3994da`, `src/agent_os/lib/agent_os/contracts/contract_spec.ex`, lines 1 to 79). The module comment makes the separation of what from how explicit: a specification is built from maps or YAML so that a user can define a contract without writing code. The companion interpreter checks artifacts against the rules and returns either success or a retry with a reason (`verify.ex`, lines 16 to 29).

AgentHero puts the contract in the type system at the node boundary. Tools may declare input and output schemas, and the executor validates a node's inputs against the tool's input schema before the node runs and its outputs against the output schema after (AgentHero at `1c3ad24011`, `crates/dag-executor/src/lib.rs`, lines 4309 to 4334). At the application level the same discipline appears as 37 closed JSON schema files, one per structured output type, under a single application's schema directory.

Implementation limits at this placement. agent-os's verification interpreter supports exactly three rule types: a file exists, a file is at least a stated number of bytes, and a key in the artifact map is present and non-empty (`contract_spec.ex`, lines 36 to 39; `verify.ex`, lines 31 to 68). All three check shape. None checks behaviour.

The dispatcher is worse than incomplete. Its final clause returns success for any rule it does not recognize (`verify.ex`, line 70). A contract author who misspells a rule name gets a passing verification, silently. It illustrates the point Section 7 makes in general: a passing check tells you what the check checked, and nothing else.

4.7 Orchestration and governance

Definition 3.7 is a stack position. What an orchestrator does when several agents contend for work, how it recovers from a member's failure, and what patterns it can express are Part IV's subject, and this Part stops at the placement.

agent-os holds that position in a 416-line pipeline module. It executes contract stages in order, each in its own microVM, preparing context before a stage and ingesting the stage's output afterwards (agent-os at `61cb3994da`, `src/agent_os/lib/agent_os/pipeline.ex`, lines 1 to 14).

AgentHero holds it in a dedicated workspace crate, `crates/orchestrator`, one of eight in the workspace. Its architecture documentation states a compile-time rule: the orchestrator must not depend on application crates. That rule is a typed boundary in the sense of Definition 3.11, enforced by a build system rather than by a schema.

Governance answers who was allowed to do this and what record exists. agent-os writes a structured audit record for every command, inference call, tool use, and stage transition into a table keyed by pipeline identifier (`src/agent_os/lib/agent_os/audit.ex`, lines 1 to 14).

CatDB places authorization before query planning rather than after. Its policy crate exposes a function documented as authorizing and filtering a query before planning, and that function rejects

a principal bound to a different deployment before any source is consulted (CatDB at `7cc9341a16`, `catdb/crates/catdb-policy/src/lib.rs`, lines 972 to 980).

CatDB also gates what the agent-facing surface advertises. A single list of tool names is read both by the publication filter and by the call-time refusal path, so what a session advertises and what it is willing to answer cannot drift apart; a comment in the source states that reason (`catdb/crates/catdb-server/src/mcp.rs`, lines 75 to 90 and 272 to 285). One list feeding both is a small decision with a large consequence. It keeps "no such tool" and "not authorized on this session" distinguishable answers.

4.8 State

Two things live in the state layer and both belong to Part V.

Persistent memory is stored state that outlives a session and is retrieved into a later session by a query rather than by a person pasting it. ContextFS is the case study for what Part V defines as persistent memory in its Definition 2.2. It exposes save and search entry points over a typed memory store (ContextFS at `a93035de04`, `src/contextfs/core.py`, lines 673 and 968), with a schema registry that validates types without being part of the stored state (`src/contextfs/types/registry.py`, line 23). Part V develops what selective retrieval requires and what it costs.

One property of that store belongs here rather than there, because Section 5 uses it. The store offers both an in-place update and an evolve operation. Only the second preserves history, creating a new record with a relationship back to the original and leaving the original unchanged (`core.py`, lines 1391 and 1515 to 1554). That is Definition 3.14 in one file.

The governed data layer, Part V's Definition 2.3, is the component between agents and systems of record that authorizes before planning, attaches source and version to returned values, and returns a typed refusal rather than a partial answer when it cannot answer soundly. CatDB is the case study and Part V develops it, along with per-field provenance and the refusal property at its Definitions 2.4 and 2.5. Its stack position is the one shown in Section 4.7: below the tool surface, above the sources.

5 Functional thinking for agent systems

The parts list of Section 4 is also a design vocabulary. That claim is what the rest of the series builds on.

5.1 Steps and boundaries

With Definitions 3.10 through 3.16 in hand, each of the nine primitives resolves into a step or a boundary.

A tool is an effectful step with a typed boundary on the way in. It is what the interoperability specification says a tool is, a name plus an input schema and an output schema [34], and it is what the case studies implement, a declared executor plus optional input and output schemas plus a policy envelope.

The other eight follow the same way. A contract is a typed boundary between a human and an agent, checkable before the agent runs and after it stops. A sandbox makes an effectful step's reach nameable, which is what lets effects sit at the edges rather than merely be hoped to stay there. A harness is the program that holds the composition together and executes its effectful steps. An orchestrator is a composition operator with the model's own choice removed from the selection. Verification happens at a boundary. An evidence class names what a particular boundary

check rules out. A context window is the budget every step's inputs are drawn against. A skill is a reusable step body.

That reading changes two things in practice. Design questions become uniform: for every arrow in a system, what crosses it, who checks it, and what happens on a violation. And a system's weakest point becomes findable by inspection, because it is the boundary nobody can name.

Sources of the vocabulary. Four of the ideas here are borrowed.

Composition as the glue, the claim that a language's contribution is what it lets you join rather than what it lets you write, is Hughes's [26]. It predates agents entirely, so applying it here is an analogy rather than evidence about agent systems. Placing effects at a typed boundary instead of scattering them through a program is Wadler's account of monads, and Definition 3.13 takes its shape from there [50]. Minimizing state, on the ground that each bit of state doubles the number of states a reader must consider, comes from a preprint that was never formally published; it is cited here as the clearest statement of that point, not as a peer-reviewed result [36].

The modern instance closest to Definition 3.11 is a framework that treats a declared input and output signature as the unit of composition and compiles over it, describing its modules as parameterized so they can learn how to apply compositions of prompting and reasoning techniques [28]. A graph framework makes the same move with nodes as functions over state and edges as conditional routing [30].

None of this is evidence that the discipline improves agent systems. It is where the words came from.

5.2 The five operators

Table 2 sets out the five operators of Definition 3.15: what each takes, what each guarantees, the prior art behind it, and where it appears in the case studies.

Five operators with stated guarantees are easier to review than an arbitrary control-flow graph. The guarantees are also what a reviewer checks, rather than what an implementation promises. An implementation offering a bounded loop with no distinct outcome on exhausting the bound has not implemented the operator. It has implemented a loop that lies.

The vendor pattern vocabulary maps onto these operators cleanly, which suggests the operators sit at the right granularity rather than being an invention. Prompt chaining decomposes a task into a sequence where each call processes the previous output, which is sequence. Parallelization runs simultaneous calls aggregated programmatically, which is fan-out and fan-in. Routing classifies an input and directs it to a specialized follow-up, which is branch. Evaluator-optimizer has one call generate while another evaluates in a loop, which is a bounded loop whose body contains a distinct reviewer. Orchestrator-workers has a central model break down tasks, delegate, and synthesize, which is fan-out with a synthesizing fan-in [3].

The case that list holds apart, fully autonomous agents, is not an operator. It is what happens when selection is delegated to the model, and it is the case Definition 3.7 distinguishes.

Each operator is older than the current tooling. Workflow engines have used a DAG as the orchestration primitive for over a decade [2]. Durable execution engines recover by replaying a recorded event history treated as the source of truth for everything that happened [47]. Process supervision with declared restart strategies is older still [20]. That lineage is a reason to trust the guarantees as stated; it is not a reason to expect them to transfer to agents, and Section 5.5 states the limit.

AgentHero's node taxonomy supplies four of the five directly. Its node kind enumeration includes `Loop`, `Branch`, and `Map` alongside `Verify`, `Gate`, `Tool`, and `Approval` (AgentHero at `1c3ad24011`,

Operator	Takes	Guarantees	Prior art	Case study
Sequence	An ordered list of steps	Each step sees the previous step's output	Prompt chaining [3]	agent-os contract stages linked by `input_from`
Parallel fan-out and fan-in	Steps with disjoint effect footprints plus a commutative or canonically ordered combining rule	Under those conditions, the result does not depend on completion order	Parallel calls [3] and DAG layers [2]	The AgentHero `Map` node, bounded by `max_items`
Branch	A decision value and a set of cases	Exactly one case runs	Routing [3]	The AgentHero `Branch` node, with a decision key, cases, and a default
Bounded loop	A body, a continue condition, and a maximum count	Termination, with exhausting the bound recorded as a distinct outcome	Evaluator and optimizer [3]	The AgentHero `Loop` node, bounded by `max_rounds`
Supervise	A child and a restart policy	A failed child does not silently become a success	Supervisor behaviour and its restart strategies [20]	agent-os supervision trees

Table 2: The five composition operators of Definition 3.15, with case-study entries read from source at the pinned commits rather than observed at run time. Control flow within and between these operators is the subject of Parts III and IV.

`crates/dag-runtime/src/lib.rs`, lines 109 to 126). The bounded loop carries a required maximum round count and a continue key (lines 189 to 196). The branch carries a decision key, a map of cases, and a default (lines 200 to 208). The map carries an items key, a required maximum item count, and an optional concurrency limit (lines 233 to 244). Supervise comes from elsewhere: agent-os inherits it from the language runtime's supervision behaviour [20].

5.3 A worked example

The vocabulary is worth nothing without an example small enough to check. Take a documentation task: given a repository path and a topic, produce a page of prose that is accurate about the code. It is small, it is real, and it fails quietly, because a plausible page about code that does not exist looks exactly like a correct one.

Listing 1 composes it. Effectful steps carry a trailing exclamation point. The drafting call is effectful because it invokes a model provider and need not return the same value twice. The deterministic citation check is pure. The types name what crosses each boundary.

```
# Types crossing the boundaries
type Request = { repo : Path, topic : Text }
type Excerpt = { path : Path, lines : (Int, Int), text : Text }
type Draft = { body : Text, cites : [Excerpt] }
type Report = { ok : Bool, unmatched : [Excerpt] }

# Step 1. Effectful: reads the filesystem.
```

```
collect! : Request -> [Excerpt]

# Step 2. Effectful: invokes a model provider; record its result.
draft! : (Request, [Excerpt]) -> Draft

# Step 3. Pure: every cited excerpt is re-checked against the
# excerpts collected in step 1. Consults the artifact, never
# the step that produced it.
check : (Draft, [Excerpt]) -> Report

# Step 4. Effectful: writes the page. Runs only if check passed.
publish! : Draft -> Path

# The composition. Sequence, then a bounded loop around
# draft and check, then a branch on the report.
pipeline = sequence
  [ collect! # recorded read
  , boundedLoop 3 (draft! >> check) (not . ok) # recorded call, pure check
  , branch ok publish! reportFailure # recorded write
  ]
```

Listing 1: A documentation task composed from four steps, with a trailing exclamation point marking each effectful step. The type signatures, not the bodies, are the boundaries.

Four properties of this composition are each a decision a designer can get wrong.

The three effectful steps have explicit boundaries and recorded results. A reviewer investigating a bad page can re-run `check` on the recorded draft and excerpts and get the same result without reconstructing the repository state. Repeating `draft!` is a new provider invocation and may produce a different draft; replay uses the recorded draft instead.

The check step takes the excerpts as an argument rather than re-reading the repository. This is Definition 3.8 applied literally. The check does not ask the drafting step whether it was careful, and it does not consult the model that wrote the draft. It compares two values, and a program can decide the result.

The loop is bounded at three, and the branch has a failure arm. Exhausting the bound produces a report with `ok = false` and a list of unmatched citations, which is a distinct outcome from a passing report. No path reports running out of attempts as success. That is the bounded-loop guarantee of Table 2, and it is the one most often missing in systems this size.

Each step's output is a new value. The second draft does not overwrite the first. If the third attempt is worse than the second, both remain readable. That is Definition 3.14.

Figure 1 shows the same composition as a diagram, with the effect boundary drawn explicitly.

5.4 Design rules

Put effects behind explicit boundaries, and record every one. Teams skip the recording half. AgentHero's observability guidance requires a fixed set of trace fields on every durable event, and its runner documentation states the same separation as a design rule: the supervisor owns side effects, and agents do not touch the database or open pull requests.

Recording has a limit, and the companion formal working series (2026), in preparation, states it in one line. Fencing protects the record, not the world, because the model has no vocabulary for an effect that was performed but never recorded. A step that writes to an external system and crashes before its write is logged has produced a state the composition cannot see. Recording effects is how you find out afterwards. It is not how you prevent it.

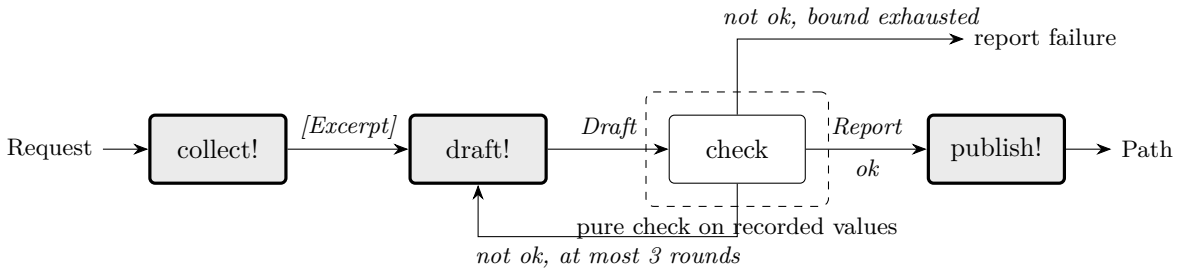

Figure 1: The composition of Listing 1, with shaded boxes for recorded effectful steps, a dashed region for the deterministic citation check, and arrow labels for the types that cross each boundary. Repeating the model call may produce a different draft; replay instead consumes its recorded output. The exhausted outcome is separate from the passing one.

Never edit an artifact in place. ContextFS provides both operations, and the difference between them is the rule. Its update path modifies a record. Its evolve path creates a new record carrying a relationship back to the original and leaves the original alone, with the docstring stating that the original memory remains unchanged (ContextFS at `a93035de04`, `src/contextfs/core.py`, lines 1391 and 1515 to 1554). A system offering both will accumulate whichever is more convenient, so the choice has to be made at design time and enforced at the boundary.

Check at the boundary, not inside the step. The mechanism is the declared schema: input and output schemas on a tool [34], closed per-output-type schemas on a node, structured output constrained by a supplied schema at the model boundary [38]. A check inside a step is a check the step can be wrong about. A check at a boundary is a check a different program performs on a value.

5.5 Scope of the argument

Functional thinking is argued here from failure modes and case inspection. It is not a measured improvement.

The tradition's own empirical record does not survive reproduction. An observational study of language and code quality covered 728 projects, 63 million lines, and 17 languages. It reported a modest defect-rate association favouring functional languages and stronger typing [44]. A peer-reviewed reproduction on the same data found the practical effect size exceedingly small [14]. The reproduction is later and more careful, and it substantially deflates the original.

Both studies concern programs written by people rather than agent systems, so their bearing here is indirect in either direction. No Part of this series presents functional thinking as an empirically established improvement.

The most direct test of typed boundaries is disputed. One study reports a significant decline in reasoning ability under format restrictions, across several open and closed models on reasoning benchmarks, comparing free-form generation against JSON or XML constrained output [46]. A response argues the comparison used non-comparable prompts between conditions, and reports 77% for structured generation against 73% unstructured on the matched JSON comparison it re-ran [19]. That response is published by a party selling structured-generation tooling, and its page carries no date.

A third position comes from the authors of a dedicated schema-adherence benchmark. Most constrained-decoding methods guarantee compliance given a schema, and the effectiveness of those methods in practice is poorly understood [21].

This Part takes no side. The dispute concerns output format at the model boundary, while the argument in Section 5.1 concerns where checks are placed. A contract can be checked after generation without constraining generation.

The strongest evidence for boundaries comes from failures, and is indirect. A taxonomy built from 150 hand-annotated multi-agent traces, with inter-annotator agreement of 0.88, identifies 14 failure modes in three categories, one of them system design and specification issues [15]. Two modes are specification defects by name: failure to adhere to the specified constraints or requirements of a given task, and failure to adhere to the defined responsibilities and constraints of an assigned role. Three more are interface defects in substance: being unaware of termination conditions, failing to ask for clarification, and withholding information. Typed boundaries address that class by construction, because a constraint stated in a schema is checked rather than hoped for.

This is the best support available, and it remains indirect. The taxonomy classifies traces; it does not measure what adding boundaries would have prevented. Its authors also state that the traces come from open-source frameworks and may not generalize to production systems.

Two critiques of the framing, and one absence. The tradition this vocabulary borrows from has been criticized from inside. One such argument holds that presenting an abstraction by analogy, before the learner has done the concrete work, is counterproductive. That is a pedagogical objection, not a claim that the formalism lacks value, and the distinction matters when citing it [57]. A peer-reviewed survey of the incompatible metaphors used to explain one such abstraction argues that none of them is adequate [42].

We found no critique specifically targeting category-theoretic framings of software engineering in the sources we reviewed (cutoff 2026-09-01). An absence is not an endorsement.

A case where composition did not help. One result cuts against the operator vocabulary directly. A three-phase pipeline with no agentic selection at all resolved 32.00% of 300 SWE-bench Lite instances at $0.70 per instance, beating contemporary agent scaffolds at lower cost [54]. A meta-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 [27].

Neither result argues against typed boundaries. Both argue against assuming that more composition is better composition. Section 9 carries the point further.

6 A six-layer reference architecture

Figure 2 draws the six layers as a stack.

The figure is a proposal, and nobody has ratified six layers; none of the three vocabularies in Section 2 supplies a layering to adopt instead.

The layers are useful even if the count is wrong, because the layer boundaries are where the design decisions sit. Whether a permission check lives in the harness or in the tool policy is a real decision with different consequences, and a stack makes that question askable.

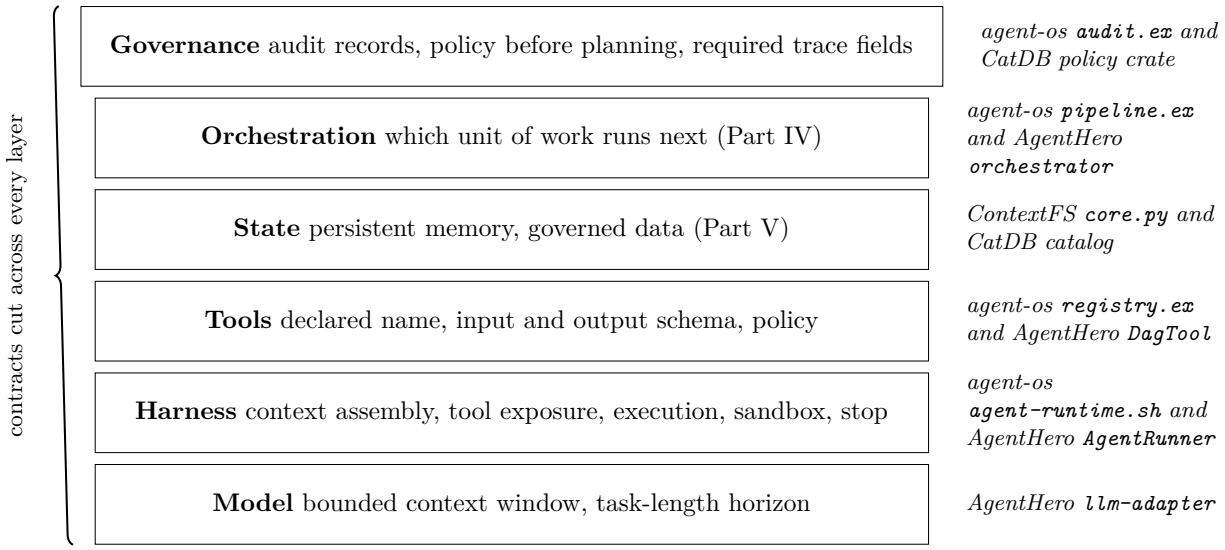

Figure 2: A six-layer reference architecture, proposed rather than reported as consensus, in which each layer names a job that needs an owner and the right column places the four case studies. Contracts are drawn as cutting across the stack because a contract is a boundary between layers rather than a resident of one.

Contracts occupy no layer. A contract is a boundary, and a boundary between two layers belongs to neither. Figure 2 therefore draws contracts as a brace rather than a box, which is the observation of Section 5.1 in another form.

Table 3 gives the full placement matrix, with paths at pinned commits.

Read the table across rather than down. No single case study fills the stack, and the empty cells are as informative as the full ones. CatDB implements a governed source of answers and has no harness. ContextFS implements state and has no orchestration.

A team assembling a system from components assembles it across this table. The boundaries between the components they pick are the ones nobody will check unless somebody decides to.

7 Verification and evidence classes

Definition 3.8 excludes a common arrangement, and the exclusion rests on measurement rather than assertion.

7.1 Self-review

A model asked to judge outputs recognizes its own. In a controlled study, one frontier model distinguished its own outputs from other models' and from human text with 73.5% accuracy, and the strength of that self-recognition correlated linearly with self-preference bias after fine-tuning [40].

Two further results point the same way. Sycophancy is a measured behaviour of assistants trained from human feedback: responses shift toward a stated user position across several task types [45]. And the paper that established the model-as-judge method reports over 80% agreement with human preference while flagging position bias, verbosity bias, and self-enhancement bias as open limitations [58].

Together they give a consistent picture. A model is a usable judge of work it did not produce

Layer	**agent-os** 61cb3994da	**AgentHero** 1c3ad24011	**CatDB** 7cc9341a16	**ContextFS** a93035de04
Model	None	crates/llm-adapter and agent-runtime/src/runner.rs:1-30	None	None
Harness	sandbox/scripts/agent-runtime.sh	crates/agent-runtime/src/runner.rs	None	None
Tools	registry.ex:1-90 and capability.ex:1-14	dag-runtime/src/lib.rs:492-523	catdb-server/src/mcp.rs:272-285	mcp/fastmcp_server.py
Sandbox	micro_vm.ex:1-21 and sandbox.ex:1-37	types.rs:105-116 and dag-executor/src/lib.rs:4245-4270	None	None
State	The memory_layer storage	Application runtime tables	The catalog crate	core.py:673,968 and types/registry.py:23
Orchestration	pipeline.ex:1-14	crates/orchestrator	None	None
Governance	audit.ex:1-14	Required trace fields	catdb-policy/src/lib.rs:972-980	None
Contract	contract_spec.ex:1-79 and verify.ex:16-29	lib.rs:4309-4334 and 37 output-type schemas	Typed API data transfer objects	schemas.py

Table 3: Stack placement across the four case studies, read from source at the pinned commits, with file names given without their directory prefix and the full path for each in the subsection of Section 4 that discusses the row. Entries state where a component sits rather than that it works, and every row is qualified by the stubs, permissive defaults, and unrecognized-rule pass recorded in Section 4.

and a biased judge of work it did. That is why Definition 3.8 bars the procedure from consulting the producing agent. Model judgment is not worthless; the constraint is about which chair the model sits in.

7.2 The evidence ladder

The independence test in Definition 3.9 yields a ranking. Table 4 orders four classes by what each rules out, and names a failure mode each one misses.

Class	Pass criterion	Recorded artifact	Not ruled out
Machine-checked proof	A proof assistant accepts the term against the stated theorem [56]	The proof term and the checker's verdict	That the theorem states what you meant
Executed tests	A named suite passes on the produced artifact	The suite output and exit status	That the tests were written to pass, or edited to pass [15]
Schema validation	The value satisfies a declared schema before the consumer runs [34]	The validator's verdict and the offending path	Anything about content, since a well-shaped wrong answer passes
Model judgment	A model that did not produce the work returns a verdict	The verdict and its stated reasons	Position, verbosity, and self-preference bias [58, 40]

Table 4: Four evidence classes ordered by what each rules out. They are different classes in the sense of Definition 3.9 because they can fail independently: an artifact can pass its schema and fail its tests, and can pass its tests and carry a false theorem.

Name the class you are relying on. A system that reports "verified" without saying which of these four produced the verdict has reported nothing a reader can act on. That is also why this paper never uses the words verified or certified in a formal sense. It says which check ran.

Do not treat a passing check as a statement about behaviour. Proposition 7.1 below states the practitioner form. The companion formal working series (2026), in preparation, states it more precisely: for its notion of a passing certificate, a pass means identity and shape matched, not that any property of behaviour was proven.

That series needs qualification. It contributes no measurement and states plainly that it contains no experiment and no benchmark, so its numbered results are cited here as formal design rules and never as established findings. It shares an author with the four code bases this paper places in the stack, and examines three of them as its own case studies, so it corroborates nothing said about them. And describing it as a companion to this series originates in this project; that series names no practitioner counterpart.

Design checks that survive redeployment. The same series gives a rule for Definition 3.9: a check reading only the wiring and the certificates survives being redeployed elsewhere, and a check reading the deployment map does not. A test that passes only on one machine's directory layout is the second kind.

Proposition 7.1 (The scope of a passing check)**.** *Let c be a check with pass criterion P applied to artifact a. A pass establishes exactly that $P(a)$ held at the moment c ran. It does not establish any property Q with $Q \neq P$, and in particular does not establish that a behaves correctly, unless P was itself a statement about a's behaviour that some procedure decided.*

Argument. The statement is close to definitional. It is worth stating because systems violate it in one specific way. A check reports on its own criterion, and where an implementation's criterion is narrower than its report, the gap is silent.

Both effects appear in the case studies. agent-os's contract verifier supports three rule types, checking file existence, file size, or key presence, and its dispatcher returns success for any rule it does not recognize (agent-os at `61cb3994da`, `src/agent_os/lib/agent_os/contracts/verify.ex`, lines 31 to 70). A contract declaring an unknown rule therefore reports a pass, and that pass means the criterion was vacuous, not that the artifact was good. AgentHero's schema validation checks a node's inputs and outputs where a schema exists, and does nothing where it does not (`crates/dag-executor/src/lib.rs`, lines 4309 to 4322). The pass again means exactly what was checked, and no more. So P must be read off the implementation, never off the report. □

7.3 A contract's life cycle

Figure 3 traces the artifacts and has no arrows back, because what happens on a failing verdict is control flow and belongs to Part III.

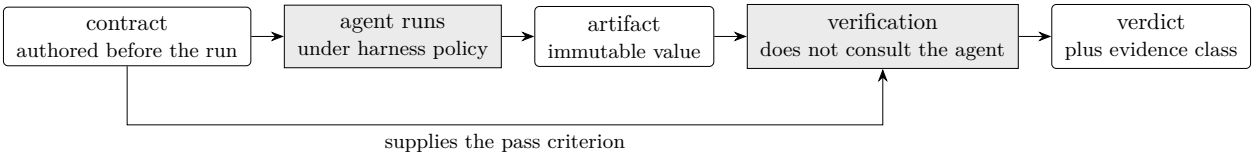

Figure 3: Artifacts and handoffs in the life of a contract, which is authored before the run and supplies the criterion that the verification applies afterwards, so that the verdict is decidable without asking the producing agent. No arrow returns from the verdict, because what a failing verdict causes is control flow and is defined in Part III.

The lower arrow carries the figure. The criterion comes from the contract, written before anyone knew what the agent would produce. A criterion written afterwards, in the light of the artifact, is a description.

8 An illustrative practitioner workflow

The workflow below is illustrative, a composite of the harness surfaces documented in Section 4.2 and the contract artifacts of Section 4.6 rather than a report of an observed team. It covers roles, artifacts, and handoffs. Control flow, iteration, and recovery belong to Part III.

8.1 Roles

Three roles separate by what each is accountable for. One person often holds all three, which is why separating them helps.

The *specifier* produces the contract. Their artifact is a written statement of what must be produced and how it will be checked. The review question is whether a program could decide satisfaction without asking the model.

The *operator* chooses the harness surface: which tools are exposed for this task, which permission mode applies, what bounds the session, and what the sandbox is. Their artifact is a configuration. The review question is whether every effectful step in the task has a named boundary.

The *reviewer* consumes evidence. Their artifact is a verdict with a named evidence class. The review question is whether they read the artifact or the agent's account of it. That distinction is Definition 3.8 in practice, and it is where the work most often goes wrong, because a summary is easier to read than a diff.

8.2 Artifacts and handoffs

Intent to contract. The specifier turns a sentence into a contract. What crosses this boundary is a document with declared outputs and declared checks. What makes it a handoff rather than a note is that the checks are decidable.

Contract to configuration. The operator reads the contract's declared outputs and effects and chooses a harness surface that bounds them. What crosses is a configuration: tool set, permission mode, sandbox policy, session bound. In one documented harness these are the plan mode, the settings file, the subagent definitions, and the lifecycle handlers [8]; in a cloud agent they include a hard session cap [22].

Configuration to run. The agent runs. Nothing crosses this boundary from the human until the run stops. The autonomy level of the run, in the sense of Part I's Definition 4.3, is the count of tool invocations between the human's decisions here.

Run to artifact. The run produces artifacts. The handoff is clean only if the artifacts can be read outside the run's session: files, diffs, structured outputs, recorded tool invocations. A transcript is not an artifact in this sense, because reading it is reading the agent's account.

Artifact to evidence. Verification applies the contract's criteria and produces evidence in one or more classes from Table 4. This is the handoff that decides whether the whole arrangement was worth setting up.

Evidence to verdict. The reviewer reads the evidence and decides. The one discipline that matters here is reading the evidence rather than the narration. An agent's summary of its own test run is model judgment about its own work, which Table 4 puts at the bottom of the ladder; the suite's exit status is executed-test evidence, two rows up.

A failing verdict often sends work back for another attempt under a bound. That repeated cycle is the loop of Part III's Definition 2.2, with the gate at its Definition 2.1 and the bound at its Definition 2.4. This Part stops at the handoffs.

8.3 Assumptions about context

Every handoff above spends context window, which Section 4.1 showed is smaller in practice than in specification. Deciding which tokens occupy that window, in what order, for each inference call is a practice with its own name and its own literature. Part V owns it as context engineering, its Definition 2.1. One vendor describes it as the set of strategies for curating and maintaining the optimal set of tokens during inference, and states that a model's ability to recall information from context falls as tokens increase [4].

Two other components sit in the state layer of Figure 2 and are developed in Part V: persistent memory, the store that carries state between sessions, and the governed data layer, which answers questions from systems of record with provenance and typed refusals.

9 Current deployment boundaries

Agentic engineering includes choosing the right execution method for each task. Five task classes show where direct authorship or a fixed workflow remains the correct component inside the larger system.

Task class	Evidence	Design and qualification
Real issues in large, mature repositories the developer knows well	Allowing AI increased completion time by 19%. The same developers predicted a 24% reduction beforehand and still believed afterwards that they had been sped up by about 20% [12]	A trial randomized per issue, with 16 experienced maintainers and 246 issues. The authors disclaim generalization to novices or unfamiliar code, and the opposing trial on 95 freelancers and a greenfield task [41] is tabled beside it in Part I's Section 5
Issue resolution where a fixed pipeline suffices	A three-phase pipeline with no agentic loop resolved 32.00% of 300 SWE-bench Lite instances at $0.70 per instance, beating contemporary agent scaffolds at lower cost [54]	One benchmark split. The same paper reports that 4.3% of that split contains the ground-truth patch in the issue text
Anything where accuracy per dollar is the criterion	The paper 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 [27]	A meta-analysis of others' benchmarks rather than new task data
Work where security requirements are not stated up front	None of 15 professional engineers specified security requirements in initial prompts, even when they had the relevant knowledge. Experience did not predict outcomes [10]	A peer-reviewed qualitative study of 15 engineers, combining interviews with task observation
Codebases where shared understanding is the constraint	Two entries in the caution ring of one volume: codebase cognitive debt, the growing gap between an implementation and a team's shared understanding of how and why it works, and coding throughput as a productivity measure [48]	A first-party industry trend report rather than a measurement. It covers a single volume, and rings change between volumes

Table 5: Task classes that currently require direct authorship, fixed control, or stronger human oversight. The third column states the design boundary established by each study.

The randomized trial gives the clearest measured boundary for current agents [12]. Sixteen developers worked on their own repositories under high contribution standards, where their context advantage was largest. An earlier randomized trial found that a completion assistant made freelancers 55.8% faster, with a 95% confidence interval of 21% to 89%, on one greenfield JavaScript task among 95 professional programmers recruited on a freelancing marketplace [41]. Part I's Section 5 places the trials beside each other. Their different populations make task allocation and acceptance criteria engineering decisions rather than assumptions about a universal speed effect.

What the trial does show is a finding about verification. A practitioner's belief about the speedup is not evidence for the speedup. The same developers who were slowed by 19% still believed

afterwards that they had been sped up by about 20%.

Review cost also varies by agent and organization. A field study of agent-authored pull requests found one agent's pull requests had both the highest merge rate, 87.5% against 75.1% for human pull requests, and the fastest median merge time [43]. The redefined discipline must measure this variation and route work accordingly.

Two larger studies expose the quality-control burden. An observational study of 302,600 verified AI-authored commits across 6,299 repositories found that assistants introduce more correctness and security issues than they fix, and that 22.7% of AI-introduced issues remain unresolved at the latest repository version [32]. A quasi-experiment with matched controls found static-analysis warnings up about 18% and cognitive complexity up about 39% in repositories adopting autonomous agents [1]. These results make independent verification and maintenance policy part of the agentic engineering stack.

On documented reverts. We found no documented case of an organization adopting coding agents broadly and then withdrawing them for measured quality or cost reasons, in the sources we reviewed (cutoff 2026-09-01).

The documented pattern is mitigation. Incidents produced added separation and planning modes rather than withdrawal. Existing restrictions govern agent-generated contributions from outside a project and are argued on reviewer-economics grounds. Organizations are responding to agentic production by engineering stronger control boundaries.

10 Limitations

The stack has not been tested against systems it did not come from. Six layers is a choice. A reader could put the sandbox inside the harness, or split governance into policy and audit, and produce a defensible alternative. The claim made here is narrower: the layer boundaries are where the design decisions sit. A system whose important decisions fall inside a layer rather than between layers would weaken it.

The case studies are one lineage, not four. agent-os, AgentHero, CatDB, and ContextFS share one author with this series. The companion formal working series (2026), in preparation, examines three of the same code bases as its own case studies. They are four instances of one lineage, not four independent confirmations.

Every placement in Table 3 was read from source at a pinned commit, and none was executed. The claims are that the code contains these structures, not that the systems behave as the structures suggest. Two of the four record substantial incompleteness in their own documents, and Section 4 reports it.

No measurement supports the compositional argument. This is the largest limitation in the paper, and Section 5.5 states it at length. The argument for typed boundaries and explicit, recorded effects rests on three things: a failure taxonomy that classifies traces rather than measuring an intervention [15], inspection of four code bases, and a programming-language tradition whose own defect-rate advantage did not survive reproduction [14].

The decisive comparison has not been run. None of the sources we located compares defect escape rates between agent systems built with and without typed boundaries at matched verification effort.

Two task classes where the primitives do not help. The randomized trial named in Table 5 identifies the first: real issues in large, mature repositories the developer knows well, under high contribution standards [12]. Nothing in this anatomy addresses that case. The human's advantage there is context that sits in no window and is expensive to put into one, and a better tool registry does not touch it.

The second comes from a vendor's own account of where its multi-agent research system fits poorly: domains requiring all agents to share the same context, or carrying many dependencies between agents [5]. The layered stack described here does not make shared context cheaper.

Sources of unequal weight are cited alongside each other. This paper cites peer-reviewed studies, preprints, vendor engineering blogs, vendor documentation, and first-party code. Each is named as such where it is cited, and the reader should discount accordingly. A vendor blog describing a vendor's own system is a primary source for what the system does and a weak source for whether the approach works. Two of the disputes reported here have a commercially interested party on one side, and both are marked.

The term inventory is dated. This field's vocabulary moved substantially between 2024 and 2026 and shows no sign of settling. The provenance research behind Section 3 found no canonical coinage for the central term [49]. The definitions here are stipulations chosen to be checkable, not reports of consensus. A reader in two years should expect the words to have moved even if the objects have not.

11 Conclusion

An agentic engineering system is made of nine primitives, each defined so that an observer or a program could check whether a given system has one, and each located in source at a pinned commit.

They layer into six proposed positions, with contracts as boundaries cutting across the layers rather than residing in any one. The proposal's value is that it makes a specific question askable at each boundary, and nobody has ratified it.

What a practitioner does with the primitives is design and check compositions. Every primitive is either a step or a boundary between steps. Five operators with stated guarantees cover the arrangements that appear in practice, and the vendor pattern vocabulary maps onto them. Three rules follow: put effects behind explicit boundaries and record every one, never edit an artifact in place, and check at the boundary rather than inside the step.

The evidence for those rules is a design argument, not a measurement. One comparison would settle the question: defect escape between agent systems built with and without typed boundaries, at matched verification effort. Nobody has run it.

Three Parts build on what is defined here. Part III composes the plan, act, and verify cycle as a bounded composition behind a gate; its Definition 2.2 is the loop this Part has cited forward throughout. Part IV composes agents behind contracts between agents, a subtype of Definition 3.4. Part V instantiates the state layer of Figure 2, developing at its Definitions 2.2 and 2.3 the two components this Part named and handed on.

References

[1] Agarwal, S., He, H., and Vasilescu, B. AI IDEs or autonomous agents? Measuring the impact of coding agents on software development. Preprint, arXiv:2601.13597, 20 January 2026, revised 27 January 2026. https://arxiv.org/abs/2601.13597

[2] Apache Airflow. Core concepts: DAGs, and project documentation. Vendor and open-source documentation, version 3.3.1 at access. https://airflow.apache.org/docs/apache-airflow/stable/project.html

[3] Anthropic. Building effective agents. Vendor engineering blog, 19 December 2024. https://www.anthropic.com/engineering/building-effective-agents

[4] Anthropic. Effective context engineering for AI agents. Vendor engineering blog, 29 September 2025. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

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

[6] Anthropic. Writing effective tools for AI agents. Vendor engineering blog, 11 September 2025. https://www.anthropic.com/engineering/writing-tools-for-agents

[7] Anthropic. Introducing the Model Context Protocol. Vendor announcement, 25 November 2024. https://www.anthropic.com/news/model-context-protocol

[8] Anthropic. Claude Code documentation: common workflows, and the hooks reference. Vendor documentation, undated and continuously updated; accessed 2 September 2026. https://code.claude.com/docs/en/common-workflows and https://code.claude.com/docs/en/hooks

[9] Anthropic. How Claude remembers your project: CLAUDE.md files and auto memory. Vendor documentation, undated and continuously updated. https://code.claude.com/docs/en/memory

[10] Bappy, F. H., Hossain, T., Meheraj, S. M., Akhand, A. S., Tabassum, T., Zaman, T. S., Hasan, R., and Islam, T. From preventive to reactive: how AI coding assistants transform developers' security awareness. SOUPS 2026, peer-reviewed. Preprint arXiv:2605.23130. https://arxiv.org/abs/2605.23130

[11] Banu, B. Harness engineering as categorical architecture: structural guarantees are harness-level properties. Preprint, arXiv:2605.12239, 16 pp., marked preprint with feedback welcome. https://arxiv.org/abs/2605.12239

[12] Becker, J., Rush, N., Barnes, E., and Rein, D. Measuring the impact of early-2025 AI on experienced open-source developer productivity. METR research report; randomized controlled trial. Preprint arXiv:2507.09089, 12 July 2025. https://arxiv.org/abs/2507.09089

[13] Bent, B. The term "agent" has been diluted beyond utility and requires redefinition. AIES 2025, peer-reviewed. Preprint arXiv:2508.05338, 7 August 2025. https://arxiv.org/abs/2508.05338

[14] Berger, E. D., Hollenbeck, C., Maj, P., Vitek, O., and Vitek, J. On the impact of programming languages on code quality: a reproduction study. *ACM Transactions on Programming Languages and Systems* 41(4), 2019, pp. 1-24. Peer-reviewed reproduction. https://doi.org/10.1145/3340571

[15] 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., and 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

[16] Hong, K., Troynikov, A., and Huber, J. (Chroma). Context rot: how increasing input tokens impacts LLM performance. Vendor technical report, not peer-reviewed, 14 July 2025. https://www.trychroma.com/research/context-rot

[17] Cursor. Cloud agents documentation. Vendor documentation, undated. https://cursor.com/docs/cloud-agent

[18] Cursor. Rules documentation. Vendor documentation, undated.
https://cursor.com/docs/context/rules

[19] Kurt, W. (dottxt). Say what you mean: a response to "Let me speak freely". Vendor blog; the publisher sells structured-generation tooling. No date shown on the page.
https://blog.dottxt.ai/say-what-you-mean.html

[20] Erlang/OTP. Supervisor behaviour. Language reference and standard documentation.
https://www.erlang.org/doc/system/sup_princ.html

[21] Geng, S., Cooper, N., Moskal, M., Jenkins, S., Berman, S., Ranchin, N., West, R., Horvitz, E., and Nori, H. JSONSchemaBench: a rigorous benchmark of structured outputs for language models. Preprint, arXiv:2501.10868, 18 January 2025. https://arxiv.org/abs/2501.10868

[22] GitHub. About GitHub Copilot cloud agent. Vendor documentation, undated.
https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent

[23] Delimarsky, D., and GitHub. Spec-driven development with AI: get started with a new open source toolkit. Vendor blog, 2 September 2025; and the spec-kit repository README.
https://github.blog/ai-and-ml/generative-ai/
spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/

[24] Hassan, A. E., Li, H., Lin, D., Adams, B., Chen, T.-H., Kashiwa, Y., and Qiu, D. Agentic software engineering: foundational pillars and a research roadmap. Academic position and roadmap paper. Preprint arXiv:2509.06216, 7 September 2025. https://arxiv.org/abs/2509.06216

[25] Hsieh, C.-P., Sun, S., Kriman, S., Acharya, S., Rekesh, D., Jia, F., Zhang, Y., and Ginsburg, B. (NVIDIA). RULER: what's the real context size of your long-context language models? COLM 2024, peer-reviewed. Preprint arXiv:2404.06654. https://arxiv.org/abs/2404.06654

[26] Hughes, J. Why functional programming matters. *The Computer Journal* 32(2), 1989, pp. 98-107. Peer-reviewed. https://doi.org/10.1093/comjnl/32.2.98

[27] Kapoor, S., Stroebl, B., Siegel, Z. S., Nadgir, N., and Narayanan, A. AI agents that matter. Preprint, arXiv:2407.01502, 1 July 2024. https://arxiv.org/abs/2407.01502

[28] Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. DSPy: compiling declarative language model calls into self-improving pipelines. Preprint, arXiv:2310.03714, 5 October 2023. https://arxiv.org/abs/2310.03714

[29] Kwa, T., West, B., Becker, J., et al. (METR). Measuring AI ability to complete long software tasks. Preprint, arXiv:2503.14499, 19 March 2025. https://arxiv.org/abs/2503.14499

[30] LangChain. LangGraph graph API documentation. Vendor and open-source documentation, version 1.2.11 at access. https://docs.langchain.com/oss/python/langgraph/graph-api

[31] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. Lost in the middle: how language models use long contexts. *Transactions of the ACL*, 2024. Peer-reviewed. Preprint arXiv:2307.03172. https://arxiv.org/abs/2307.03172

[32] Liu, Y., Widyasari, R., Zhao, Y., Irsan, I. C., Chen, J., and Lo, D. Debt behind the AI boom: a large-scale empirical study of AI-generated code in the wild. Preprint, arXiv:2603.28592, 30 March 2026, revised 26 April 2026. https://arxiv.org/abs/2603.28592

[33] Liu, J., Wang, K., Chen, Y., Peng, X., Chen, Z., Zhang, L., and Lou, Y. Large language model-based agents for software engineering: a survey. Accepted by *ACM TOSEM*. Preprint arXiv:2409.02977, 4 September 2024. https://arxiv.org/abs/2409.02977

[34] Model Context Protocol. Specification: tools; and specification versioning. Open standard. Revision identifiers 2024-11-05, 2025-03-26, 2025-06-18, 2025-11-25, and 2026-07-28, the last being the current version at the access date.
https://modelcontextprotocol.io/specification/2026-07-28/server/tools

[35] Modarressi, A., Deilamsalehy, H., Dernoncourt, F., Bui, T., Rossi, R. A., Yoon, S., and Schütze, H. NoLiMa: long-context evaluation beyond literal matching. ICML 2025, peer-reviewed. Preprint arXiv:2502.05167, 7 February 2025. https://arxiv.org/abs/2502.05167

[36] Moseley, B., and Marks, P. Out of the tar pit. Preprint, never formally published, dated 6 February 2006. https://curtclifton.net/papers/MoseleyMarks06a.pdf

[37] OpenAI. Function calling and other API updates. Vendor announcement, 13 June 2023. https://openai.com/index/function-calling-and-other-api-updates/

[38] OpenAI. Structured model outputs. Vendor documentation; feature launched 6 August 2024 per the API changelog. https://developers.openai.com/api/docs/guides/structured-outputs

[39] OpenAI. Agents SDK documentation: handoffs and guardrails. Vendor documentation, undated. https://openai.github.io/openai-agents-python/

[40] Panickssery, A., Bowman, S. R., and Feng, S. LLM evaluators recognize and favor their own generations. NeurIPS 2024, peer-reviewed. Preprint arXiv:2404.13076. https://arxiv.org/abs/2404.13076

[41] Peng, S., Kalliamvakou, E., Cihon, P., and Demirer, M. The impact of AI on developer productivity: evidence from GitHub Copilot. Randomized controlled trial. Preprint, arXiv:2302.06590, 13 February 2023. https://arxiv.org/abs/2302.06590

[42] Petricek, T. What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming* 2(3), 2018. Peer-reviewed. https://doi.org/10.22152/programming-journal.org/2018/2/12

[43] Popescu, R. M., Gros, D., Botocan, A., Pandita, R., Devanbu, P., and Izadi, M. Investigating autonomous agent contributions in the wild: activity patterns and code change over time. Preprint, arXiv:2604.00917, 1 April 2026. https://arxiv.org/abs/2604.00917

[44] Ray, B., Posnett, D., Filkov, V., and Devanbu, P. A large scale study of programming languages and code quality in GitHub. FSE 2014, pp. 155-165, peer-reviewed; journal version *CACM* 2017. https://doi.org/10.1145/2635868.2635922

[45] Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., et al. (Anthropic). Towards understanding sycophancy in language models. Preprint, arXiv:2310.13548, 20 October 2023, revised 10 May 2025. https://arxiv.org/abs/2310.13548

[46] Tam, Z. R., Wu, C.-K., Tsai, Y.-L., Lin, C.-Y., Lee, H.-Y., and Chen, Y.-N. Let me speak freely? A study on the impact of format restrictions on performance of large language models. Preprint, arXiv:2408.02442, 5 August 2024. https://arxiv.org/abs/2408.02442

[47] Temporal. Workflow execution documentation. Vendor documentation, undated. https://docs.temporal.io/workflow-execution

[48] Thoughtworks. Technology Radar, Volume 34, April 2026: the techniques entries "codebase cognitive debt" and "coding throughput as a measure of productivity", both in the caution ring. Industry trend report, first party. https://www.thoughtworks.com/radar/techniques/codebase-cognitive-debt

[49] Willison, S. Agentic engineering patterns. Practitioner guide, introduction dated 23 February 2026. Cited as one of several independent parties that converged on the term between roughly mid-2025 and mid-2026; the guide introduces the term as its author's own and does not attribute it to anyone else. https://simonwillison.net/guides/agentic-engineering-patterns/

[50] Wadler, P. Comprehending monads. Proceedings of the 1990 ACM conference on LISP and functional programming, pp. 61-78. Peer-reviewed. https://doi.org/10.1145/91556.91592

[51] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: an open-ended embodied agent with large language models. Preprint, arXiv:2305.16291, 25 May 2023, revised 19 October 2023. https://arxiv.org/abs/2305.16291

[52] Wang, Y., Zhong, W., Huang, Y., Shi, E., Yang, M., Chen, J., Li, H., Ma, Y., Wang, Q., and Zheng, Z. Agents in software engineering: survey, landscape, and vision. Academic survey. Preprint arXiv:2409.09030, 13 September 2024. https://arxiv.org/abs/2409.09030

[53] Wiz Research. s1ngularity: supply chain attack leaks secrets on GitHub. First-party vendor security research, 26 August 2025. https://www.wiz.io/blog/s1ngularity-supply-chain-attack

[54] Xia, C. S., Deng, Y., Dunn, S., and Zhang, L. Agentless: demystifying LLM-based software engineering agents. Preprint, arXiv:2407.01489, 1 July 2024, revised 29 October 2024. Figures cited here are from v2. https://arxiv.org/abs/2407.01489

[55] Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: agent-computer interfaces enable automated software engineering. NeurIPS 2024, peer-reviewed. Preprint arXiv:2405.15793. https://arxiv.org/abs/2405.15793

[56] Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: theorem proving with retrieval-augmented language models. Preprint, arXiv:2306.15626, 27 June 2023. https://arxiv.org/abs/2306.15626

[57] Yorgey, B. Abstraction, intuition, and the "monad tutorial fallacy". Blog post, 13 January 2009. https://byorgey.github.io/blog/posts/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy.html

[58] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. NeurIPS 2023 Datasets and Benchmarks, peer-reviewed. Preprint arXiv:2306.05685. https://arxiv.org/abs/2306.05685