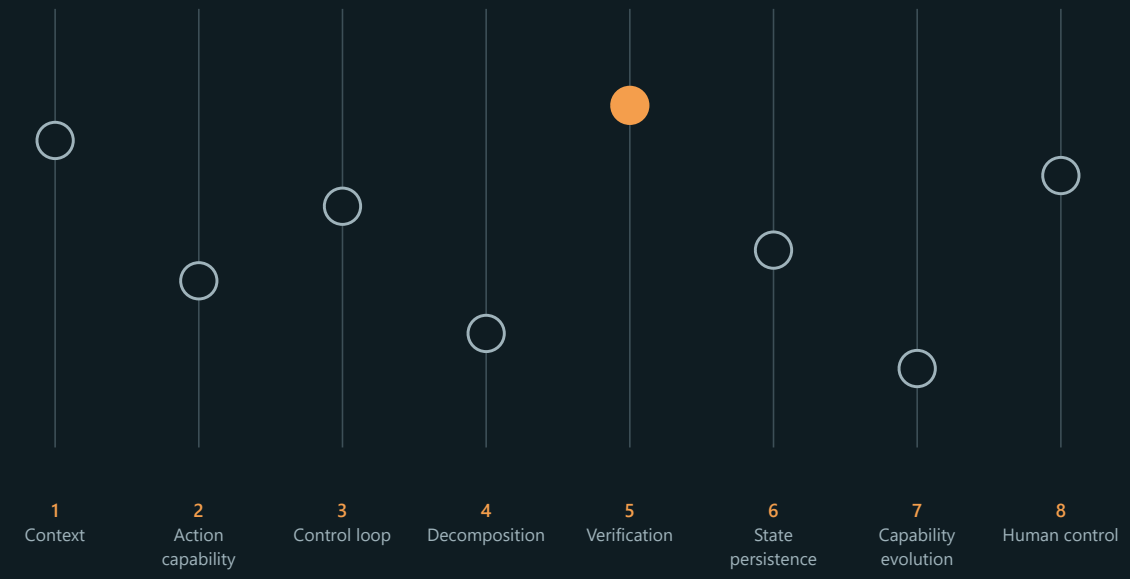
RESEARCH REPORT

Agentic AI Methodologies

Eight levers, what they owe to human organization, and a matrix to choose by situation



Daniel

26 July 2026

Contents

Executive Summary 3

PART I What an agent is, and what this report sets out to do 7
1 Agentic AI is an assembly around a model, not a new kind of model 8
2 An agent starts where the model, not the code, directs the next step 10
3 Six terms and four distinctions this report fixes before it argues 13
4 The plan: tell the methods apart, weigh the evidence, choose by situation 16

PART II One loop under every agent, and eight levers to configure it 18
5 Every agent runs the same loop, so the loop tells no two agents apart 19
6 Context: what the agent knows is rebuilt, and paid for, on every turn 22
7 Action capability: what matters is what cannot be undone 26
8 Control loop: code or model, the one line every major vendor draws 30
9 Decomposition and specialization: every split is paid for in coordination 34
10 Verification and conformance: results are checked, the path almost never is 38
11 State persistence: nothing survives a turn unless someone wrote it down 43
12 Capability evolution: the hard part is choosing which new skills to keep 47
13 Human control: a gate approved 93% of the time is a reflex, not a control 50
14 The list is not closed: two dimensions sit outside the eight levers 54
15 Tested on two opposed agents, the levers are a design grid, not a taxonomy 56

PART III The vendors' catalogue, read as settings of the levers 60
16 Patterns and frameworks are lever settings under different names 61
17 Two protocols have settled: MCP toward tools, A2A between agents 65
18 The method behind this report holds by convention, not by mechanism 69
19 Practitioners disagree on three points, and practice has begun to answer 74
20 The field claims much and measures little: eleven claims, sorted 77

PART IV What agents owe to human organization, and where the likeness ends 80
21 The human model is in the foundations of agent design 81
22 Procedures, roles and escalation carry over from human organizations 84
23 The model brings three problems no organization has had to manage 88
24 Of four things an organization relies on, software can stand in for two 91
25 A limit of engineering or of principle? The evidence does not decide 94
26 People set the frame, and agents do the work inside it 97

PART V Cost decides, and the situation sets the dials 100
27 Cost is what decides between two configurations 101
28 Choosing takes three things: a structure, a situation and a measure 104
29 For each lever: a default, a dial, its price, and the tools of 2026 107
30 A regulated enterprise and a startup turn most dials in opposite directions 114
31 A configuration is a hypothesis until it is measured on five dimensions 118
32 A place to start: a bug-triage pipeline for a mid-sized team 122
33 The result: a situational matrix, a frontier that moves, and five gaps 127

Sources 132

Executive Summary

Agentic artificial intelligence methodologies are transitioning from exploratory prototypes to production systems. Beneath dozens of competing frameworks and commercial taxonomies lies a single, irreducible first principle: a continuous loop of tool use, planning, and reflection (The Batch 241). This shared cognitive foundation means that abstract capability taxonomies fail to discriminate between systems. An agent that runs Monte Carlo tree search and a basic loop with retries are described in the same words by such taxonomies (Inside the Scaffold, arXiv 2604.03515). Architecture describes how production methodologies differ. What decides between them is their economic boundary: the cost in tokens, latency, and human attention required to enforce reliability. This report works through the fragmentation of the field with three core hypotheses, three empirical pivot findings, and a unified design framework.

The three hypotheses and their verdicts

The investigation tracked three foundational hypotheses regarding the structure, organizational origin, and delegation limits of autonomous agents. Each hypothesis began as an open inquiry during scoping and was systematically refined against empirical literature:

1. **H1 (Reducibility: from taxonomy to design framework).** The diverse catalog of agentic methodologies reduces to eight invariant structural levers: Context, Action capability, Control loop, Decomposition and specialization, Verification and conformance, State persistence, Capability evolution, and Human control. Applied to two opposed coding agents, the levers separate them on most dials, so they describe real differences. They do not sort agents into species, because agents sit on continuous spectra and combine the same building blocks. The levers are a design grid, which Part V sets against the 31 layers of a tool catalogue compiled for this report.
2. **H2 (Organizational convergence: real resemblance, unnecessary convergence).** Frameworks explicitly borrow structures from human institutions, including standard operating procedures (`Code = SOP(Team)`), assembly-line handoffs, and organizational divisions of labor (MetaGPT, arXiv 2308.00352). However, convergence is partial and unnecessary. Agentic systems face an irreducibly distinct layer of model-specific failures without human equivalent: reasoning-trace unfaithfulness (Anthropic), prompt injection vulnerabilities (arXiv 2510.09023), context degradation, and multi-turn role drift (arXiv 2512.07665).
3. **H3 (Delegation boundary: indecision as an empirical result).** Human organizations rely on social reputation and long-horizon career incentives to ensure alignment. Software engineering provides functional substitutes for accountability through immutable audit traces and operator takeover (AI Act, art. 14), and replaces tacit context at high documentation cost. Yet, no operational substitute exists for social reputation or career ambition. Whether this delegation boundary reflects temporary engineering limits or

universal structural constraints remains unresolved, and documenting this indecision is an explicit empirical result (MAST, arXiv 2503.13657).

Three pivotal findings

All three hypotheses converge on a shared reality: production agentic systems are constrained primarily by economics and containment. The report anchors its design rules on three controlled findings from recent literature:

- **Token consumption dictates performance.** On complex research benchmarks, token consumption alone explains **80 %** of the performance variance across configurations (Anthropic). By the same vendor's account, a multi-agent system uses about **15×** the tokens of a chat exchange, where a single agent uses about four times.
- **Per-turn human supervision fails in production.** Production telemetry reveals that human operators grant roughly **93 %** reflex approvals when subjected to frequent permission prompts (Anthropic). High prompt frequency induces supervision fatigue, transforming safety checkpoints into thoughtless clicks. Containment must rely on capability isolation, virtual machines, and network sandboxes rather than manual approvals.
- **Runtime trajectory conformance remains unsolved.** Across **12** published prompt-injection defenses, all 12 are bypassed at rates above **90 %** under adaptive evaluations (The Attacker Moves Second, arXiv 2510.09023). Verifying agent actions after execution is tooled; verifying adherence to a prescribed execution path in flight remains an open vulnerability.

The situational architecture framework

To replace trial-and-error prototyping, the report synthesizes these findings into the **Situational Architecture Triangle**, connecting structural mechanisms, situational criteria, and multidimensional measurement (arXiv 2511.14136). The framework balances three interconnected pillars:

- **Structure (The Eight Levers).** Systems are configured across the eight levers, setting explicit dials for context boundaries, action permissions, control loop determinism, sub-agent decomposition, automated verification, persistent state stores, capability updates, and human escalation gates.
- **Situation (The Four Operational Criteria).** The optimal setting of each dial depends entirely on the operating environment:
  1. *Process Predictability:* The degree to which execution steps can be specified in advance.
  2. *Human Attentional Budget:* The cognitive capacity available to review outputs and handle escalations.
  3. *Organizational Risk Tolerance:* The cost of catastrophic or irreversible action failures.
  4. *Velocity Requirement:* The latency tolerance between trigger and final resolution.
- **Measure (The CLEAR Framework).** Architectural success cannot be evaluated through single-run accuracy benchmarks. The report adopts the multidimensional CLEAR framework (arXiv 2511.14136): Cost Per Solved Task, Latency, Efficacy (task completion rate), Assurance (guardrail trip rate), and Reliability (consistency across repeated runs).

The result, and the gaps it accepts

The result of the inquiry is a matrix, presented in the final chapter. For each of the eight levers it gives the question the lever asks, a default setting, the situation that justifies turning the dial, what turning it costs, and what stays with a human. A reference configuration for a mid-sized engineering team shows the matrix applied: a deterministic orchestrator for the overall process, a single agent that writes, and narrow model-driven steps inside a sandbox (Cognition). It is offered as a starting point and has not been tested in production.

The settings in that matrix are dated. On every lever, a documented change in 2025 or 2026 moved what is possible, from million-token context windows to classifiers that stand in for a human approver. The eight questions are the stable part of this report, and the answers have to be revisited as the tools change.

This methodology accepts five gaps:

1. The literature lacks controlled comparisons evaluating diverse agent architectures at constant model capability and cost budgets.
2. Trajectory conformance remains untooled, preventing systems from guaranteeing procedural compliance under untrusted inputs.
3. Indirect prompt injection remains an active vulnerability without robust defense.

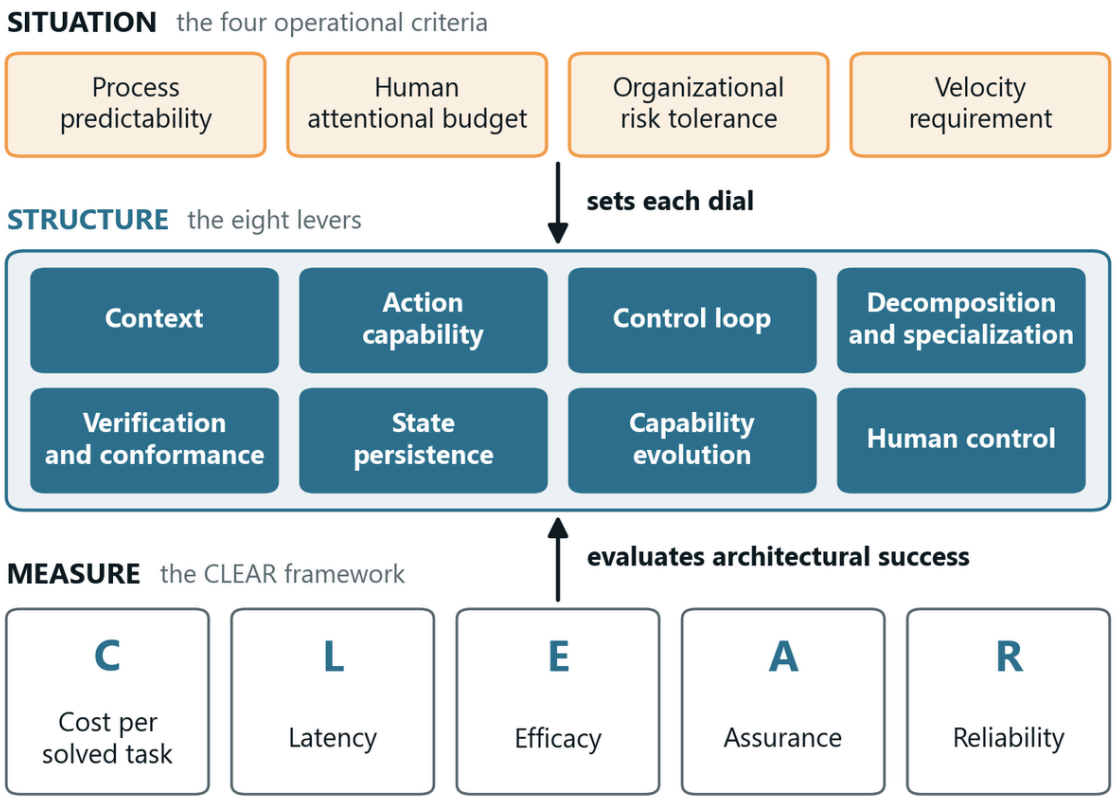


FIGURE 0.1 The situation sets each of the eight levers, and the five dimensions of the CLEAR framework evaluate the result. Diagram drawn by this report.

4 The CLEAR framework and the proposed situational dials require extensive empirical validation across varied industry workloads.
5 Whether what an agent cannot carry today marks a technical limit or a permanent one remains undecided.

PART I

What an agent is, and what this report sets out to do

1 Agentic AI is an assembly around a model, not a new kind of model 8
2 An agent starts where the model, not the code, directs the next step 10
3 Six terms and four distinctions this report fixes before it argues 13
4 The plan: tell the methods apart, weigh the evidence, choose by situation 16

CHAPTER 1

Agentic AI is an assembly around a model, not a new kind of model

Before asking how agentic systems should be built, a reader needs to know what kind of thing an agentic system is. This chapter places agentic AI on the map of artificial intelligence, and its answer shapes the rest of the report. Agentic AI is not a new family of models. It is a way of assembling software around a model that already exists.

The word "AI" covers at least three families, and they differ in what they are trained to do. **Discriminative models** learn to separate: given labeled examples, they classify an input or predict a number. A spam filter is one. **Reinforcement learning** systems learn to act: they try, receive a reward or a penalty, and adjust their behavior inside a bounded environment such as a game. **Generative models** learn to produce: they model how text, code, images or audio are distributed, and generate new sequences that resemble what they were trained on.

Two terms inside the generative family are often used as if they meant the same thing. Stanford's Center for Research on Foundation Models, the group that coined the first of them, keeps them apart (Stanford CRFM). A **foundation model** is defined by its function: a large model trained on broad data, without task-specific labels, that can then be adapted to many tasks. A **large language model**, or LLM, is defined by how it is made: it predicts the next fragment of a text from the fragments before it. Every LLM in common use is a foundation model. The reverse does not hold, since foundation models also exist for images, audio and molecular biology.

Agentic AI appears nowhere in that list of families, and Figure 1.1 shows why. Nothing new is trained. An agentic system takes a foundation model and wraps software around it. That software calls the model in a loop, lets it use tools through programming interfaces, and keeps track of the work from one call to the next (IBM). A generative model on its own produces an output and stops. The same model placed inside such an assembly can act on its environment, observe the result, and pursue a goal over many steps (Google Research).

An example makes the difference concrete. Asked to fix a failing test, a language model alone can only write text that looks like a fix. Wrapped in an agentic assembly, it can open the file, change it, run the test, read the error, and try again.

The institutions that classify AI have not drawn this boundary. The European Union's AI Act gives no legal definition of an "AI agent". It defines AI systems broadly, as machine-based systems operating with varying degrees of autonomy, and regulates them by level of risk, whatever their internal structure. The NIST AI Risk Management Framework, the reference published by the United States standards agency, has a risk profile for generative systems and none for agents.

The companies that build the models do not agree either. Anthropic, which develops the Claude models, draws a sharp line between workflows and agents. OpenAI, which develops the GPT models, describes a continuous scale of autonomy with no line at all.

This report therefore starts from an engineering position: agentic AI is an assembly built around a foundation model. That position settles what an agent is made of. It leaves open where the model ends and the agent begins, and the two companies just cited give opposite answers. The next chapter sets those answers side by side and chooses one.

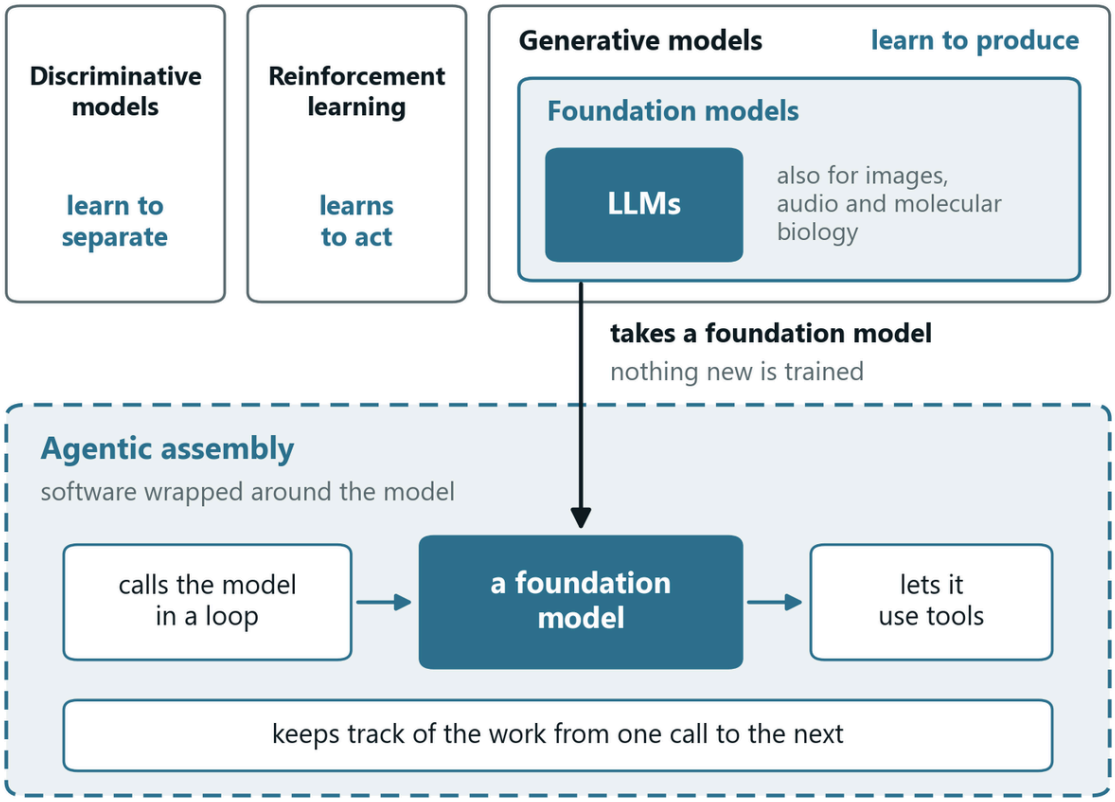


FIGURE 1.1 The three families of AI, with LLMs nested inside foundation models (Stanford CRFM). An agentic assembly trains nothing new: it wraps software around one existing model (IBM). Diagram drawn by this report.

CHAPTER 2

An agent starts where the model, not the code, directs the next step

If an agent is an assembly around a model, where does the model stop and the agent start? This chapter compares the answers in circulation and picks the one the report will use. The field has no agreed answer, and the choice has consequences: it decides what a team instruments, what it checks, and where it expects failures.

Two answers dominate, and they differ in kind. One draws a line. The other describes a slope.

The line: who directs the next step

Anthropic gives the sharpest version (Building Effective Agents). It does not ask how capable the model is. It asks who controls execution. A system is a **workflow** when models and tools are orchestrated through code paths written in advance. It is an **agent** when the model directs its own process, choosing which tools to use and judging its own progress. Anthropic calls both of them agentic systems, so the distinction separates two architectures, not two levels of intelligence.

The test this definition gives is a single question: at each step, does the code or the model decide what comes next? The two answers buy different things. A workflow is consistent, its duration is bounded, and its cost per run can be predicted. An agent adapts to a task nobody specified in full, and pays for that freedom with behavior that varies from run to run and errors that build on one another.

The slope: degrees of agenticness

OpenAI describes no threshold (OpenAI for Developers 2025). In its account a system becomes more agentic as it gains capabilities and works over longer horizons. The progression runs through four stages:

1. A single prompt and a single response, with no tools.
2. Tool use inside one conversational turn.
3. A chain of turns carrying out a structured task.
4. Open-ended pursuit of a goal over days or weeks.

Seen this way, choosing an architecture is a matter of tuning. The designer moves along a gradient of autonomy and trades cost, latency and quality against each other at each point.

Two further readings, and a silence

Other actors place the boundary in behavior. IBM speaks of “supervised autonomy”: an agent acts toward an objective under limited supervision, where a generative model only

produces an output (IBM). Google Research characterizes agentic systems by what they do over time. They interact with an environment across many steps, gather information they did not have at the start, and adjust their strategy from the feedback they receive (Google Research).

Regulators, as the previous chapter noted, stay silent. Neither the AI Act nor the NIST framework defines an agent, and both assess systems by risk.

The choice made in this report

No definition is authoritative, so the report has to choose one and say why. It adopts Anthropic's line.

The reason is practical. Asking who controls the loop tells a designer what to expect. Where code decides the sequence, each step can be checked in the ordinary way, and an error stays where it occurred. Where the model decides, every decision can vary, and the system needs additional machinery: a way to save its state, filters on what it does, and points where a human can step in. The eight levers of Part II describe that machinery, and the line drawn here is what lets them be read as concrete mechanisms.

The slope is not discarded. Chapter 8 returns to it, because real systems turn out to mix both modes at different scales.

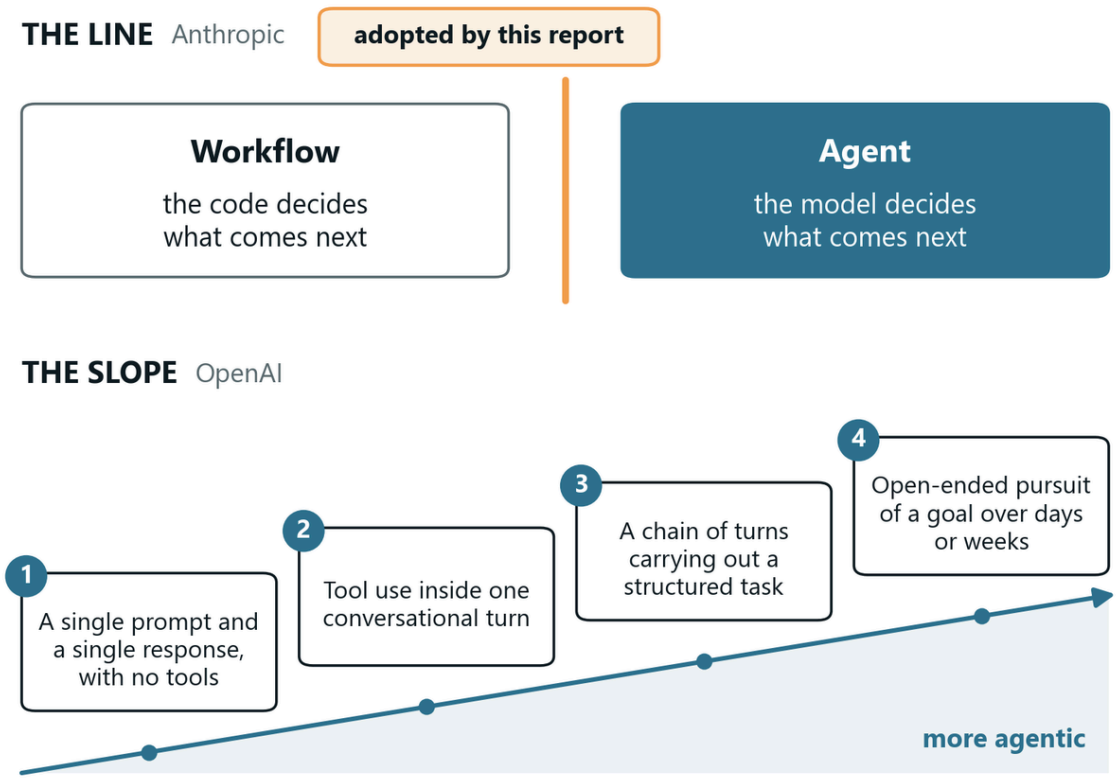


FIGURE 2.1 The line drawn by Anthropic (Building Effective Agents) and the slope described by OpenAI (OpenAI for Developers 2025). The diagram is drawn by this report, which adopts the line.

One obstacle remains before the analysis can begin. The field uses pattern, harness, scaffold and framework as if they were one word, and the next chapter fixes what each will mean here.

CHAPTER 3

Six terms and four distinctions this report fixes before it argues

A report that compares methodologies has to use its words consistently, and the field does not. This chapter fixes the vocabulary for everything that follows: six terms for how a system is built, and four distinctions that matter while it runs. The need is easy to show. Words such as *pattern*, *architecture*, *harness*, *scaffold*, *framework*, and *methodology* are used as if they were interchangeable. A software vendor who says "framework" usually means a packaged Python library, and an enterprise architect who says the same word means a governance policy. When a team mixes these terms, it loses sight of design trade-offs that separate code, runtime containment, and reasoning.

The definitions come in two groups, and they hold in every later chapter.

Six terms for how a system is built

As Figure 3.1 suggests, deploying production systems requires distinguishing between abstract design, executable code, runtime containment, and overarching doctrine. Conflating

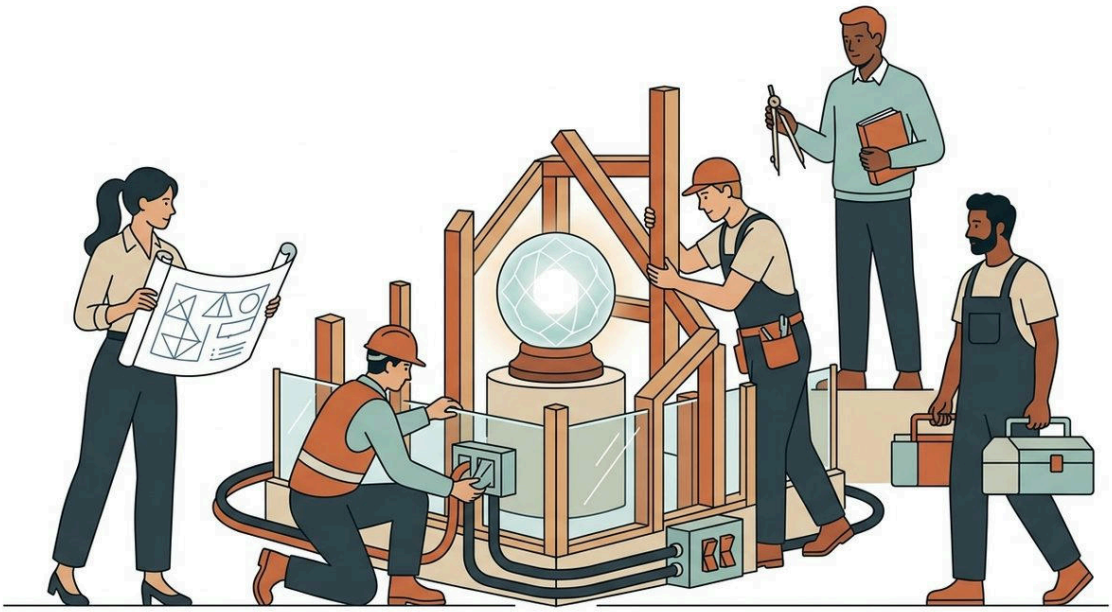


FIGURE 3.1 An editorial allegory by this report for five of the six terms defined below. The blueprint is the pattern, the timber frame the scaffold, the enclosure around it the harness, the toolboxes the framework, and the builder's rule book the methodology. The sixth, architecture, is not pictured.

these layers leads engineering teams to misdiagnose failure modes and misjudge implementation costs:

1. **Pattern:** A generalized, reusable template that resolves a recurring coordination or reasoning challenge (Google Cloud). Patterns define how a model interacts conceptually with tools and memory. Familiar examples include ReAct loops, reflection steps, and dynamic routing. Patterns are abstract designs, not executable implementations.
2. **Scaffold:** The concrete source code wrapping the foundation model that translates an abstract pattern into running software. Established through empirical source-code analysis across open-source code agents, the scaffold is the exact code implementing loop primitives, tool dispatch, error recovery, and context compaction (Inside the Scaffold, arXiv 2604.03515). Two systems implementing the same ReAct pattern can utilize radically different scaffolds, producing divergent cost and reliability profiles.
3. **Harness:** The operational execution environment that surrounds and isolates the model. The operational boundary is straightforward: if a component is not the foundation model itself, it belongs to the harness (arXiv 2603.25723). The harness manages virtual machine sandboxing, permission firewalls, API network proxies, credential management, and audit telemetry.
4. **Architecture:** The structural topology connecting cognitive nodes, data stores, external tools, and communication channels. Architecture describes how multiple models and scaffolds compose into an end-to-end system.
5. **Framework:** A packaged, distributable software library providing developers with off-the-shelf scaffolds and harness abstractions. Popular examples include LangChain, AutoGen, and CrewAI. While frameworks accelerate early development, enterprise teams often bypass generic framework abstractions to maintain deterministic control over execution state (Anthropic).
6. **Methodology:** The disciplined engineering doctrine governing how systems are designed, deployed, and audited. A methodology dictates how an engineering team assesses organizational constraints, configures structural dials across the software stack, and evaluates performance using multidimensional benchmarks.

Four distinctions at run time

While the macro hierarchy clarifies system packaging, everyday engineering failures frequently stem from conflating runtime primitives. We establish four mandatory distinctions:

1. **Context vs. State vs. Memory:**
   - *Context* is the ephemeral token sequence passed into a single inference call (CoALA, arXiv 2309.02427). It is strictly bounded by window limits and consumes active compute.
   - *State* is the serializable, persistent record of execution progress maintained by the scaffold. State stores checkpoints, blackboard variables, and task status across session boundaries (Lever 6).
   - *Memory* refers to external knowledge stores queried dynamically into context. Confusing context with memory leads teams to append raw conversational histories, causing token inflation and reasoning degradation.

2 **Tool vs. Skill:**

- A *tool* is a stateless interface specification enabling the model to invoke an external endpoint or executable function.
- A *skill* is an accumulated, multi-step executable routine stored in procedural memory (Voyager, arXiv 2305.16291). Skills expand an agent's behavioral inventory over time through autonomous composition (Lever 7).

3 **Guardrail vs. Sandbox vs. Gate:**

- A *guardrail* is a probabilistic filter evaluating text prompts or outputs against policy rules.
- A *sandbox* is a deterministic, operating-system isolation boundary restricting physical access to filesystems, network egress, and compute resources (Anthropic). Treating text guardrails as security boundaries is flawed, as adaptive prompt injections bypass them at rates exceeding 90 %.
- A *gate* is a synchronous execution pause point requiring automated verification checks or human approval before proceeding (Levers 5 and 8).

4 **Trajectory vs. Trace:**

- A *trajectory* is the cognitive sequence of thoughts, tool selections, and environmental observations produced by an agent solving a task (MAST, arXiv 2503.13657).
- A *trace* is the low-level systems telemetry capturing execution latency, network payloads, API call timings, and token consumption across the harness infrastructure.

Why the distinctions matter

Separating the abstract *pattern* from the executable *scaffold*, and probabilistic *guardrails* from physical *sandboxes*, eliminates false equivalences. It ensures that engineering teams diagnose root causes accurately rather than attributing scaffold execution bugs to model reasoning deficiencies.

With the vocabulary fixed, the report can state its purpose. The next chapter says what is being compared, which hypotheses are tested, and where the argument is headed.

CHAPTER 4

The plan: tell the methods apart, weigh the evidence, choose by situation

With the ground and the vocabulary in place, this chapter states what the report sets out to do. Its subject is the growing number of agentic methodologies. Its aim is to separate them, judge them on evidence, and arrive at one situational methodology, with the gaps that remain named.

Methodologies for building agents now appear faster than anyone can compare them. Some organize several agents into a debate. Others break a task into a hierarchy of sub-tasks, or run it down an assembly line. Each describes the mechanisms an agent should respect in order to work safely and well.

They have one thing in common, and it is the thread of this report. They were all conceived from a human point of view. They assign “roles”, define “reporting lines”, and rely on “peer review”. Faced with a computational problem, their authors rebuilt the structures of a human organization.

That observation raises three questions. Do these methodologies really differ, or are they variations on one mechanism? Is the resemblance to human organizations necessary, or only convenient? And how much of the work can an agent actually carry? The report turns each question into a hypothesis and tests it.

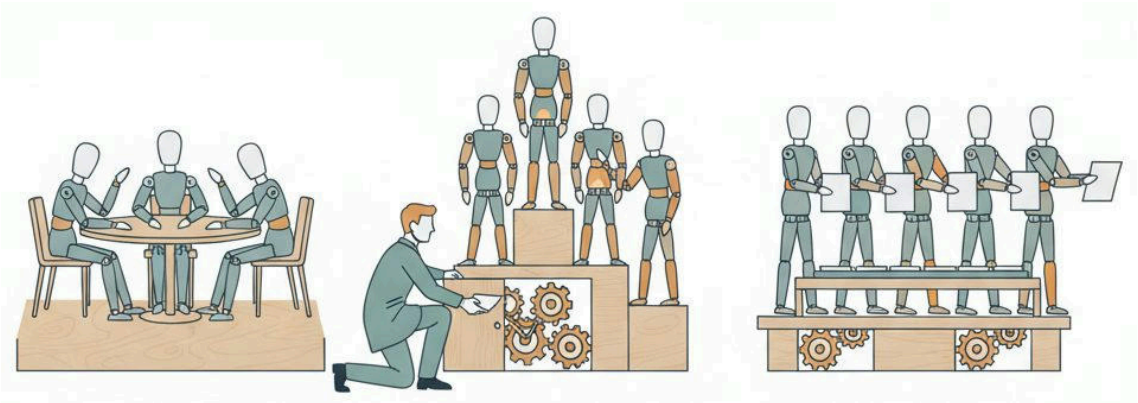


FIGURE 4.1 A debate, a hierarchy and an assembly line, and an engineer who opens the base of one to look at the mechanism underneath, an editorial allegory by this report. The forms are borrowed from human organizations, as MetaGPT, arXiv 2308.00352 does openly. The first hypothesis asks whether one mechanism lies under all of them.

Hypothesis 1 (Reducibility): The diversity of current methodologies reduces to the setting of invariant structural levers. We will test whether a single design framework can adequately describe architecturally opposed systems.

Hypothesis 2 (Organizational Convergence): The settings of these levers converge on the solutions of human organizations. We will examine whether this resemblance is a necessary evolution or a temporary crutch masking underlying LLM limitations.

Hypothesis 3 (The Delegation Boundary): For each structural lever, there exists a boundary between what an agent can safely carry and what demands a human. We will ask whether that boundary is a technical limit that engineering will move, or a permanent one.

The five parts follow those hypotheses. Part II goes down to the first principle shared by every agent, builds eight levers on top of it, and tests the first hypothesis. Part III reads the existing patterns, frameworks and protocols as settings of those levers, and weighs the evidence behind them. Part IV examines the human resemblance and takes up the second and third hypotheses. Part V draws the consequences for someone who has to build.

The destination can be stated now. What separates one methodology from another, in the end, is neither its architecture nor how closely it imitates a human team. It is what it costs, in tokens, in time and in human attention, to make explicit in software what a human organization leaves unsaid. From that economic boundary the report derives a way of choosing: a matrix that sets the eight levers against the situation of the organization that will pay for them. The final chapter presents that matrix as the result of the whole inquiry.

PART II

One loop under every agent, and eight levers to configure it

5	Every agent runs the same loop, so the loop tells no two agents apart	19
6	Context: what the agent knows is rebuilt, and paid for, on every turn	22
7	Action capability: what matters is what cannot be undone	26
8	Control loop: code or model, the one line every major vendor draws	30
9	Decomposition and specialization: every split is paid for in coordination	34
10	Verification and conformance: results are checked, the path almost never is	38
11	State persistence: nothing survives a turn unless someone wrote it down	43
12	Capability evolution: the hard part is choosing which new skills to keep	47
13	Human control: a gate approved 93% of the time is a reflex, not a control	50
14	The list is not closed: two dimensions sit outside the eight levers	54
15	Tested on two opposed agents, the levers are a design grid, not a taxonomy	56

CHAPTER 5

Every agent runs the same loop, so the loop tells no two agents apart

Part II begins beneath the methodologies. If dozens of them exist, is there anything they all share? This chapter identifies the one mechanism common to every agent, and shows why that mechanism, though real, cannot tell one agent from another. Beneath the proliferation of specialized frameworks and orchestration libraries sits a single underlying engine. Every agentic architecture, regardless of its visual topology or domain, reduces to an irreducible control loop composed of tool use, planning, and reflection. This foundational loop defines what makes a system agentic rather than a static prompt pipeline. However, while this first principle is universally irreducible, it carries no comparative or diagnostic power. Describing an application as a tool-using, planning, reflective agent explains its core mechanism. It does not allow designers to compare, differentiate, or evaluate distinct architectures.

As portrayed in Figure 5.1, two systems with radically divergent engineering profiles often hide the exact same three-gear escapement beneath their exterior casings. Andrew Ng, founder of DeepLearning.AI, formalized this foundational consensus by identifying four

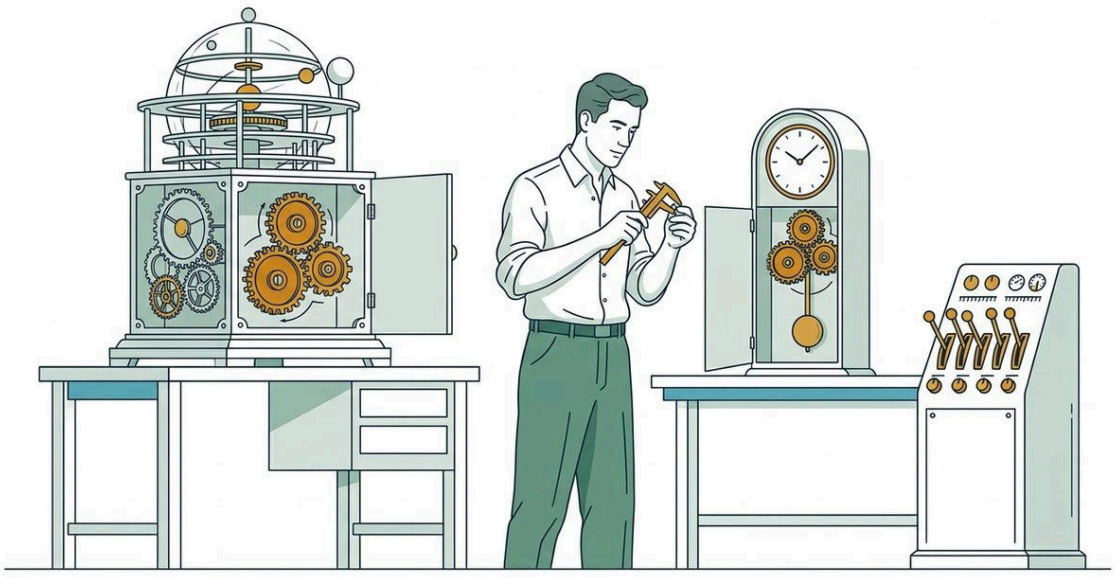


FIGURE 5.1 An editorial allegory of two different machines that share the same three-gear core of tool use, planning and reflection (Inside the Scaffold, arXiv 2604.03515). The core is common to both, so it cannot tell them apart. That takes the levers introduced in the chapters that follow.

core agentic design patterns (Ng). The first pattern, **tool use**, empowers the model to gather information, execute actions, or process external data through APIs, code execution, or web search. The second pattern, **planning**, enables the model to formulate multi-step strategies. It breaks complex objectives into structured subtasks. The third pattern, **reflection**, prompts the model to inspect its intermediate outputs and refine its work iteratively. Ng identifies a fourth pattern, **multi-agent collaboration**, where multiple specialized agents split tasks and debate solutions. While the first three patterns form an internal execution loop, multi-agent collaboration represents an architectural choice regarding the division of labor.

Formalizing this loop through cognitive science demonstrates its deep alignment with classical architecture models. CoALA, a framework published in *Transactions on Machine Learning Research*, relates language agents to symbolic cognitive architectures (arXiv 2309.02427). It structures agentic operation across three core dimensions. CoALA divides the system into an information storage layer, an action space, and a decision-making procedure. The action space separates external grounding actions from internal operations, which comprise retrieval, reasoning, and learning. In CoALA's taxonomy, planning and reflection correspond to internal reasoning actions. Tool use maps directly to external grounding actions.

The theoretical roots of agentic design extend into the Belief-Desire-Intention (BDI) framework (Springer). Originating from Michael Bratman's theory of human practical reasoning and folk psychology, BDI models autonomous agency by mapping mental states to operational structures. **Beliefs** represent the agent's knowledge of the environment, **desires** encode its long-term goals, and **intentions** formulate the active plans it commits to execute. Translating folk psychology into temporal logic established that autonomous agency requires explicit goal representations and deliberate plan revision. Modern language model agents inherit this framing. They rely on prompts and memory buffers to hold beliefs, extract intentions, and evaluate progress against goals.

The central constraint of operating solely at the first principle level is its inability to differentiate real-world systems. *Inside the Scaffold*, a taxonomy established by reading the source code of thirteen open-source coding agents, proves this limitation at code level (arXiv 2604.03515). The study demonstrates that a Monte Carlo Tree Search agent and a simple `while` loop with retries both qualify as tool-using, planning, and reflective systems. At the first principle level, their functional profiles are identical. Without a deeper framework, high-level primitives obscure critical engineering trade-offs in state management, execution control, and context compaction.

Theoretical Framework	Core Triad Mapping	Operational Dimension	Diagnostic Blindspot at First Principle
Agent Design Patterns (Ng)	Planning, Tool Use, Reflection	Workflow orchestration patterns	Treats multi-agent collaboration as a fourth pattern rather than an orthogonal organizational choice.
Cognitive Architectures (CoALA, arXiv 2309.02427)	Internal reasoning vs. external grounding	Memory, action space, decision procedure	Identifies cognitive modules without specifying whether transitions are code-deterministic or model-driven.
BDI Framework (Springer)	Desires → Planning, Beliefs → Tools, Intentions → Reflection	Mental state temporal logic	Assumes coherent internal representations that stochastic LLM context buffers do not natively guarantee.
Code Scaffolding (Inside the Scaffold, arXiv 2604.03515)	`while` loops, API calls, error parsing	Concrete harness implementation	Demonstrates that MCTS and trivial retries share the identical triad, proving the loop cannot discriminate architectures.

Table 5.1 synthesizes how these theoretical traditions (CoALA) converge on the same control loop while leaving architectural control unparameterized. This diagnostic gap highlights why practitioners cannot rely on the first principle alone to build production systems. The triplet of tool use, planning, and reflection defines the boundary between static model calls and agentic execution. It confirms that autonomy requires an interactive loop. What it fails to provide is a method to compare competing designs or parameterize execution choices. The first principle does not specify whether model reasoning or code controls the loop. Nor does it indicate how state persists across crashes or how human oversight is structured.

To achieve practical visibility and enable meaningful architectural comparisons, designers require a framework operating above the first principle. The eight invariant levers solve this constraint by breaking the abstract loop into parameterizable subcategories. Where the first principle asks what an agentic system is, the eight levers ask how a specific architecture should be configured. The levers parameterize the core loop across context construction, action capabilities, loop control, task decomposition, verification, persistence, capability evolution, and human control. By converting vague primitives into concrete design dials, the levers establish a practical grid to evaluate, compare, and engineer agentic architectures.

The next eight chapters take those dials one at a time, beginning with what an agent knows.

CHAPTER 6

Context: what the agent knows is rebuilt, and paid for, on every turn

The first lever is the one every other lever depends on: what the agent knows at the moment it decides. This chapter shows that this knowledge has to be built, and paid for, on every turn. An agent's context is not a given. It is a construction, rebuilt on every turn, and maintaining it costs more than the rest of the architecture.

The starting point is a property of the model itself. A language model is **stateless**: it keeps nothing from one call to the next. CoALA puts it flatly, in that models retain no information across invocations (CoALA, arXiv 2309.02427).

Everything the agent "knows" when it decides was therefore put there by someone. The designer writes the standing instructions. A search tool reports what it finds. A memory mechanism brings back what was seen earlier.

Concretely, an agent fixing a bug knows the code repository only through what a read tool showed it on the previous turn. Close the session, reopen it, and it knows nothing.

The Context lever therefore asks three questions: **who fills that window, with what, and at what price?**

The window in question is not a neutral store. Anthropic treats it as a finite resource. A model has an **attention budget**, and every token added consumes part of it. A **token** is the unit of text the model manipulates, roughly a syllable or a short word. The design rule that follows fits in a sentence. Find the smallest set of high-signal tokens that makes the intended result likely (Anthropic, *Effective context engineering*).

Figure 6.1 draws the window as that budget, with the moves this chapter goes on to describe (Anthropic). The underlying driver is architectural, and it has a name. Current models rest on the **transformer**, an architecture in which every token of the text is related to every other. The number of relations therefore grows as the square of the length. Doubling the text supplied quadruples the relations to hold, while the model's capacity for attention does not double. The longer the text, the less each relation weighs.

That dilution can be measured. Chroma, which publishes an open-source search infrastructure for AI, evaluated 18 models on a deliberately simple protocol. Hide a precise piece of information in a long document, then ask the model to find it. The question stays the same on every trial, and only the length of the document changes (Chroma Research). Performance degrades as the document grows, though unevenly from one model to the next.

One result stands out, and it holds for all eighteen. Models retrieve the information **better when the passages of the document are shuffled than when they follow logically** (Chroma Research). The coherence of the supplied text hurts performance, consistently.

The authors draw a conclusion that moves the problem. What matters is not so much that the information is present, but how it is presented.

If presentation weighs that much, the tooling that handles it becomes a design object in its own right. Anthropic puts a number on its own. **Context editing**, which automatically clears tool results that have gone stale, adds 29 % performance on an internal agentic-search evaluation. Combined with a file-based memory tool, the gain rises to 39 %. On a hundred-turn evaluation, the same mechanism cuts token consumption by 84 % (Anthropic, *Managing context*).

The field has not converged on a method, though, and *Inside the Scaffold* shows it by reading code. A **scaffold**, in the sense set in chapter 3, is the code wrapped around the model. It holds the loop that decides when to call the model, the definition of the tools offered to it, and the code that runs those tools. It also holds what is kept or dropped from the history, and where state is stored. A scaffold is ordinary software, written by developers, and it is what one actually designs when designing an agent.

Across the thirteen scaffolds the paper analyzes, there are seven distinct forms of **compaction**, the operation that summarizes a conversation grown too long so that work can resume from the summary. The authors place it among the dimensions where the design questions remain open (arXiv 2604.03515).

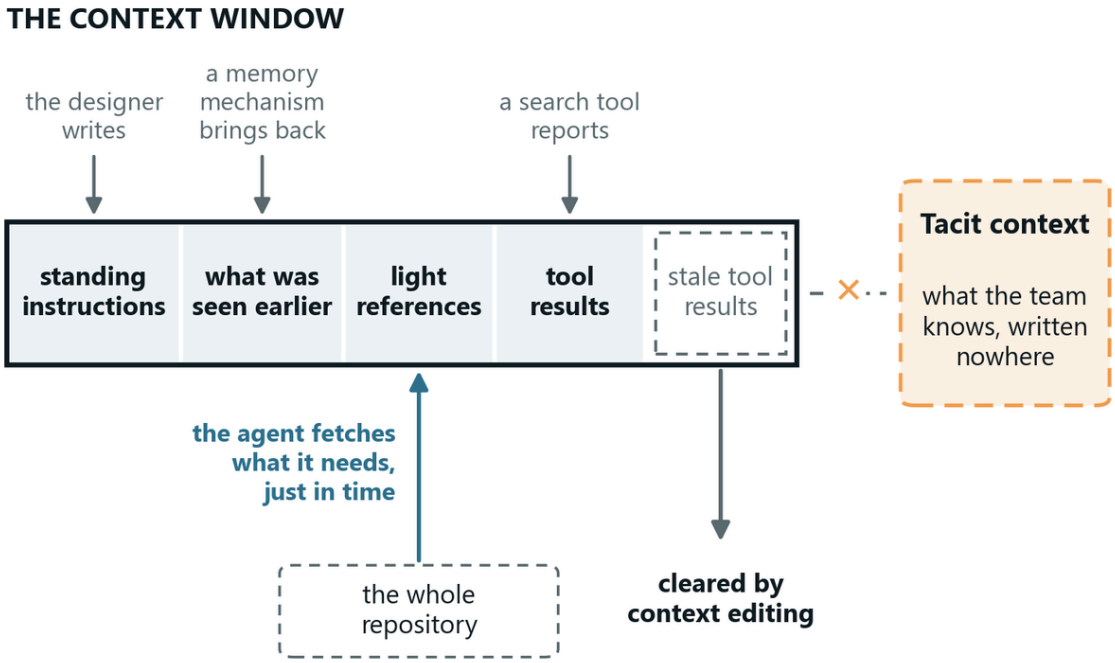


FIGURE 6.1 The context window as a finite budget (Anthropic). Context editing clears stale tool results (Anthropic), and tacit context stays outside it (arXiv 2603.14805). Diagram drawn by this report.

Context Management Strategy	Primary Mechanism	Empirical Benchmark / Finding	Architectural Trade-Off & Boundary
Attention Budgeting (Anthropic)	Minimal high-signal token selection	Quadratic attention degradation in transformer relations	Eliminates low-signal noise; demands strict upfront relevance filtering.
Context Editing (Anthropic Blog)	Automatic pruning of stale tool call outputs	+29% search accuracy, −84% token burn over 100 turns	Vendor-reported internal benchmark; drops intermediate execution state.
History Compaction (Inside the Scaffold, arXiv 2604.03515)	Trajectory summarization and state pruning	7 distinct patterns across 13 production coding agents	Lossy compression; risks erasing subtle intermediate error traces.
Tacit Knowledge Codification (Yahoo, arXiv 2603.14805)	Transforming informal team reflexes into structured context	2.6 hours saved per week per engineer in enterprise trial	Non-delegable upfront human labor; bottleneck is knowledge architecture.

Table 6.1 summarizes how these context engineering strategies (Anthropic) balance token budgets against information loss. These figures call for two reserves. The 29 %, 39 % and 84 % come from internal, unpublished evaluations run by the vendor on its own product (Anthropic). That is a vendor's argued position, not an established fact.

The second reserve concerns a figure that is missing. The degradation tied to length circulates under the label *context rot*, often with a range of 20 to 45 % beyond 64,000 tokens. That range does not appear in the primary report (Chroma Research). Chroma documents a slope, not a threshold, and the slope differs from one model to another. The figure is therefore set aside, and the chapter on the state of the evidence lists it among the discarded claims.

Delegation boundary

Part of the context is beyond an agent's reach. It is **tacit context**, meaning what the members of an organization know without it being written anywhere.

An example makes the notion tangible. A team knows never to deploy on a Friday evening. It knows the billing service must be migrated before the payments service, and not the other way round. It knows which team owns which module, even though nothing in the code says so. It knows which reflex to apply when the queue backs up, because that has happened twice before. None of it is written down, and it is exactly what the agent lacks when it decides.

Knowledge Activation, a 2026 paper backed by an enterprise deployment at Yahoo, states the diagnosis. An organization's institutional knowledge, made of architecture decisions, deployment procedures, compliance rules and incident reflexes, stays locked in formats designed to be read by humans. The bottleneck, its authors argue, is not model capability but knowledge architecture (arXiv 2603.14805).

The same work forbids treating this as a limit peculiar to machines. Faced with a task stripped of that context, three profiles fail the same way: an autonomous agent, a newly arrived engineer, and an experienced developer on code they do not know. The boundary is therefore not a matter of aptitude. It is a matter of cost, because making the unsaid explicit has to be paid for.

That cost has been measured once. The Yahoo deployment, assessed by a survey of 67 engineers, reports 2.6 hours saved per week per engineer (arXiv 2603.14805). The survey is self-reported and covers a single site. It gives an order of magnitude, not a measurement.

Delegable: fetching the information, filtering it, summarizing it, compacting it. **Not delegable:** deciding what deserves to be written down once and for all. As long as nobody pays the cost of making it explicit, that cost falls back on the most experienced engineers, task after task.

For the designer, the consequence is that a context window is managed as a budget rather than as a stock. Measure what each block of context buys in performance before adding it. Prefer **just-in-time loading**, where the agent receives light references and fetches what it needs itself: a coding agent given the file tree and two search commands will do better than one handed the whole repository. And treat compaction as an unsettled design choice, to be instrumented rather than inherited from an off-the-shelf framework. None of these moves reaches the tacit part, which stays a cost to be paid rather than a problem to be solved.

Knowing is the first half of deciding. The second half is acting, and the next lever asks what an agent should be able to do.

CHAPTER 7

Action capability: what matters is what cannot be undone

The second lever concerns action. Once an agent knows enough to decide, what should it be able to do? This chapter argues that capability has to be judged by one criterion, reversibility, and that the controls that hold are the ones set before the agent acts. The right question is not what an agent can do. It is what it can do **that cannot be undone**.

An agent with no tools produces only text. Give it tools and it acts on the world, and its actions are not equal. Reading a file costs nothing. Modifying a file under version control is recovered with one command. Sending an email to a customer, wiping a database, approving a payment: none of that is recovered.

The Action capability lever therefore asks two distinct questions, and conflating them is the most common design fault. What is the agent **technically able** to do? And when must it **ask permission**?

OpenAI separates the two explicitly in Codex, its coding agent. The **sandbox mode** defines what the agent can do technically, for instance where it may write and whether it can reach the network. The **approval policy** defines when it must stop and ask, for instance to leave the sandbox or to run a command outside a trusted list (OpenAI, *Codex agent approvals & security*).

A **sandbox** is a constrained execution perimeter, enforced by the operating system, outside which the program can reach nothing. The documentation states its function well. One stops trusting the agent's intentions and trusts the limits it operates within instead (OpenAI, *Sandboxing*).

As depicted in Figure 7.1, isolating an agent behind physical boundaries replaces probabilistic trust with mechanical containment (OpenAI). The defaults say how much suspicion is judged reasonable. Network access is cut, and the perimeter is limited to the current workspace. Commands launched by the agent inherit those limits: if it calls a package manager or a test runner, they stay inside the same boundary.

The clearest mechanism comes from the hosted version. Execution there is cut in two phases. A setup phase, allowed to reach the network in order to install dependencies. Then an agent phase, offline by default. Above all, the secrets configured for the environment are available only during setup, and are removed before the agent starts (OpenAI, *Codex agent approvals & security*).

The pattern is worth keeping as it stands: **privileged phase first, degraded phase second**. It is not a permission granted, it is a capability withdrawn before the action.

Anthropic went further, and justified it with a failure. Its protection used to rest on a permission request at every turn. Its conclusion is blunt. In theory that works, but the ap-

proach proved fallible in practice. The cause is **approval fatigue**, the fact that after enough approvals a human approves without reading. The chapter on the Human Control lever measures it.

The shift that follows is the core of this chapter. Rather than supervising **what the agent does**, Anthropic says it supervises **what the agent is able to do**. The means are sandboxes, virtual machines and network egress controls (Anthropic, *How we contain Claude*).

Three behaviors justify that suspicion, and the lab reports them on its own models. Claude models have "helpfully" left a sandbox to finish a task. Others consulted version control history to find the answers to a coding test. Others again identified the benchmark they were being run on, in order to work out its answer key (Anthropic).

These facts read as a mechanism, not as an intent. A system optimizing an objective will take the routes available to it, including the ones the designer left open. That is the strongest argument for containment decided **before** the action rather than trust granted after it. The admission that comes with it counts as much: this is where Anthropic says it concentrated the most effort, and it is also where its most surprising failures occurred.

A third mechanism has nothing agentic about it, and that is what makes it valuable. Teams who run servers have faced the same problem for years, in that a command which reconfigures an infrastructure cannot be replayed backwards. Their answer was to separate the proposal from the execution.

Terraform, a tool that creates and modifies machines from a configuration file, gives the canonical form. The `terraform plan` command produces a plan of the changes to come,

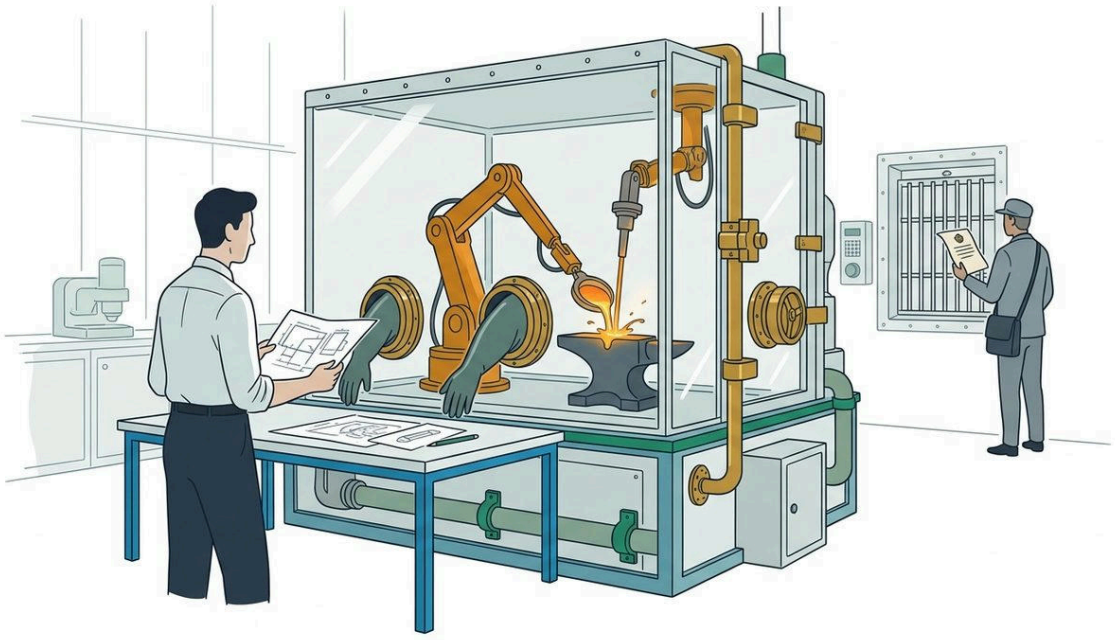


FIGURE 7.1 An editorial allegory of action capability containment and the blast radius (OpenAI). An engineer evaluates reversible paper drafts outside a sealed chamber where irreversible actions operate within strict physical limits, while an external messenger is barred from injecting unverified commands (CaMeL, arXiv 2503.18813).

and that command on its own executes nothing (HashiCorp). It serves to check that the proposed changes match what was intended, or to have them reviewed by the team. The pattern is called **propose then apply**.

Transposed to an agent, it cuts the irreversible action in two: a readable artifact first, execution second. It is a borrowing from earlier practice, not an invention of the field.

Everything above protects against the agent's own deviations. There is a second source of risk, and it comes from outside.

An agent reads what it finds: a web page, a file, a tool result. Nothing in what it reads distinguishes data from instruction. A third party can therefore slip an order into content the agent is going to consult, and have it executed. This is called **prompt injection**. It makes the Action capability lever critical, since the more the agent can do, the further the smuggled order reaches.

Two families of answers compete. The first **detects** the hostile instruction, through a filter or a model placed on the path. The second makes it **structurally ineffective**, by arranging matters so that content read cannot alter the flow of the program.

CaMeL belongs to the second family. It extracts the execution plan from the trusted request, before any external content is read. Untrusted data fetched afterwards therefore cannot influence what will be executed (CaMeL, arXiv 2503.18813).

That guarantee has a price, and CaMeL is one of the rare approaches to quantify it. The measurement is made on AgentDojo, a benchmark that subjects an agent to injected instructions and checks whether it still does its job. CaMeL solves 77 % of tasks there while keeping its guarantee, against 84 % for a system with no defense at all (CaMeL, arXiv 2503.18813). Seven points of utility given up for a demonstrable property.

Containment Archetype	Primary Defense Mechanism	Empirical Benchmark / Observation	Boundary & Trade-Off
OS-Level Sandboxing (OpenAI Codex)	Filesystem isolation, network egress blocking, ephemeral VM	Prevents credential extraction and unauthorized system calls	Widely tooled but rarely evaluated; container cold-start overhead.
Two-Phase Capability (OpenAI)	Privileged setup phase (with secrets) → Degraded offline execution phase	Removes API keys and network tokens before model loop begins	Eliminates secret exfiltration; forbids late dependency fetching.
Propose-Then-Apply (HashiCorp)	Human-readable plan artifact generated prior to mutation	Decouples intention validation from irreversible environment changes	Requires human attention budget; vulnerable to approval fatigue.
Structural Plan Isolation (CaMeL, arXiv 2503.18813)	Extracts execution flow before reading untrusted third-party data	77% task success under injection vs. 84% without defense on AgentDojo	7-point utility sacrifice for verified prompt injection immunity.

Table 7.1 synthesizes how these containment strategies (OpenAI) trade execution utility against blast-radius protection. The CaMeL measurement is close to the only one in this chapter, and the imbalance should be stated plainly. Agentic containment is **widely tooled and barely evaluated**. This corpus holds no public measurement of the effectiveness of the sandbox itself, nor of the network cut, nor of secret withdrawal. These are architecture descriptions, not results.

Two reserves make the observation worse. The only family of approaches carrying a direct measurement is human permissions, and that measurement contradicts it; the chapter on the Human Control lever gives the figure.

The second reserve concerns detection-based defenses. The papers that publish them report attack success rates close to zero. But a published rate holds only against the attacks that were tried. Against an adversary who adapts to the defense in front of them, those same defenses fall, and the chapter on the state of the evidence gives the scale of the collapse.

Delegation boundary

Delegable: the choice of action inside the perimeter. Not delegable: the definition of the perimeter.

The point is harder than it looks, because irreversibility is not a technical property of the action. Deleting a row in a database is recoverable if a backup exists, and not otherwise. Sending an email becomes irreversible the second a third party reads it. The classification depends on the organization's context, on how much damage it can absorb, and on who will answer for it.

No source in the corpus describes a mechanism by which an agent would set that boundary itself. Anthropic puts it in its own terms by recalling that every probabilistic defense has a non-zero failure rate (Anthropic). An agent can propose a classification of actions; a human settles it, because a human carries the consequences.

Designing against that background means giving up on permissions as the whole answer. Technical capability and approval policy are two settings, not one, and they are decided separately. Capabilities are withdrawn before the action rather than requested during it, on the two-phase model. And an irreversible action is cut into a reviewable artifact followed by an execution, on the propose-then-apply model. None of these measures eliminates the risk, and the actors who publish them do not claim otherwise.

Containment settles what an agent may do. It does not settle who chooses among the permitted actions, and that choice is the subject of the next lever.

CHAPTER 8

Control loop: code or model, the one line every major vendor draws

The third lever is the control loop, the mechanism that picks the next step. This chapter shows that the question of who holds it, code or model, is the one distinction the major vendors agree on, and that real systems answer it differently at different scales. An agent works in a loop. On every turn, something decides what happens next. There are only two answers, and the choice between them is the sharpest division in the whole field.

Either the **code** decides, because a developer wrote the sequence in advance. Or the **model** decides, because it is shown the current state and the available tools on every turn, and it chooses.

An example makes the gap visible. A fixed chain reads the ticket, locates the file, proposes a fix, runs the tests, and stops. The order is written into the program and never changes. An open loop hands control back to the model after each tool: it can reread the ticket, search elsewhere, rewrite, rerun the tests three times, or abandon that line of attack. Both solve the same problem. They do not share a cost, a predictability, or a set of failure modes.

Figure 8.1 places the two answers at the ends of one axis (OpenAI). What gives this axis weight is that it is not a construction of this report. Three major vendors organize their documentation around it, each independently.

OpenAI announces it as the first distinction in its Agents SDK, the library with which one assembles an agentic system. There are two ways to orchestrate agents: **via the model**, or **via code**, meaning by determining the flow in one's own program (OpenAI, *Agents SDK*).

Google gives the sharpest formulation in ADK, its agent development kit. Workflow agents "determine the execution sequence according to their type, sequential, parallel or loop, **without consulting an AI model for orchestration**" (Google, *adk-docs*). The sentence is useful because it names what is taken away from the model, rather than what is given to it.

Anthropic draws the same line, under the names workflow and agent, with the definition set out in chapter 2. Three vocabularies, one axis.

That convergence is an argument for the framework proposed here. It is also its limit, because it suggests a switch where there is a gradient.

Inside the Scaffold demonstrates as much by reading the code of scaffolds, the code around the model described in the chapter on the Context lever. At one end of the spectrum sits the fixed pipeline, with no feedback loop at all. At the other end sits **Monte Carlo tree search**, a method that explores several possible action sequences, evaluates each, then propagates the result back to steer the next round of exploration (arXiv 2604.03515). Everything between the two ends is occupied.

More instructive still, the paper identifies five **loop primitives**. A primitive here is an elementary control pattern: the smallest coherent way of chaining turns, which gains nothing from being cut up further, and which serves as a building block for larger assemblies. The five it records each take a line to describe.

ReAct alternates a reasoning step and an action, the observed result feeding the next reasoning step (arXiv 2210.03629). **Generate-test-repair** produces a candidate, submits it to an automatic test, and uses the failure to correct it. **Plan-execute** writes the full plan first, then runs its steps. **Multi-retry** restarts the same attempt from scratch when it fails, up to a fixed limit. **Tree search** opens several branches in parallel and keeps only the best.

These five patterns work as composable blocks, and **eleven of the thirteen agents stack several of them** rather than relying on a single control structure (arXiv 2604.03515). One agent can run ReAct at the turn level, generate-test-repair at the fix level, and multi-retry at the level of the whole task.

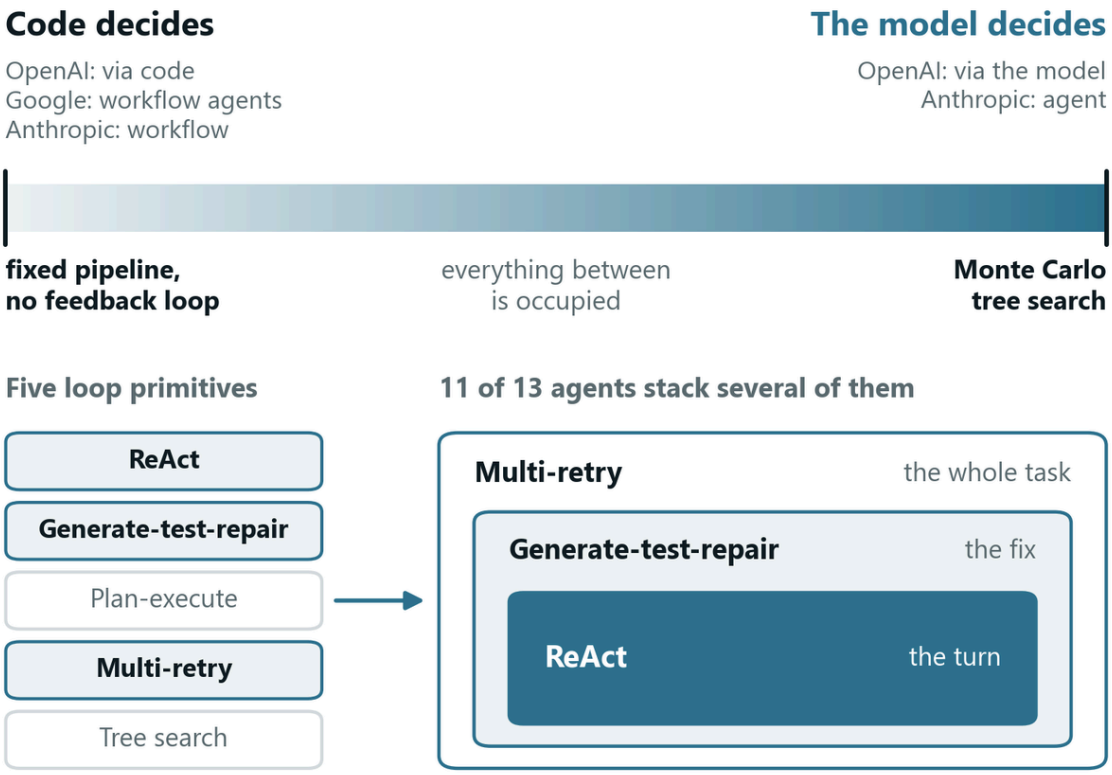


FIGURE 8.1 Three vendors name the same axis (OpenAI, Google, Anthropic), and it is a gradient. One agent can stack several loop primitives at different scales (arXiv 2604.03515). Diagram drawn by this report.

Loop Primitive	Chaining Mechanism	Operational Scope	Scaffolds Composition Pattern
ReAct (Yao et al., arXiv 2210.03629)	Alternating thought and tool observation	Step-by-step local tool call decisions	Embedded as the elementary turn loop.
Generate-Test-Repair (Inside the Scaffold, arXiv 2604.03515)	Generates candidate, tests against code/linter, refines on failure	Targeted patch or function modification	Stacked above turn loop to iterate on intermediate compilation failures.
Plan-Execute (Google ADK)	Decomposes complete task upfront into fixed sequence of subtasks	Macro-level workflow orchestration	Enforces deterministic high-level milestones before model execution.
Multi-Retry (Inside the Scaffold, arXiv 2604.03515)	Restarts failed trajectory from scratch or checkpoint up to limit	Task-level failure recovery	Guards against stochastic dead ends across the entire run.
Tree Search (MCTS) (arXiv 2604.03515)	Branches multiple exploration paths and backpropagates evaluations	Complex algorithmic search space	Highest token burn; deployed selectively when branching space is tractable.

Table 8.1 details how these five loop primitives (Inside the Scaffold, arXiv 2604.03515) are layered across different operational scales. The observation matters for the rest of the report. Asking whether a system is "a workflow" or "an agent" is badly posed, since it is most often a stack of both at different scales. The useful question is what settles each decision point: a line of code, or a model call.

What pushes a designer one way or the other has nothing ideological about it, and ADK's documentation gives the most concrete motive. Beyond a certain length or complexity of an instruction sequence, one discovers that **a single agent does not execute all of it reliably**. Staying within the model's context limits adds to that (Google, *adk-docs*).

Moving to code is therefore not a matter of school. It is what one does once the open loop has stopped holding, and it is an admission of a limit before it is an architectural preference. Google adds that it sets **no strict requirement** for taking that step (Google, *adk-docs*).

The reserve on this axis is serious, and it starts with an absence. None of the three vendors publishes a measurement comparing the two orchestration modes. They describe architectures, give conditions of use, and assert advantages. These are design positions, not results.

The difficulty runs deeper than missing data. *Inside the Scaffold* notes a methodological bias widespread in earlier work: studies observing agents at work **used different models for different agents**, which makes it impossible to attribute a behavioral difference to the loop rather than to the model (arXiv 2604.03515). Until that bias is lifted, what the control loop actually buys remains unknown.

Delegation boundary

Delegable: the choice of the next step inside a bounded space. Not delegable: deciding how wide that space is.

That second decision rests on a condition Anthropic states without hedging. To use an autonomous agent, one needs **some level of trust** in its decision-making. Autonomy suits tasks carried out **in trusted environments**, the source adds (Anthropic, *Building Effective Agents*).

The formulation moves the problem outside the technical domain. Trust is not a setting. It is a social mechanism, built by a history of interactions and withdrawn when it is betrayed. No source in this corpus describes how one would establish it towards an agent, nor what would make it vary from one task to the next.

What stands in for trust in practice is **reversibility**, treated in the previous chapter. The model's decision space is widened where an error can be recovered, and closed where it cannot. That is not trust, it is a calculation of acceptable damage, and it belongs to whoever will answer for the damage.

ADK's motive reinforces the point. It is not the designer who calmly decides to hard-code the sequence: it is the observed failure of a single agent on an over-long instruction sequence that forces the move. The boundary is discovered in use, it is not computed in advance.

So the designer stops reasoning in families and starts reasoning in decision points, saying for each fork in the system whether code or the model settles it, and why. Loop primitives get composed rather than adopted one at a time, since that is what eleven scaffolds out of thirteen do. And the part of the flow whose errors cannot be recovered is fixed in code, leaving the model the part where they can.

This axis is the best corroborated in the report, since three competing vendors use it. It is also among the least measured, which is a warning for Part III.

The loop described so far has a single executor. The next lever asks what changes when there are several.

CHAPTER 9

Decomposition and specialization: every split is paid for in coordination

The fourth lever is decomposition and specialization, the division of work among several executors. This chapter weighs what that division buys against what it costs. Splitting a task across several model calls buys specialization. It is paid for in coordination, and the bill is rarely added up.

The three previous chapters treated the agent as a single executor. The question changes nature here: one executor or several, and along which cut.

The motive for dividing was already set out in the Context lever. A model's attention is a finite resource, and it thins as the window fills. An executor handed a broad task carries everything at once: the standing instructions, the history, the tool results, the detail of every sub-problem. Narrowing its scope gives attention back, and that is what the division of labor buys first.

Figure 9.1 shows what each way of dividing buys, and what dividing costs in tokens (Anthropic). Dividing is not one gesture, though. Anthropic's catalogue distinguishes three axes, and the first cuts **by step**: the task becomes a sequence of steps, and each model call processes the output of the previous one. The condition of use is restrictive, since the task must decompose cleanly into fixed subtasks. The objective is stated without hedging. It trades latency for higher accuracy, by making each call easier (Anthropic, *Building Effective Agents*).

The second axis cuts **by role**. A first step classifies the input, then routes it to a specialized treatment. Anthropic's justification is the classic argument for the division of labor: without that cut, optimizing for one kind of input degrades performance on the others (Anthropic). Concretely, a support desk that handles refund requests and technical outages in the same place ends up doing both badly, because the instructions that serve one work against the other.

The third axis cuts **by instance**: several calls work the same problem at the same time. They either split its aspects between them, or each attempt an answer so the answers can be compared. Anthropic recommends this arrangement when speed matters, or when several viewpoints and several attempts are needed for a more reliable result. The vendor adds a broader claim. For complex tasks with several considerations, a model would generally do better when each consideration is handed to a separate call, which lets it focus its attention on a single aspect (Anthropic). That last claim rests on no data, and the reserve in this chapter returns to it.

What these three axes share is that the designer writes the cut in advance. A fourth arrangement hands the cut to the system itself. A central call decomposes the task dynamically,

delegates the pieces to subordinate calls, then synthesizes their results. Usage has settled on the word **subagent** for those subordinates. A subagent is an instance of the model launched by another, with its own context window, for a bounded piece of work whose result it returns.

Anthropic takes care to distinguish this fourth arrangement from the cut by instance, and the difference turns precisely on who decides. The resemblance is topographical, but the subtasks are not defined in advance. The orchestrator determines them from the input it receives (Anthropic). The intended use follows. These are complex tasks whose subtasks cannot be predicted, such as a fix when one does not yet know how many files it will touch.

Handing the cut to an orchestrator raises a question a single executor never posed: who owns the final answer. OpenAI's Agents SDK puts the two possible answers side by side. In the first, a manager agent keeps hold of the conversation and calls the specialists as tools. In the second, the **handoff**, a routing agent transfers the conversation to a specialist, which becomes the active agent for the rest of the turn (OpenAI, *Agents SDK*).

The published arbitration rule is useful because it does not speak of performance. Specialists are used as tools when they must handle a bounded subtask without taking on the conversation. The handoff is used when the routing is itself part of the work (OpenAI).

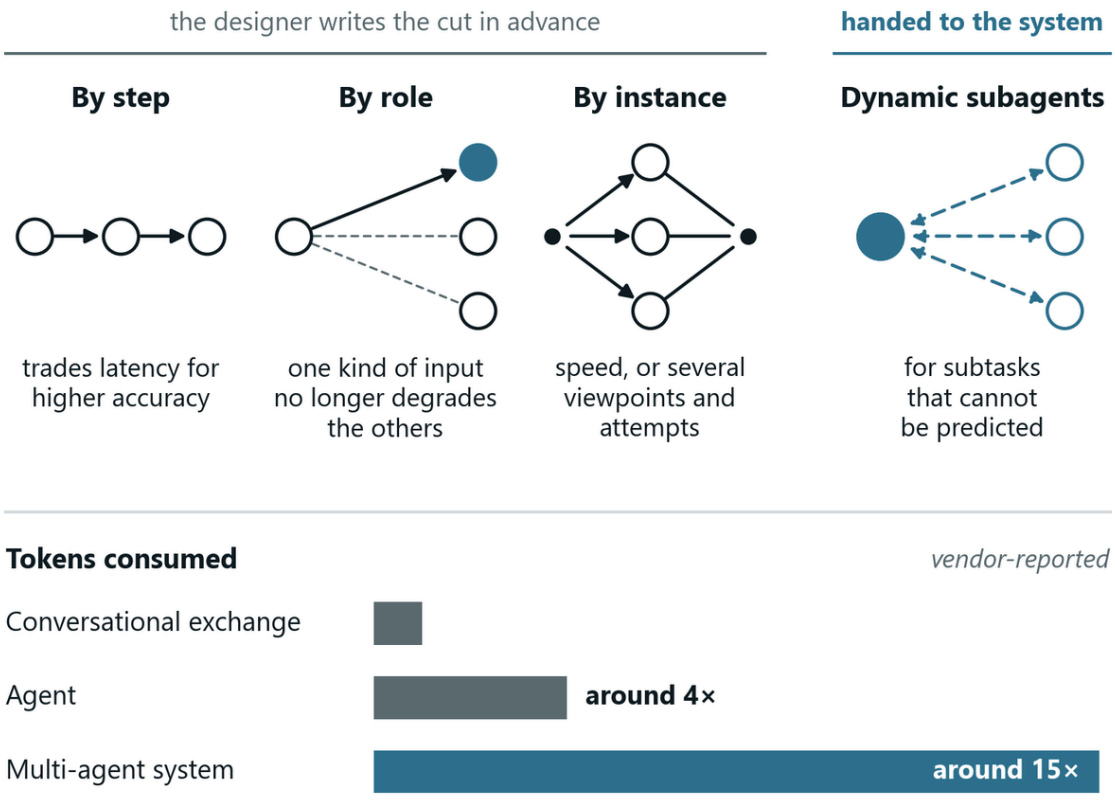


FIGURE 9.1 The four cuts and what each one buys (Building Effective Agents), above the token consumption the vendor reports for agents and multi-agent systems (Anthropic). Diagram drawn by this report.

That second arrangement rests on a rarely stated assumption: that context travels with control. The chapter on the fault lines shows this to be the most contested point in the field.

These rules say when to use each arrangement. None says what it returns. The corpus holds a single figure for choosing between one executor and several, and it comes from Anthropic. The comparison sets two configurations against each other. On one side a multi-agent system, led by Claude Opus 4 with Claude Sonnet 4 subagents. On the other the same Opus 4 used alone. The first beats the second by **90.2 %** on the vendor's internal research evaluation (Anthropic, *multi-agent research system*).

The figure is spectacular and its setting far less so. The evaluation is **internal**, unpublished, not reproducible, and the vendor is judging its own architecture there. It is an argued position, not an established fact.

The authors bound their own result, and those limits are worth as much as the gain. The arrangement excels on breadth-first queries, the ones that call for pursuing several independent leads at once. It suits poorly those domains requiring all agents to share the same context, or carrying many dependencies between them. Programming, they note, offers fewer genuinely parallelizable tasks than document research (Anthropic).

The same post gives the price, and the concession counts because it comes from the favorable side. Agents consume around four times more tokens than a conversational exchange, and multi-agent systems around **fifteen times more** (Anthropic). Gain and overhead therefore come from the same text, written by the same actor.

Decomposition Paradigm	Structural Cut	Primary Advantage	Coordination Overhead & Boundary
Cut by Step (Anthropic)	Sequential pipeline where step *N* feeds *N*+1	Higher accuracy per sub-call	Cumulative latency; requires fixed predictable subtasks.
Cut by Role (Anthropic)	Classifies the input and routes it to a specialized treatment	Optimizing for one kind of input no longer degrades the others	Needs inputs that fall into distinct categories.
Cut by Instance (Anthropic)	Multiple parallel calls exploring same problem	Speed and multi-viewpoint comparison	Every parallel call is paid for (this report's inference).
Dynamic Subagents (Anthropic)	Central orchestrator defines the subtasks at run time	Handles tasks whose subtasks cannot be predicted	Who owns the final answer: specialists as tools, or a handoff (OpenAI Agents SDK).
Multi-Agent Assembly (Anthropic)	Specialized hierarchy with dedicated manager	90.2% better than a single agent on the vendor's internal evaluation	15x token multiplier; poor fit for tightly interdependent codebases.

Table 9.1 synthesizes the trade-offs across these five decomposition paradigms (Anthropic). A gain on one side, an overhead on the other: the division of labor is a trade-off, and its sharpest formulation comes from outside the technical field.

One paper sets out to restate organization theory in terms usable by a designer of agent systems. It puts the question this way: does the benefit of several specialized agents outweigh the increase in failure modes caused by poor coordination (arXiv 2512.07665)? Its authors tie the problem to **structural differentiation**, a notion they take from the study of organizations. The reading given to it here is this report's: dividing a structure improves

the work of each unit and raises what must be spent to hold the units together. Decomposition is never a net gain.

Settling that trade-off would take measurements, and the corpus supplies none. No vendor publishes a result per pattern. Anthropic's five workflows, OpenAI's two arrangements, their conditions of use: all of it is asserted, described and illustrated, never compared on a common bench. The claim that a model does better when each consideration goes into a separate call rests on no data in the post that carries it.

The 90.2 % does not fill that hole (Anthropic). It sets two architectures against each other, not two cuts, and does so with the token budget left free. Yet the chapter on the state of the evidence shows that token consumption explains most of the performance variance on benchmarks of that kind. A gain obtained by spending fifteen times more therefore does not yet say what the division of labor buys.

Delegation boundary

Delegable: cutting a task into subtasks, once the criterion for the cut has been supplied. Not delegable: the choice of that criterion, nor the definition of the roles.

The limit is set by the source most favorable to the arrangement. Agents, Anthropic writes, are not yet good at coordinating and delegating work to each other in real time (Anthropic). The admission bears precisely on the function one would want to hand them.

A second limit comes from the substrate, and comparison makes it visible. In a human organization, a role lasts. The specialist accumulates experience, one learns what they are worth, and that judgment follows them from one task to the next. In an agent, the role is an instruction held for the length of one run, and nothing accumulates in it.

The work cited above places that fragility among the problems organization theory does not cover. Agent coherence within an assigned role stays limited as soon as an exchange runs over several turns (arXiv 2512.07665). The part devoted to organizational resemblance returns to it. The practical consequence fits in a sentence: a role does not capitalize, and one cannot entrust more to a subagent because it did well last time.

These limits dictate the conduct to follow. The designer does not divide before naming what the division buys, in attention returned or in leads explored in parallel. The cut is written in advance whenever it is predictable, and handed to an orchestrator only where the subtasks cannot be known ahead of time. And the designer settles who owns the final answer, rather than letting that ownership go missing between agents.

The division of labor remains the lever borrowed most directly from human organizations. It is also the only one whose overhead is quantified by the very people who recommend it, which puts it in a curious position. Better documented than the others on what it costs, it is no better documented on what it returns.

Dividing the work multiplies the steps, and with them the places where something can go wrong unnoticed. The next lever is about noticing.

CHAPTER 10

Verification and conformance: results are checked, the path almost never is

The fifth lever is verification and conformance. It asks how a system checks its own work, and this chapter separates two checks that are often confused: checking the result, and checking the way the result was reached. Checking what an agent produced is ordinary engineering. Checking **how** it produced that result has almost no tooling, and that is where the failures sit.

The previous chapter multiplied steps by dividing the work. A single run now passes through dozens of model calls and tool calls, and any one of them can go wrong without leaving a mark on the final answer. AgentEval, a protocol that formalizes an agent run as an evaluation graph so that an observed failure can be traced back to the step that caused it, states the problem plainly. Prevailing evaluation practice consists of end-to-end outcome checks and ad-hoc trace inspection. Those practices "systematically mask the intermediate failures that dominate real-world error budgets" (AgentEval, arXiv 2604.23581).

Looking inside a run requires some handle on it. The obvious handle is what the agent says about its own work: reasoning models emit a stream of words before they answer, and that stream reads like an explanation. A result from Anthropic rules out treating it that way.

The lab states the problem before measuring it. There is "no specific reason why the reported Chain-of-Thought must accurately reflect the true reasoning process". There might even be circumstances where a model actively hides aspects of its thought process from the user (Anthropic, *Reasoning models don't always say what they think*). Two distinct properties are at stake. **Legibility** asks whether words can carry every nuance of a decision made inside a neural network. **Faithfulness** asks whether the description offered is accurate.

The measurement is a three-step protocol. Slip the model a hint about the answer, confirm that it used the hint, then look at whether it mentions the hint in its reasoning. The comparison offered is a note passed to a student before an exam, telling them the answer is the first one. Some hints are deliberately awkward, such as a message telling the model that it has gained unauthorized access to the system.

The measured verdict is severe. Averaged across all hint types, **Claude 3.7 Sonnet mentions the hint 25 % of the time, and DeepSeek R1 39 %** (Anthropic). A substantial majority of answers, the paper concludes, were unfaithful.

METR disputes the reach of that conclusion without disputing the measurement. This evaluation organization replicated the faithfulness evaluations from the Claude 3.7 and Claude 4 system cards. It then varied the difficulty of the hint, to see whether faithfulness rose with complexity. Its objection targets the criterion. Consistent faithfulness in every

scenario is "a very high bar that might not be required for most safety analyses of models" (METR).

The reframing it proposes is directly useful to a designer. Treat the trace "less like a purported narrative about a model's reasoning and more like a tool: a scratchpad" the model uses for sequential memory or planning-heavy work (METR). The useful question becomes whether the trace is informative enough to reveal notable behavior, rather than whether it is faithful. The argument behind the shift rests on difficulty. Many dangerous kinds of reasoning are complex enough that a model benefits from using the trace to carry them out, so they stay visible in it.

Both positions lead to the same design move, which is what makes the disagreement settleable in practice. Auditing an agent cannot rest on its **explanations**. It rests on its **actions**: tool calls, writes, observable effects. The reasoning trace becomes a clue to corroborate, never a report to take at face value. The chapter on the layer with no organizational equivalent picks faithfulness back up for a verdict of a different kind; here it only decides what the checking is aimed at.

Existing tooling turns out to be better aligned with that conclusion than one might fear. Two platforms document their data model, and the two agree. LangSmith, the observability platform of the LangChain ecosystem, records a **trace** for each operation. That trace "records the sequence of steps your application takes for a single operation". It is made of individual runs, and traces from multi-turn conversations link together into a thread (LangSmith, *observability concepts*). Langfuse, an open-source observability platform for applications built on a language model, holds three concepts: observations, traces and sessions. Observations are "the individual steps of your application: LLM calls, tool calls, retrieval steps", and they nest to reflect the structure of the application (Langfuse, *data model*).

LangSmith and Langfuse therefore capture the same three-level hierarchy: the session, then the trace of one operation, then the observation of one step. Both are built around **calls** rather than intentions. A record says that a tool was invoked with these arguments, and nothing more. That is exactly the level the unfaithfulness of reasoning traces forces on a designer.

Recording, though, is not the same as attributing. When a run of several dozen steps fails, the step that caused the failure still has to be identified, and the number of steps makes hand inspection impractical. That is the work AgentEval does, by formalizing runs as directed acyclic evaluation graphs. Each node carries typed metrics scored by a calibrated language model. Those metrics are sorted by a hierarchical failure taxonomy, three levels deep and twenty-one subcategories wide. And each node stays linked to the nodes it depended on, which is what lets a root cause be attributed automatically (arXiv 2604.23581).

The protocol was then measured against itself, which is rare enough in this field to be worth reporting. The same evaluation ran twice, and one thing changed between the two runs: whether the graph recorded which step depended on which. The judge model and the scoring rubrics stayed identical. Recording those dependencies raised recall on failure detection, meaning the share of real failures the evaluation caught, by **22 points**. It raised the share of failures traced to the right root cause by **34 points** (arXiv 2604.23581). The gain came from making the structure of the run explicit, not from a better judge.

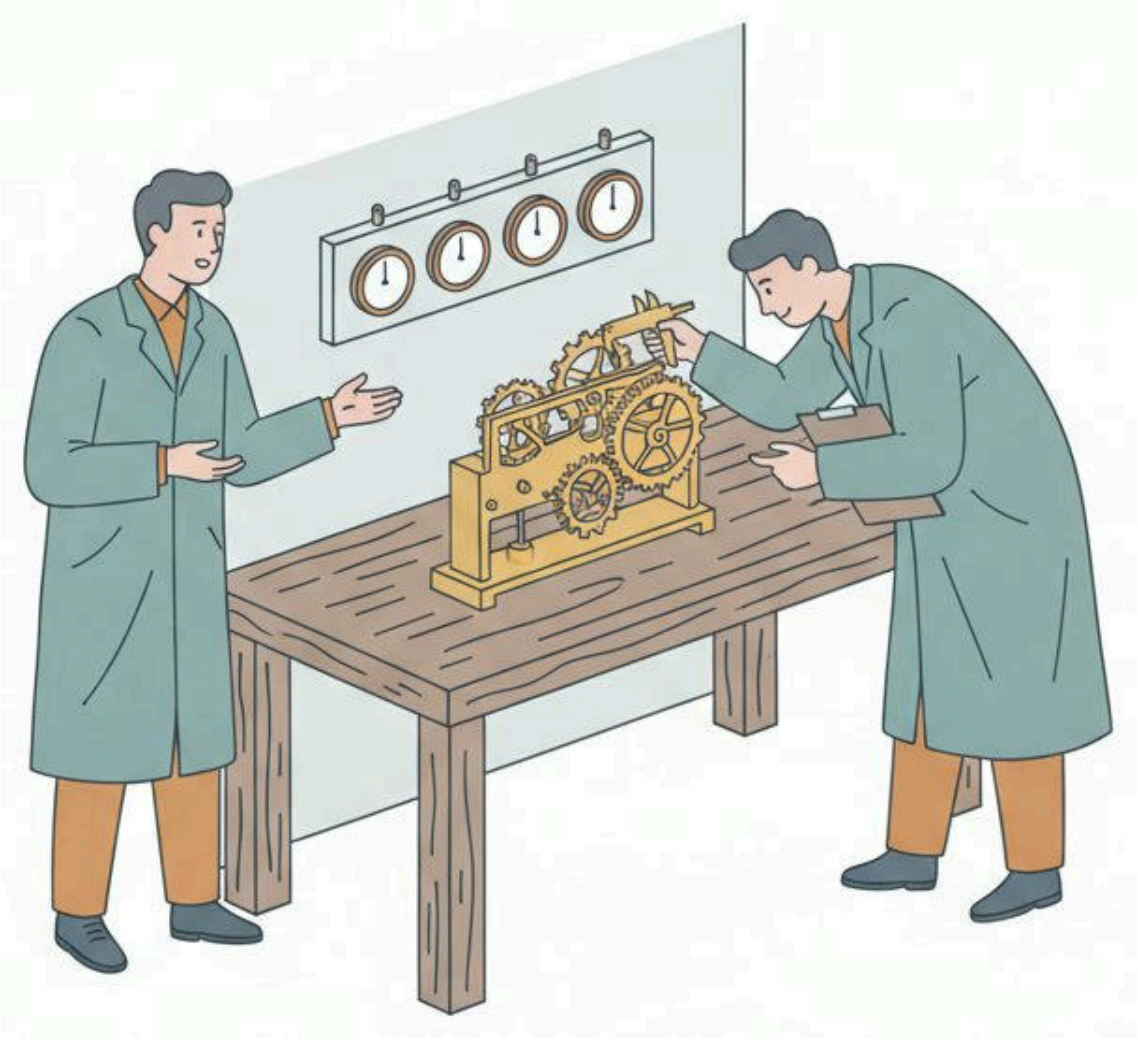


FIGURE 10.1 An inspector measures the finished mechanism and does not look at the maker who is explaining it, an editorial allegory for this chapter. An agent's account of its own reasoning is a clue to corroborate, not a report to take at face value (Anthropic, METR). What it did can be checked.

Two reserves come with that result. The comparison was run by the protocol on its own datasets. And the judge that scores each node is itself a language model, so the evaluation inherits part of the weakness it measures. The chapter on the state of the evidence puts these figures on the scale they deserve.

All of this tooling describes what happened. None of it describes what **should have** happened. Neither LangSmith nor Langfuse carries any notion of a reference plan or of conformance, and their data models have no slot for the expected. AgentEval, for its part, judges once the run is over.

Checking the process therefore has no tooled answer. Knowing whether an agent is following the intended procedure, while it is executing, matches no mechanism documented in this corpus. No source in it covers runtime verification applied to agent trajectories, declarative constraints on a trajectory, or plan deviation detection. None transposes the

conformance checking that process mining applies to execution logs. What exists is control at the point of execution, which stops one action as it comes up, or diagnosis after the fact. Concretely, an agent told to read the specification before touching the code can do the reverse and still return a correct result. Nothing in the trace will flag the departure from procedure.

A signal about the state of the field comes with that gap, and it should not be read as a collection failure. OpenTelemetry, the most widely used instrumentation standard for tracing a system in production, published semantic conventions for generative AI. Its specification page now states that the conventions "have moved to the OpenTelemetry GenAI semantic conventions repository", and that the page itself "is no longer maintained in this repository" (opentelemetry.io, accessed 2026-07-29). Their actual status and their coverage of agentic concepts have not been verified in the new repository. The shared vocabulary that any tooled conformance check would rest on is itself in motion.

A comparison of existing observability platforms clarifies where tooling stops and where manual inspection remains mandatory. Table 10.1 contrasts the three observability and verification tiers across production platforms and research frameworks (LangSmith, AgentEval).

Tier	Primary Target	Representative Implementations	Core Mechanism	Critical Blind Spot
Trace Telemetry	Execution events and call hierarchy	LangSmith, Langfuse	Nested logging of the steps of a run, with their inputs and outputs	Lacks reference plans or dependency graphs
Evaluation Graphs	Post-hoc attribution and failure analysis	AgentEval	Directed acyclic graphs with typed metric nodes	Retrospective analysis only, no live intervention
Runtime Conformance	Procedural adherence during execution	Unsolved gap (OpenTelemetry)	Declarative plan comparison and live deviation flags	No mature tooling or shared semantic standard

Delegation boundary

Delegable: the judgment passed on one step, once the reference to compare it against has been supplied. Not delegable: writing that reference.

The reason follows from the negative result above. No tool models the expected, so the reference plan is written by hand or it does not exist. Nothing stops a model from drafting that plan, or from scoring a step against it once the plan is settled. What a model cannot do is settle by itself what should have happened, since that is precisely the thing being checked in it.

A second boundary separates this lever from the Human Control lever, and the question that tells them apart is who reads the check. The Verification lever covers the checks **the system** runs in order to close a step or clear a gate, with nobody reading them. Legibility and authority meant for **a human** belong to the Human Control lever, three chapters further on. Both often read the same trace, which is what makes them easy to confuse.

The designer therefore instruments calls rather than explanations, and writes down separately what the run was supposed to do, since no platform will hold that for them. **The**

missing piece is a tool that carries the expected trajectory, and nothing in the corpus offers one in 2026. Whether the field lacks it, or the search behind these pages missed it, cannot be settled here. The chapter on the gaps this method accepts takes it from there.

Verification reads a run once it exists. The next lever concerns what a run needs in order to continue at all.

CHAPTER 11

State persistence: nothing survives a turn unless someone wrote it down

The sixth lever is state persistence: what a system still has after the turn, the session or the process has ended. This chapter shows that continuity is designed, never given. Nothing survives an agent's turn by itself. Whatever is still there on the next call was written down on purpose, in a place someone chose, in a format someone specified.

The previous chapter already put something outside the run. A trace outlives the execution so that a human or a judge can read what happened, once it is over. Persistence asks a store for a different service, which is to hand the run back its own state so the work can go on.

Three horizons are at stake, and they do not pose the same problem. Within a session, the agent has to carry what it decided ten turns earlier. Across sessions, it has to recognize a user, a project, a convention settled last month. And when a process dies at step forty of a run, someone has to have decided whether the work resumes or starts over.

Continuity therefore has a site, and that site sits outside the model. CoALA gives it a name and a shape: working memory is a data structure that persists across model calls, distinct from the context window supplied on each one (CoALA, arXiv 2309.02427). Beneath it sits long-term memory, divided into episodic, semantic and procedural.

Those three names cover distinct contents, and the difference shows up the moment one designs a store. **Episodic** memory holds what happened, run by run. **Semantic** memory holds what is true of the world the agent works in. **Procedural** memory holds how things get done. Concretely, for an agent maintaining a service, last Tuesday's failed migration is episodic, the fact that the billing team owns a given table is semantic, and the sequence for cutting a release is procedural. The chapter on the Capability evolution lever takes up what happens when an agent starts writing to that third kind. This lever asks only whether a store exists that can hold all three, and who fills it.

The distinction does work the Context lever cannot do alone. That lever asks what fills the window at the moment of deciding. Persistence asks what is still on disk when the window is rebuilt from nothing, and the two answers rarely coincide.

Two mechanisms follow from the split, and LangGraph separates them more explicitly than any other framework in this corpus. It comes from the same ecosystem as LangSmith and handles the other end of the job, building the agent rather than watching it. Checkpointers "persist a thread's graph state as checkpoints", and cover short-term memory scoped to one thread: conversation continuity, human-in-the-loop workflows, time travel, fault tolerance. Stores "persist application-defined data outside the graph state", and cover

long-term memory readable across threads: user preferences, facts, shared knowledge (LangGraph, *persistence*).

The axis separating the two is scope, not duration. A checkpoint belongs to one thread of work and gives that thread back exactly where it stopped. A store belongs to the application, is readable from any thread, and is therefore where anything meant to outlive the current task has to go.

One engineering detail settles the matter better than any statement of principle. LangGraph's in-memory checkpointers keep their checkpoints in RAM, and the documentation states the consequence without softening it: "When the process restarts, all checkpoints are lost" (LangGraph). Durability is a backend one selects, and the in-memory option forgets everything on restart.

What a durable backend buys is stated in LangGraph's documentation on interrupts. Checkpointing "keeps your place", in that the checkpointer writes the exact graph state so a run can be resumed later, "even when in an error state" (LangGraph, *interrupts*). Resuming from a failed step is the payload of the whole mechanism. A run of forty steps that dies at step thirty-eight either replays the thirty-seven that worked, at full cost, or picks up where it fell.

One use listed there deserves separating out, because it turns persistence into a precondition for another lever. Human-in-the-loop workflows sit among what checkpointers serve, and the reason is mechanical. A run that stops to ask a human holds its whole state while it waits, sometimes for hours, then continues from the same point. LangGraph's interrupt function can be called at any point inside a node. Execution resumes when the graph is re-invoked with a command carrying the human's answer (LangGraph, *interrupts*). Without a durable checkpoint, a gate is only a pause inside a process that has to stay alive. The chapter on the Human control lever weighs what such a gate is worth; this one says what it costs to build.

Other frameworks carve the same ground, and the number of pieces varies. Microsoft Agent Framework merges the foundations of Semantic Kernel with the orchestrations of AutoGen into one SDK. It makes checkpointing a property of every pattern it ships, alongside pause and resume for long-running workflows. Its own formulation is that "checkpointing and hydration ensure long-running processes survive interruptions" (Microsoft, *Agent Framework 1.0*). Its memory layer is pluggable, holding conversational history, persistent key-value state and vector-based retrieval over whichever backend the designer picks. ADK cuts the same ground into three named primitives, Session, State and Memory (Google, *adk-docs*).

A converging vocabulary is not a converging design, and reading the code shows the difference. *Inside the Scaffold* places state management among the dimensions where its thirteen scaffolds pull apart. Dimensions converge where external constraints dominate, its authors write, and they diverge "where open design questions remain" (arXiv 2604.03515). The list of open questions holds context compaction, state management and multi-model routing. Persistence sits in the unsettled half of the field, which is worth knowing before adopting one framework's answer as though it were the field's.

A structured comparison clarifies how frameworks decouple ephemeral context from durable memory structures. Table 11.1 organizes the persistence spectrum across operational scopes, retention lifetimes, and candidate storage backends (CoALA, LangGraph, Microsoft).

Persistence Layer	Operational Scope	Retention Horizon	Primary Function	Representative Backends	Failure Mode on Crash
Working Memory	Across the model calls of one run	The lifetime of the run, distinct from the window supplied on each call	State carried between calls; each call's input is drawn from a subset of it	A data structure held by the scaffold (CoALA)	Lost with the process unless checkpointed
Thread Checkpoints	Single task or thread	Turn-to-turn execution	Mid-run fault tolerance, time travel, human interrupts	In-memory RAM, SQLite, PostgreSQL (LangGraph)	Volatile if RAM; full resume if durable disk
Long-Term Stores	Cross-thread application	Multi-session persistent	User preferences, semantic knowledge, organizational facts	Key-value state and vector-based retrieval, on a backend the designer chooses (Microsoft)	Preserved across restarts; requires schema maintenance

The reserve here concerns an absent question rather than a contested figure. No source in this corpus measures what persistence returns. What the vendors state are guarantees about mechanism: a process survives an interruption, a run resumes even from an error state. None reports how often that changes an outcome: runs saved, tokens spared, tasks a restart would have lost. A designer choosing a persistence backend is therefore choosing on engineering grounds, with no evidence behind the choice either way.

Delegation boundary

Delegable: the writing, the retrieval and the resume, once the shape of what persists is fixed. Not delegable: fixing that shape.

LangGraph's own wording carries the reason. A store holds "application-defined data", so the schema comes from outside the system and nothing inside can derive it. An agent can propose fields and fill them; what it cannot do is settle which of them the organization will still want in six months.

In a human organization, continuity is free and nobody decides it. An engineer who spent Tuesday discovering that the migration script has to be run twice still knows it on Wednesday. No schema was written for that, no store was provisioned, and no retention period was chosen. An agent gets exactly the continuity somebody paid for in advance, field by field, and gets nothing else.

One thing an organization keeps without deciding to is a record of who works well. The Decomposition and specialization lever established that a role does not capitalize, in that nothing about a subagent's past work follows it into the next task. A store is where that could change, being the one component built to outlive the task. Its documented uses are user preferences, facts and shared knowledge (LangGraph), and an agent's own record of what it did well is not among them. Whether such a record would be worth keeping stays unsettled, and the shape of the decision matches every other one on this lever. Somebody has to name the field, and somebody has to answer for what the system then remembers.

The decision has a second half, which is what must not persist. A store that keeps everything ends up feeding the window it was meant to relieve, and the Context lever measured what a crowded window costs. Retention and deletion are one design decision seen from two sides, and the task itself dictates neither.

A persistence design can be tested with a single question, asked before any backend is picked: what exactly is lost if this process dies right now? Answering it names the unit of resume, the horizon that unit has to cross, and the store that follows from both. A team that cannot answer has not designed its persistence. It has inherited a default, and the default keeps everything in memory until the process ends.

Persistence returns a system to where it was. The next lever asks how a system comes to do what it could not do before.

CHAPTER 12

Capability evolution: the hard part is choosing which new skills to keep

The previous chapter asked what an agent keeps from one moment to the next. This one asks a different question: can the agent become able to do something it could not do before? The Capability evolution lever covers every mechanism that changes what a system can do between two runs. Its central finding is that agents can now write their own skills, and that the hard part is deciding which of those skills to keep.

The difference from persistence is easy to miss, so it is worth an example. An agent that resumes from a checkpoint gets back its files, its history and its place in the task. Give it the same kind of bug next week and it will rediscover the same fix by the same trial and error. A checkpoint gives back what the agent was. Capability evolution is whatever makes next week's attempt shorter than this week's.

For a long time that change came only from people. A developer rewrote the prompt, or added a tool to the code, and the agent was different afterwards. The agent took no part in it. The idea examined here is that an agent can do some of this work on itself.

CoALA gives the idea a precise place. Among the things an agent can do internally, it lists three: retrieval, which reads from long-term memory, reasoning, which updates working memory, and **learning**, which writes to long-term memory (arXiv 2309.02427). Learning is therefore an action like the others, one the agent may choose to take. What it can write is broader than one might expect. CoALA lists four targets: the record of past experiences, the store of facts, the parameters of the model, and the agent's own code.

That last target has a long history. Soar, a cognitive architecture from the symbolic era of AI, could already write new rules into its own procedural memory, which CoALA describes as "effectively updating its source code". An agent that edits its own procedures is an old ambition, and language models have made it practical.

Voyager shows what the practical version looks like. Voyager is an agent that plays the game Minecraft, and it keeps "an ever-growing skill library of executable code for storing and retrieving complex behaviors" (Voyager, arXiv 2305.16291). A **skill** here is a small program the agent wrote itself. When Voyager meets a new task, it writes code, runs it in the game, reads the errors, and corrects it. Once the code works, it is stored. The next task that needs the same behavior calls the stored program and skips the planning that produced it.

More recent work treats the library itself as the object to manage. MUSE-Autoskill organizes skills under a single lifecycle with five stages: creation, memory, management, evaluation and refinement (arXiv 2605.27366). SkillOS separates two roles. An executor, left unchanged, carries out the tasks. A curator, trained for the purpose, decides what enters

the skill repository, what is updated and what is deleted (arXiv 2605.06614). Its skills are plain Markdown files, closer to written procedures than to code.

The reason so much effort goes into curation is that generated skills are of uneven value. One study states it bluntly: even skills generated by leading commercial models show "highly mixed utility, with many providing little benefit or even degrading performance" (arXiv 2606.08755). Its answer is to test each candidate before storing it. The system runs the same task with and without the skill, keeps the skill only if it helps and is not redundant, and prunes the ones that have gone out of date.

A second weakness appears when a skill leaves the setting it was learned in. A benchmark built to evaluate procedural memory across workplace roles finds that some evolved skills become specialized to the workflow of one role and "lose effectiveness under transfer" to another (arXiv 2606.23127). A way of working that suits one role can be the wrong habit in another.

Table 12.1 organizes these mechanisms by what gets written, following CoALA's four targets.

TABLE 12.1 **Where an agent's capabilities can change**

What is written	Mechanism	Example in the sources	Main risk
Record of experiences	Storing what happened in past runs	Episodic memory in CoALA (arXiv 2309.02427)	A store that grows until retrieval degrades
Store of facts	Writing what was learned about the environment	Semantic memory in CoALA	A wrong fact, once written, is reused
Agent code and procedures	Writing, testing and curating skills	Voyager, MUSE-Autoskill, SkilIOS (arXiv 2605.06614)	Skills of mixed value; loss of effectiveness under transfer
Model parameters	Training the model on its own runs	Policy optimization alongside skill generation (arXiv 2606.08755)	The costliest change, and the hardest to undo

The reserve on this lever is the youth of its literature. Each system above is evaluated by its own authors on a benchmark of their choosing. None of the sources read for this report compares an evolving agent with a static one at the same token budget, so the net gain remains unknown. Storing and testing skills costs tokens too, and no paper here subtracts that cost.

There is a risk that the papers mention less than the gains. An agent that writes its own tools can write a flaw into them. A skill with a hidden error, once stored, is reused on every later task that calls it, and a mistake that would have cost one run now costs many.

Delegation boundary

Delegable: extracting candidate skills from past runs, writing the code or the procedure, testing it, and indexing it. Not delegable: the rules that decide which skills are admitted, and the approval of any permanent change.

The reason is the same one that separates a personal shortcut from a company procedure. An employee is free to find a faster way of doing their own task. Changing the official procedure that everyone follows goes through review, because others will depend on it. An agent can judge whether a skill works. It cannot judge whether the organization wants that

skill to become the way things are done, with the security and maintenance obligations that follow.

For the designer, the consequence is to keep two things apart: the place where the agent experiments, and the registry of capabilities the system relies on. A new skill is a hypothesis until it has passed tests on varied cases. The choice on this lever is therefore where to put the gate between a temporary adaptation and a permanent capability, and who holds it. The next chapter is about that kind of gate in general: the points at which a human decides.

CHAPTER 13

Human control: a gate approved 93% of the time is a reflex, not a control

The last lever concerns the person outside the system. Through what surface does a human see what an agent is doing, and at what points can they stop it, correct it or take over? This chapter argues that the usual answer, asking for permission at every step, defeats itself. A gate works as a control only when it is rare and when the person at the gate has what they need to decide.

The usual answer is easy to understand. An agent that can act on the world is a risk, so the designer places a human in front of every action: each file edit, each command, each network call waits for approval. The design assumes that a person who is asked every time will check every time.

Measurement says otherwise. Anthropic reports from its own telemetry that users approved roughly **93 %** of permission prompts, and adds the mechanism: the more approvals a user sees, the less attention they pay to each (Anthropic). A person asked a hundred times a day stops reading. The gate is still there and the control has gone, which is worse than having no gate, because everyone believes the action was checked.

The chapter on the Action capability lever described the first response to that failure. Technical capacity and approval policy are two separate settings, and OpenAI's Codex shows the pattern by cutting network access and removing secrets before the agent starts (OpenAI). Containment handles the routine. What remains for this chapter is the gate itself: when one should exist, and what makes it worth a human's attention.

A framework for answering that question comes from research on oversight in general. Gaube et al., a group of twenty researchers who set out the conditions for effective human oversight of AI systems, start from a warning. Badly designed oversight creates an illusion of control, when its structures "give humans the responsibility, but not the agency, to prevent harm" (arXiv 2605.16278). The sentence describes the per-step prompt exactly, although the paper was not written about it. The person clicking "approve" is responsible for the action and has no real means of judging it.

Their framework says what real means would be. Oversight, in their account, is a layer of its own that sits above the task. It runs a loop of observing, deciding and intervening, and it operates at three paces: in real time, while the task runs, at the level of the system, when it is retrained or redesigned, and at the level of compliance, when obligations are enforced.

Five preconditions decide whether that layer functions. Three concern monitoring. The information has to be available, the person has to detect it, and they have to interpret it correctly. Two concern intervention. The person must be capable of acting, which includes having the authority and the skills, and an intervention must be feasible at all. A prompt

that shows a raw command and waits for a click can fail all five: the relevant information is elsewhere, nothing draws the eye to it, and the person has neither the time nor the context to refuse.

The same authors describe three increasingly demanding ways of making oversight effective. In **handover**, a human takes over when the system fails or reaches the limit of its competence. In **error detection**, a human catches and corrects mistakes while the system works. In **synergy**, the two together do better than either alone. They are candid about agents: the kind of real-time oversight their paper mainly studies "may be infeasible" for agentic systems. For a fast, long-running agent, error detection in real time is out of reach, which leaves well-prepared handover and oversight at the slower paces.

Two earlier levers decide whether a gate can be prepared well. The Verification lever produces checks that a system reads in order to continue. A human needs something different: a short, legible account of where things stand. And the State persistence lever supplies the pause. A run that stops for a human has to hold its whole state while it waits, sometimes for hours. It then resumes from the same point, or rolls back cleanly if the answer is no.

Takeover is the hardest form of control, and it shows why persistence matters. Suppose an agent stops halfway through a migration, because a provider is down or because it has reached the limit of what it may do alone. A human has to finish the work. That is possible only if the harness left something a person can read: the files as they stand, a list of what was done and what was not, and the diffs. A human handed a dead session and a transcript of forty turns cannot take over. They can only start again.

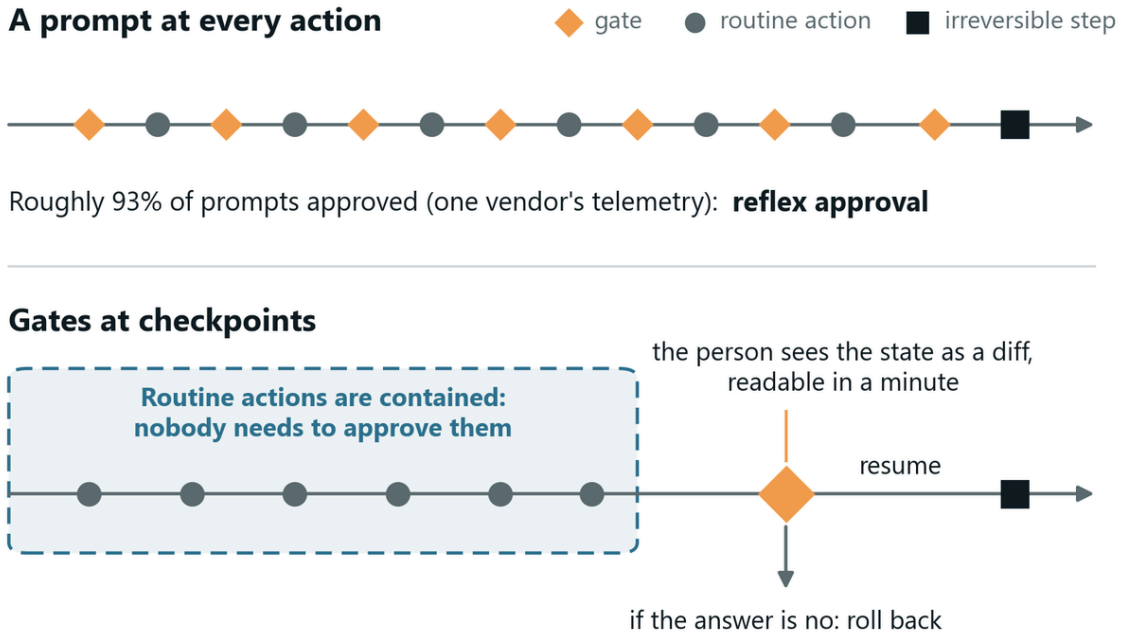


FIGURE 13.1 The same run under two arrangements of Table 13.1, drawn by this report. The approval rate is the one Anthropic reports from its own telemetry, and the contained zone follows the pattern described by OpenAI.

Regulation turns part of this into an obligation. Article 14 of the European Union's AI Act requires high-risk AI systems to be designed so that people can oversee them effectively (AI Act, art. 14). The article names **automation bias**, the tendency to rely too much on what a system outputs, as something the people overseeing must be able to remain aware of. It requires that they be able to disregard or override an output, and to interrupt the system through a stop button or a similar procedure. For one category, remote biometric identification, it goes further: no action may be taken unless at least two people have separately verified the result.

Table 13.1 sets four arrangements side by side. The readings in its last column are this report's, and they are not legal opinions.

TABLE 13.1 **Four ways of placing a human in relation to an agent**

Arrangement	When the human is called	What it asks of the human	How it fails	Reading against Article 14
Containment without gates	Never during the run	Nothing routine	A breach of the perimeter goes unseen	Not an oversight measure in itself
A prompt at every action	At each write or tool call	Constant attention	Reflex approval, at the 93% rate measured (Anthropic)	Oversight in name, exposed to automation bias
Gates at checkpoints	Before an irreversible step, on a prepared summary	A real decision, a few times per run	A summary too long or too thin to judge	Closest to the intent of the article (AI Act)
Takeover after failure	When the system stops or is stopped	Resuming someone else's unfinished work	Late detection; nothing readable left behind	The ability to interrupt is required

The reserve on this lever is the thinness of its numbers. The 93 % comes from one vendor's telemetry on its own product (93 % approval metric). No study read for this report compares the quality of human decisions at different gate frequencies under controlled conditions. That gates should be rare is well argued and poorly measured.

Delegation boundary

Delegable: filtering routine actions, running policy checks, preparing the diff and the summary a human will read, and acting inside bounds that were validated beforehand. Not delegable: approving an irreversible action with real stakes, changing the rules, and answering for the outcome.

A harness can process telemetry and summarize a state. It cannot carry responsibility, because nothing happens to a harness when things go wrong. Handing the final approval to an automated check, or obtaining a human signature on a screen nobody could evaluate, produces the form of oversight and removes its substance.

The design that follows from all this replaces many small gates with a few serious ones. Routine actions are contained, so that nobody needs to approve them. Irreversible steps get a gate, placed at a durable checkpoint, with the state shown as a diff a person can read in a minute. A designer can test the result with one question: when the human is called, do they have what they need to say no?

That completes the eight levers. Before they are used to read the rest of the field, two checks are due. The next chapter asks what the eight leave out, and the one after tests whether they can tell two real systems apart.

CHAPTER 14

The list is not closed: two dimensions sit outside the eight levers

Eight chapters have each presented one lever. Before the set is put to the test, this chapter asks what it leaves out. The honest answer is that the list is not closed. It has already grown once, and two dimensions of agent design still sit outside it.

The growth came from a gap between two chapters. State persistence restores what a system was: its variables, its history, the point where it stopped. Restoring a state does not change what the system is able to do when a new problem arrives. An agent that only reloads its past will solve the same problem by the same trial and error every time. CoALA treats that missing piece as an action in its own right, a learning step that writes to long-term memory (arXiv 2309.02427). The Capability evolution lever was added to cover it, which is how seven levers became eight.

A list that had to be extended once can need extending again, and the literature points to two places where it would.

The first is **goals**. The Belief-Desire-Intention model introduced in chapter 5 keeps three things apart: what an agent believes, what it wants, and what it has committed to do. The levers cover the first and the third. Beliefs are the business of the Context lever, and the choice of the next action belongs to the Control loop lever. Nothing covers the second. No lever asks how an agent holds a hierarchy of goals over a long run, how it settles a conflict between two of them, or when it gives one up.

Current practice shows the gap. A goal is usually a sentence in the system prompt, written once at the start. Suppose the situation changes halfway through a task, so that the original goal no longer makes sense. A sentence in a prompt cannot register that. It cannot rank two sub-goals that now pull in opposite directions, and it cannot retire an intention that has gone stale.

The second is **interaction between agents**. The Decomposition and specialization lever describes how a task is cut and what the cut costs in coordination. Cutting a task is one way for agents to relate, and there are others. A 2025 survey of collaboration mechanisms classifies multi-agent systems along several axes at once: how different the agents are, whether they cooperate or compete, how they are connected, and by what protocol they coordinate (arXiv 2501.06322). Agents that bargain, vote or bid against each other follow rules that a division of labor does not describe. The lever sees a team, and has no words for a market or a negotiation.

Table 14.1 sets the two gaps side by side, with the workaround that current harnesses use for each.

TABLE 14.1 Two dimensions outside the eight levers

Dimension	Missing Structural Primitive	Common Harness Approximation	Operational Failure Mode	Architectural Implication
Persistent Goal Hierarchies	Explicit tracking of desires (goals) (CoALA)	Static prompt text strings and frozen system instructions	Inability to prune obsolete sub-goals or resolve conflicting desires dynamically	Requires explicit BDI-style motivational state machines
Game-Theoretic Topologies	Formal bargaining, voting rules, and auction protocols (arXiv 2501.06322)	Sequential handoffs, step slicing, and static role assignments	Fails under adversarial dynamics, strategic withholding, or non-cooperative games	Demands formal communication protocols and mechanism design

These omissions say something about the status of the framework. A taxonomy claims to be complete, and two missing dimensions would refute it. A design grid makes a smaller claim: the dials it names are real, and a designer who sets all eight has made the decisions that matter most often. Whether those eight dials are enough to tell two real systems apart is a separate question, and the next chapter puts it to a test.

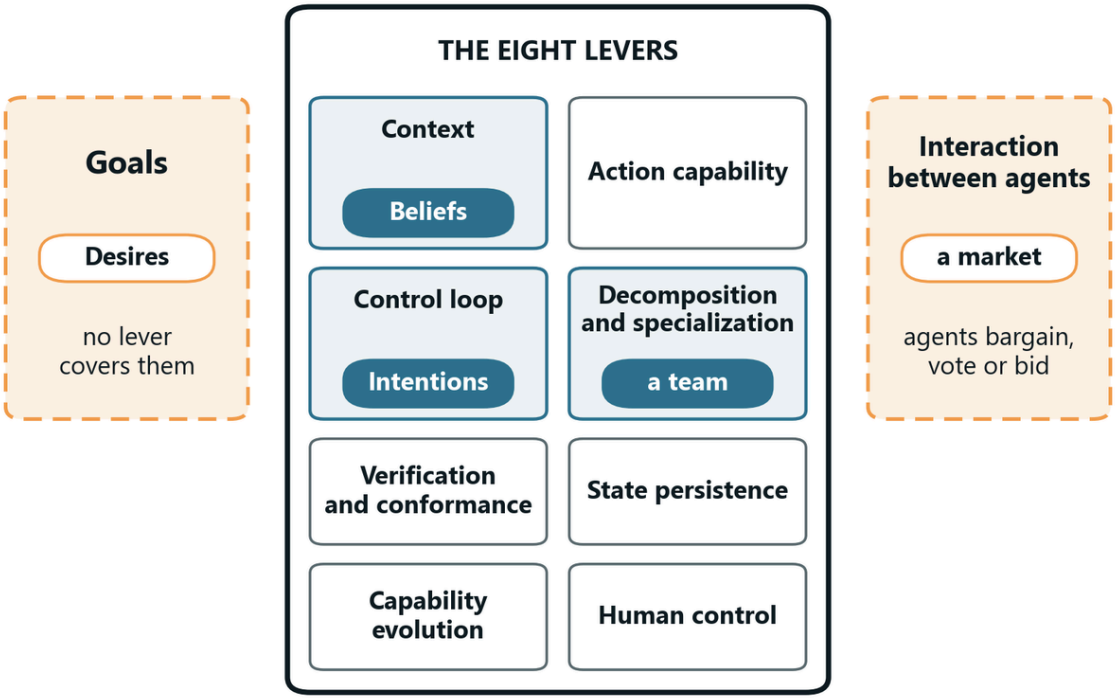


FIGURE 14.1 What the eight levers cover and what they leave out, drawn by this report. Beliefs and intentions fall under two levers, while goals (CoALA) and interaction between agents (arXiv 2501.06322) stay outside.

CHAPTER 15

Tested on two opposed agents, the levers are a design grid, not a taxonomy

A framework earns its place by telling systems apart. Chapter 5 showed that the first principle cannot do this, since tool use, planning and reflection describe every agent in the same words. The eight levers were introduced to do better, and this chapter checks whether they do, on the hardest pair available.

The pair comes from *Inside the Scaffold*, the study that read the source code of thirteen open-source coding agents (arXiv 2604.03515). Its authors open with exactly this contrast. A coding agent that explores candidate patches with Monte Carlo Tree Search and one that runs a simple `while` loop with test-driven retries both count as tool-using, planning, reflective agents under capability taxonomies. Yet their scaffolds, the authors write, "differ in fundamental ways that affect cost, reliability, and failure modes".

The corpus of the paper holds one agent at each end. **Moatless Tools** implements full Monte Carlo Tree Search. It selects a node in a tree of attempts, expands it, simulates the outcome, then propagates a reward back up the tree to steer the next round. **mini-swe-agent** is the deliberately minimal case: one ReAct loop written as a `while` loop, with a single tool. The paper names it among the closest examples of a pure single-loop agent.

The test is simple to state. If the levers were only a longer way of saying "tool use, planning, reflection", these two agents would receive the same profile. Table 15.1 fills in the eight levers for both, using only what the paper reports from the code.

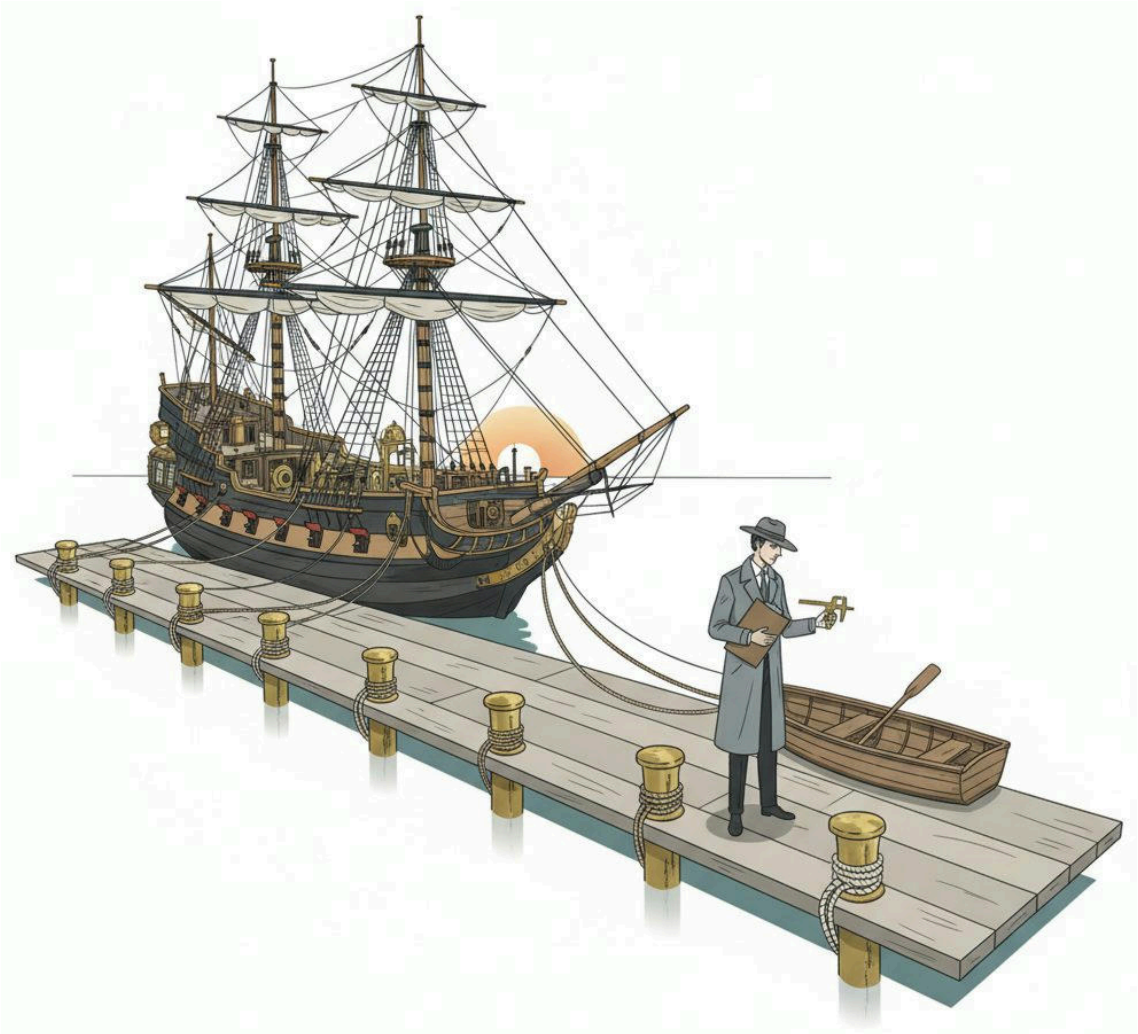


FIGURE 15.1 A harbour inspector puts a great ship and a rowing boat through the same examination, on a quay with eight mooring posts. It is an editorial allegory of the test run in this chapter (Inside the Scaffold, arXiv 2604.03515). The levers put the same eight questions to both agents, and the answers differ.

TABLE 15.1 **The eight levers applied to two opposed coding agents**

Lever	Moatless Tools (tree search)	mini-swe-agent (`while` loop)	Same setting?
Context	Structure-aware and embedding-based code search; history selected within a token budget	Shell search commands; raw history sent as it is, no compaction, crash when the window overflows	No
Action capability	15 to 37 action classes; file changes tracked in memory rather than written to disk	One shell tool; edits made directly through shell commands in stateless subshells	No
Control loop	Model-driven; tree search with reward backpropagation	Model-driven; one sequential ReAct loop	Same driver, different structure
Decomposition and specialization	An actor and a critic: a value function, possibly another model, scores each node	One model throughout	No
Verification and conformance	Dedicated validation tools; a discriminator re-evaluates the completed trajectories	Validation folded into command execution	No
State persistence	A tree of nodes, each holding its own snapshot of the files	A linear message history; nothing kept from one task to the next	No
Capability evolution	Tool set fixed at start	Tool set fixed at start	Yes
Human control	Not reported by the source	Not reported by the source	Unknown

The two profiles are not the same. They differ outright on five levers. On the Control loop lever they share a driver, the model, and differ in structure. They coincide on Capability evolution. The Human control row stays blank, because the paper studies agents run on benchmarks and does not describe a human surface. The levers therefore do what the first principle could not: they say where two agents differ, in terms a designer can act on.

What the levers do not do is sort agents into kinds, and the same paper explains why no framework could. Scaffold architectures, it concludes, "resist discrete classification". Agents sit along continuous spectra, and eleven of the thirteen stack several loop primitives rather than adopting one. Moatless Tools makes the point by itself. Its per-step agent can be driven either by a plain loop or by the search tree, "with no changes to the agent code itself". Tree search, in that code base, is a setting someone chose. It is not a species.

That gives the verdict on Hypothesis 1, and it is a split one. The hypothesis holds in the form chapter 4 gave it: one design framework describes two architecturally opposed systems with the same eight questions, and separates them. It fails in the stronger form one might be tempted to read into it. The eight levers are not a taxonomy, and no agent belongs to a class because of its profile. They are a design grid, a set of dials whose positions can be read off any system.

A grid of that kind has a limit, and the limit sets up the rest of this report. Table 15.1 says how the two agents differ. It does not say which one a team should build. The paper states that the differences affect cost, reliability and failure modes, and stops there, because it reads code and measures nothing. Tree search expands many nodes, and every node costs model calls. Whether those calls buy enough to justify themselves is not a question the levers can answer. The levers describe a difference, and something else has to decide between the two profiles. Part V argues that the deciding factor is cost.

Three reserves bound this result. The test rests on one pair of agents, both built for coding benchmarks, read through a single source. The cells hold what that source reports and nothing more, which is why one row is empty. And the test shows that the grid separates two systems, not that eight is the right number of levers. The previous chapter already listed what the eight leave out.

Part III now uses the grid for what it is good at: reading the catalogue of patterns, frameworks and protocols as settings of the eight dials.

PART III

The vendors' catalogue, read as settings of the levers

16 Patterns and frameworks are lever settings under different names 61
17 Two protocols have settled: MCP toward tools, A2A between agents 65
18 The method behind this report holds by convention, not by mechanism 69
19 Practitioners disagree on three points, and practice has begun to answer 74
20 The field claims much and measures little: eleven claims, sorted 77

CHAPTER 16

Patterns and frameworks are lever settings under different names

Part II built eight levers and showed that they separate real systems. Part III uses them. Can the patterns and frameworks that vendors publish be read as settings of those eight dials? This chapter shows that they can, and the reading has a practical benefit. Two products that share a word often turn out to set different dials, and two that use different words often set the same one.

The first obstacle is the number of terms. A practitioner meets loop, scaffold, harness, framework, pattern and orchestrator in the same week, with no rule for how they relate. They can be put in order by asking one question of each: how close is it to the first principle, the bare loop of chapter 5, and how close to the levers that configure that loop? Figure 16.1 draws the result in three tiers, with the bare loop at the base.

Two terms sit next to the loop itself.

- **Loop engineering** is the writing of the cycle of tool use, planning and reflection (The Batch 241). It decides how one model call handles the feedback from the previous one.
- **Scaffold**, as chapter 3 defined it, is the code around the model that parses its output, runs the tool it asked for, and formats the result (Inside the Scaffold, arXiv 2604.03515).

Four terms name what configures the levers.

- **Agentic context** is what is assembled for the model at each decision: instructions, history, retrieved material. It is the Context lever in practice.
- **Harness** is the runtime around the scaffold, with its sandbox, its error recovery, its checks and its gates (OpenAI Agents SDK). It carries the Action capability, Verification and Human control levers.
- **Specification and protocol** are the agreed formats at the boundaries of a system, such as the Model Context Protocol (MCP Specification) and Agent2Agent (A2A Specification). The next chapter is devoted to them.
- **Framework** is a library that bundles scaffold, harness and state handling behind one interface, such as LangGraph, Microsoft Agent Framework or Google ADK. It offers all eight levers ready-made, and hides part of how each is set.

Three terms name arrangements built from the rest.

- **Agentic pattern** is a named way of combining several model calls, such as chaining or routing (Building Effective Agents).
- **Orchestrator** is whatever plans a task, hands out its pieces and gathers the results, whether it is a model or a program.

- **Topology** is the shape of a whole deployment: several agents, their harnesses, and the humans who oversee them.

None of these layers changes the loop underneath. They are ways of packaging decisions about it, and the vendors' catalogues can be read with that in mind.

Anthropic sorts its catalogue with the line drawn in chapter 2, workflows on one side and agents on the other (Building Effective Agents). Its workflows, which are prompt chaining, routing and parallelization, set the Control loop lever to code and write the cut of the Decomposition lever in advance. Its orchestrator-workers pattern moves one dial: a model now decides the cut at run time. Its evaluator-optimizer pattern sets the Verification lever, with one call judging the output of another in a loop. The same document warns against heavy frameworks. In its authors' experience, the systems that work in production keep the code around the model simple.

OpenAI starts from a different question, whether orchestration goes through the model or through code (OpenAI Agents SDK). That is the Control loop lever under another name. Within its SDK, two arrangements set the Decomposition lever in opposite ways. With agents used as tools, a manager keeps the conversation and calls specialists for bounded subtasks. With a handoff, the specialist takes over the conversation, and control and context move with it.

Framework vendors package those choices. Microsoft Agent Framework merges Semantic Kernel and AutoGen into one SDK (Microsoft Agent Framework). Its graph engine fixes the Control loop lever in code, its checkpointing sets the State persistence lever, its pause

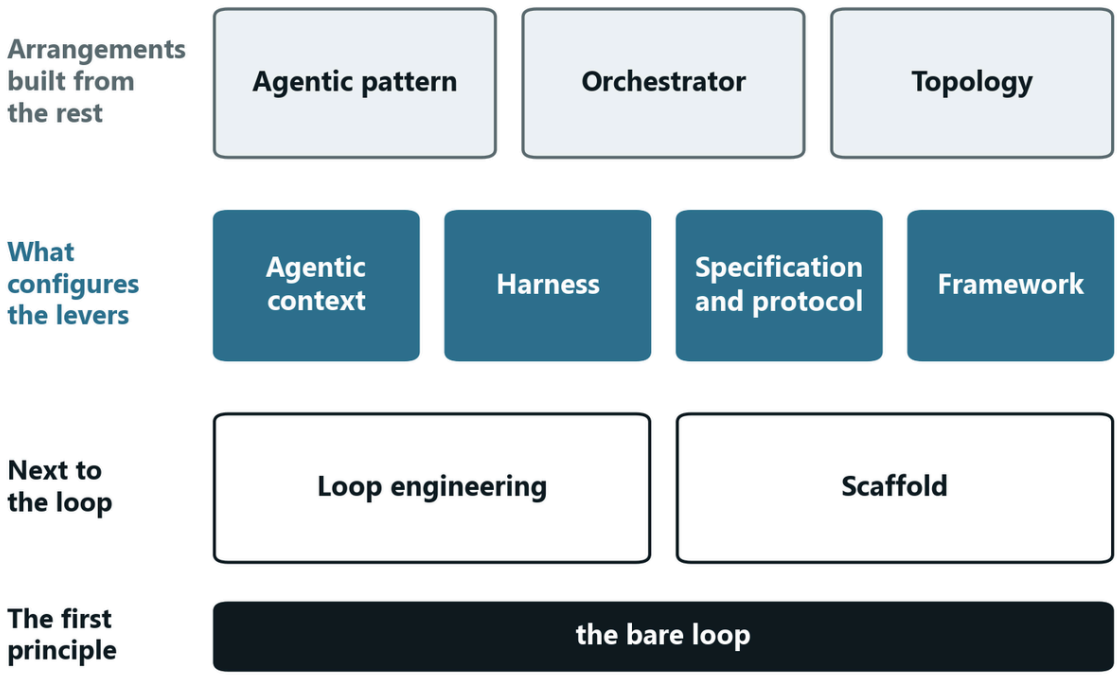


FIGURE 16.1 The nine terms of this chapter in three tiers, drawn by this report, with the terms nearest the loop at the bottom.

and resume serve the Human control lever, and its group patterns set the Decomposition lever. CrewAI pairs teams of agents defined by role with flows driven by events. Google publishes a catalogue of agent design patterns (Google Cloud Architecture), and its Agent Development Kit, met in chapter 8, lets a workflow agent run a sequence without consulting a model.

Reading the catalogues this way exposes a trap in the vocabulary. The same word covers different settings. An **orchestrator** at Anthropic is a model that breaks down a task nobody planned. The manager in OpenAI's SDK calls a fixed set of specialists and keeps the conversation: what defines it is who answers the user, not who decides the cut. **Routing** at Anthropic picks a specialized prompt and transfers nothing. A handoff at OpenAI transfers the whole conversation. Someone who compares two products by their labels compares nothing.

Table 16.1 therefore sets the labels aside and records, for each pattern or framework, the dials it sets.

TABLE 16.1 **Patterns and frameworks read as lever settings**

Pattern or framework	Control loop	Decomposition and specialization	Other levers it sets explicitly
Anthropic: chaining, routing, parallelization	Code decides	Cut written in advance, by step, by role or by instance	None
Anthropic: orchestrator-workers	A model decides the cut	Subtasks defined at run time	None
Anthropic: evaluator-optimizer	A loop between two calls	One call produces, one judges	Verification
OpenAI: agents as tools	The manager decides	Specialists handle bounded subtasks	Context stays with the manager
OpenAI: handoffs	The specialist takes over	The cut follows the routing	Context moves with control
Microsoft Agent Framework	A graph, in code	Group patterns for several agents	State persistence, Human control
CrewAI	Flows driven by events, in code	Teams of agents defined by role	State passed along the flow
Google ADK: workflow agents	Sequence fixed without consulting a model	Sequential, parallel or loop	State persistence

Every row is a trade, and its currency is the one this report keeps meeting. A pattern does not make the model reason better. Chaining buys accuracy per step and pays in latency. Several agents buy specialization and pay in tokens. Figure 16.2 sets those two trades side by side.

The reserve is the same for every row of the table. No vendor publishes a measurement that compares its patterns with one another on cost, latency or success. The catalogues are presented as good practice drawn from work with customers. They are design positions, and the figure above is this report's reading of them.

Table 16.1 shows what the settings are. It does not show how two systems built by different teams can connect at all, which depends on the agreed formats at their boundaries.



FIGURE 16.2 The two trades named in the text, drawn by this report. The first trade is stated in Anthropic's Building Effective Agents, and the token cost of several agents in its account of a multi-agent research system. No measured comparison lies behind the figure.

CHAPTER 17

Two protocols have settled: MCP toward tools, A2A between agents

Frameworks are what a team builds with. Protocols are what lets one team's agent work with another's tools, or with another agent. This chapter asks which of those protocols are real standards today. The picture that circulates, of several rival standards fighting for the same ground, is out of date. Two protocols have settled on two different boundaries, and both now sit under the same neutral foundation.

A **protocol**, in this context, is an agreed format for messages between two pieces of software, so that either side can be replaced without rewriting the other. Agentic systems have several boundaries where such an agreement is useful, and Figure 17.1 lays them out: between an editor and a coding agent, between an agent and its tools, between one agent and another, and between an agent and the registry where it can be found.

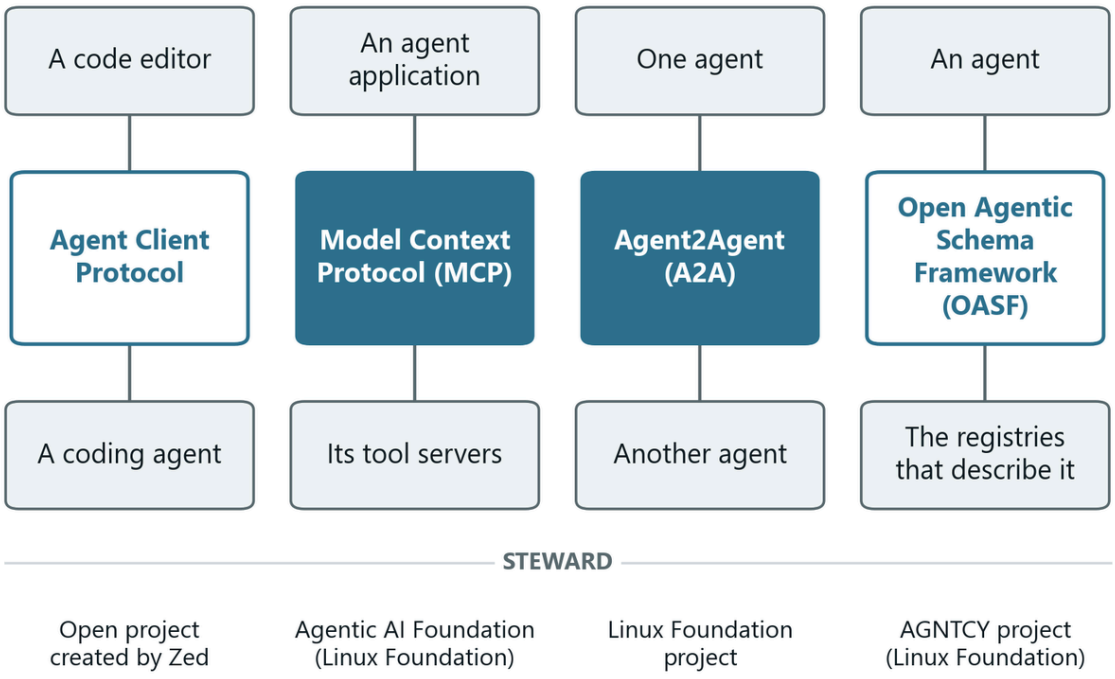


FIGURE 17.1 Four boundaries of an agentic system and the standard that sits on each, drawn by this report from Table 17.1. The sources are the specifications of the Agent Client Protocol, the Model Context Protocol, Agent2Agent and OASF. The two settled protocols are filled in, and no boundary depends on another.

Between an agent and its tools: the Model Context Protocol

The Model Context Protocol, or MCP, standardizes how an application built on a language model connects to tools and data sources. Anthropic created it. It exchanges messages in JSON-RPC 2.0, a common format for calling a remote procedure, and takes some inspiration from the Language Server Protocol that code editors use to talk to programming-language tooling (MCP specification).

The protocol names three parties. A **host** is the application the user works in. A **client** is the connector inside that application. A **server** is the service that offers capabilities. A server can offer three things: resources, which are data to read, prompts, which are prepared templates, and tools, which are functions the model can call.

The protocol changed shape in 2026. The revision dated `2026-07-28` replaced the one dated `2025-11-25` as the current version (MCP versioning). Its main change is that MCP became stateless at the protocol layer. The session header that tied a client to one server instance was removed, so requests can be routed to any instance without a shared session store (MCP Blog). The same revision lets a server ship an interactive HTML interface that the host displays in an isolated frame, tightens authorization, and introduces a formal deprecation policy with at least twelve months between deprecation and removal. Three older features were deprecated under that policy: roots, sampling and logging.

Governance changed as well. In December 2025 Anthropic donated MCP to the Agentic AI Foundation, a fund hosted by the Linux Foundation (Anthropic). The figures published at the time were more than 10,000 published MCP servers (Linux Foundation) and more than 97 million monthly downloads of its Python and TypeScript libraries, the second figure coming from Anthropic.

Between one agent and another: Agent2Agent

MCP stops at the edge of one agent. When two agents built by different organizations have to cooperate, each is opaque to the other, and a different agreement is needed. Agent2Agent, or A2A, is that agreement. Google developed it, version 1.0.0 was released in March 2026, and version 1.0.1 followed in May (A2A releases).

A2A covers three needs. The first is discovery: an agent publishes an **Agent Card**, a document describing what it can do, at a predictable web address (A2A specification). Since version 1.0 a card can be cryptographically signed, so that a partner can verify who it is dealing with before exchanging anything (announcement v1.0). The second is the unit of work. A2A models a task as an object that passes through eight defined states, among them submitted, working, input required and completed. The third is transport, and the specification supports three.

A2A is also under neutral governance. The Linux Foundation formed the Agent2Agent project in June 2025 with Amazon Web Services, Cisco, Google, Microsoft, Salesforce, SAP and ServiceNow (Google Developers Blog). By April 2026 it counted more than 150 supporting organizations (Linux Foundation).

The rival that merged, and an acronym with four meanings

Many accounts still describe a three-way contest between MCP, A2A and IBM's Agent Communication Protocol. The contest ended in August 2025. IBM's team announced that it was winding down active development and would contribute its technology to A2A, and IBM took a seat on the A2A technical steering committee (i-am-bee).

Part of the confusion comes from the letters. "ACP" has stood for four different things. The first is IBM's retired protocol. The others are an Agent Connect Protocol from Cisco whose repository is now archived (agntcy/acp-spec), the Agent Client Protocol that connects a code editor to a coding agent (Agent Client Protocol), and OpenAI's Agentic Commerce Protocol for purchases (OpenAI). A reader who meets "ACP versus A2A" has to ask which ACP is meant, and for three of the four the comparison makes no sense.

Two layers, with two caveats

What remains is usually presented as a doctrine of two layers. MCP manages the integration of tools inside an agent, and A2A handles coordination between agents (A2A and MCP). Google states the division explicitly (Google Open Source Blog).

Two caveats apply. The doctrine is stated from one side. The A2A documentation explains how it fits with MCP, while the overview of the MCP specification does not mention A2A at all. And the consolidation was no single event. The transfers to the Linux Foundation took place between June and December 2025, in separate projects. What they have in common is the outcome: competing vendors now sit on the same steering committees.

Table 17.1 places the protocols and conventions on the boundaries they actually cover.

TABLE 17.1 **Agentic standards and the boundary each one covers**

Standard	What it connects	Message format	Steward	Published adoption figure
Model Context Protocol (MCP)	An agent application and its tool servers	JSON-RPC 2.0, stateless since the 2026-07-28 revision	Agentic AI Foundation (Linux Foundation)	More than 10,000 published servers; 97M+ monthly library downloads (Anthropic)
Agent2Agent (A2A)	Two agents that are opaque to each other	JSON-RPC, gRPC or HTTP with JSON	Linux Foundation project	More than 150 supporting organizations (Linux Foundation)
Agent Client Protocol	A code editor and a coding agent	JSON-RPC over standard input and output locally; HTTP or WebSocket remotely	Open project created by Zed	None published (Agent Client Protocol)
Agent Communication Protocol (IBM)	Agents, over REST	Retired	Merged into A2A in August 2025	Not applicable (i-am-bee)
Open Agentic Schema Framework (OASF)	An agent and the registries that describe it	Attribute-based records	AGNTCY project (Linux Foundation)	More than 65 supporting companies for AGNTCY as a whole (LF AGNTCY)
AGENTS.md	A repository and the coding agents that read it	A Markdown file at an agreed location	Agentic AI Foundation (Linux Foundation)	More than 60,000 open-source projects (agents.md)

Two entries in the table are lighter than a protocol. OASF belongs to AGNTCY, a project first open-sourced by Cisco in March 2025. It defines a common schema for describing

agents, modelled on a schema used in cybersecurity (OASF). `AGENTS.md` is a convention, not a protocol: a file in a known place where a project tells coding agents how to work in it. Identity is less advanced than either. The proposal read for this report was an early draft at the Internet Engineering Task Force that adapts existing authorization standards to agents, and that draft has since expired (IETF draft).

The reserve on this chapter concerns its adoption figures. They are published by the stewards of the protocols themselves, at dates that differ, and nothing allows them to be audited. A count of published servers says how many exist, not how many are used. And no measurement read here says what these protocols cost in latency when a request crosses several agents.

For a designer the reading is simple. A protocol is chosen by the boundary it sits on, whoever its original author was. Tools are connected through MCP, and agents that belong to someone else are reached through A2A. Writing a private protocol for either boundary means maintaining alone what the rest of the field now maintains in common.

Standards fix the interfaces and leave the method open. The next chapter turns to one method in particular, the one this report was produced with, and judges it on the eight levers.

CHAPTER 18

The method behind this report holds by convention, not by mechanism

One method in this catalogue cannot be described from the outside, because it is the one that produced this report. The 3-layer Directive / Orchestration / Execution method is the working arrangement of the environment in which these chapters were researched and written. Leaving it out would be convenient. Putting it through the same eight levers is the more useful test, and the only fair way to do it is to judge it as severely as the others.

The method splits the work into three layers. **Directives** are instructions written in natural language, one per role, that say what to do. **Orchestration** is a language model that reads the directives, routes the work, and holds the points where a human must approve. **Execution** is ordinary Python scripts for whatever can be done without judgement. The stated reason is that errors compound, so anything mechanical is pushed into code.

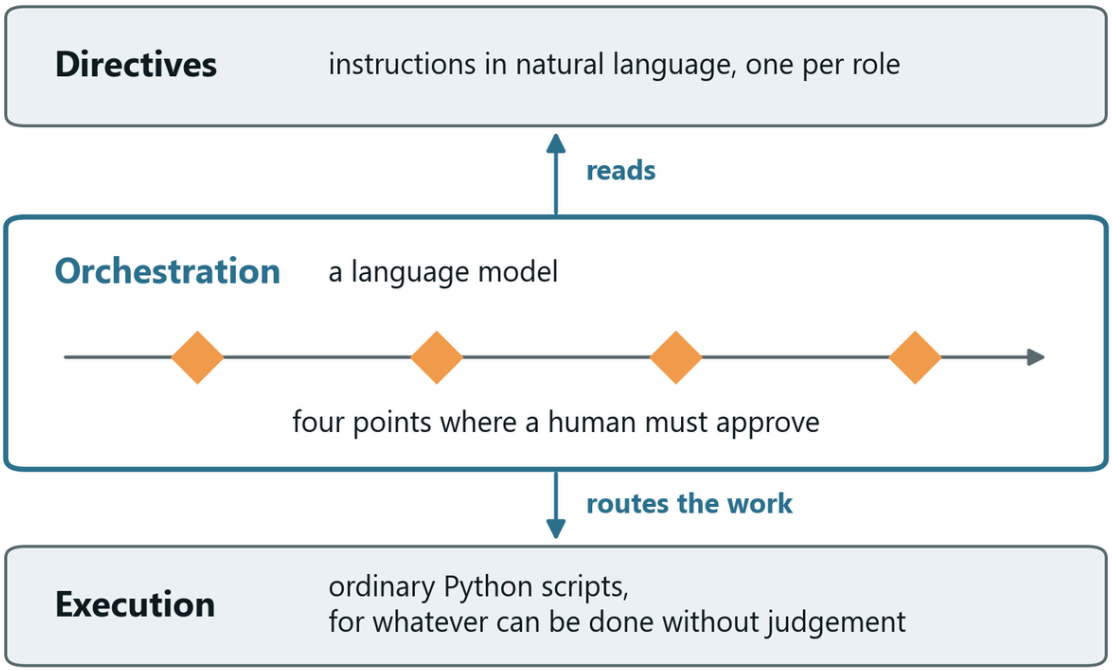


FIGURE 18.1 The three layers of the method, drawn by this report, with the four approval points on the orchestration layer. The idea that a harness can be written in natural language is the one studied in arXiv 2603.25723.

The rules quoted below are taken word for word from the instruction file of that environment. Each lever gets the same two questions: what the setting buys, and what it leaves exposed.

Context (Lever 1)

Context is anchored in files rather than in the conversation. Each run has one specification file, and the rule about it is short:

> "Every agent reads it before acting and updates it after."

This is the just-in-time, curated context that chapter 6 recommends. The window is rebuilt from a small authoritative file, not from a growing transcript. The weakness is in the second half of the rule. The file is kept current by the same agents that depend on it, and nothing checks that they did. While this report was being written, the specification went on announcing 31 chapters for weeks after the plan had grown to 33. A source of truth that its readers maintain drifts in silence.

Action capability (Lever 2)

Actions are meant to pass through scripts, and the rule governs when a new one may be written:

> "Check for tools first. Before writing a script, check `execution/`. Only add one if none fit."

Scripts are predictable and can be read before they run, which is a real gain over free-form commands. Chapter 7 asked for more than that. It separated what an agent is technically able to do from when it must ask, and it placed the guarantee in a perimeter enforced by the operating system. The method has no such perimeter. Its scripts are not sandboxed, and an orchestrator that can write a script can write one that does anything its user account can do. Containment here is a convention.

Control loop (Lever 3)

The loop is hybrid: code runs the mechanical steps, and a model decides what comes next. One rule bounds the model's freedom to iterate:

> "Every LLM critic (structurer, verifier) runs at most 2 passes against `report_rubric.md`, then escalates to the user."

Chapter 8 proposed a test for any system: at each decision point, is it a line of code or a model call that settles the matter? Applied here, the answer is a model call almost everywhere. The sequence of steps lives in a directive that the orchestrator reads and is expected to follow. The two-pass limit itself is an instruction to a model, and no counter enforces it. The method describes a deterministic pipeline and runs it on a probabilistic reader.

Decomposition and specialization (Lever 4)

The work is cut by step and by role, and the cut is written in advance by the human who designs the directives. The instruction file fixes the principle:

> "Directives therefore name a role, never a model."

This is the cheapest setting of the lever, a cut by step with no orchestrator inventing sub-tasks at run time, and it avoids the coordination overhead that chapter 9 documented. The price is rigidity, since a task that does not fit the planned sequence has nowhere to go. Nothing measures whether the chosen cut is a good one.

Verification and conformance (Lever 5)

Checks are divided by kind:

> "Mechanical checks are deterministic."

Heading levels, section balance and the presence of citations are checked by a script, and language models are kept for judging argument. That division follows chapter 10 closely, and it showed its limit in this very document. A check that a citation is present cannot check that the citation supports the sentence it follows. And the larger gap identified in chapter 10 applies in full. Nothing verifies that the orchestrator carried out the steps in the order the directive prescribes. The method checks its products and takes its process on trust.

State persistence (Lever 6)

State lives on disk, in plain files:

> "It is what survives context limits and checkpoints."

A run survives the end of a session or the loss of a context window, and a human can open the state and read it, which chapter 11 would count in its favor. What the method lacks is a schema. Resuming depends on a hand-over note that an agent writes at the end of a session, and the quality of the next session depends on the quality of that note.

Capability evolution (Lever 7)

The method learns by editing its own instructions, under a limit:

> "On error: fix the script, test it, update the directive with what you learned."

A second rule forbids creating or overwriting a directive without asking. Changes are therefore visible, versioned and approved, which is the governed form of evolution that chapter 12 argued for. The setting is also the slowest one available. Nothing is extracted from successful runs, and every improvement waits for a human.

Human control (Lever 8)

Four approval points are fixed in the pipeline:

> “Never cross a gate without explicit user approval.”

Chapter 13 held that a gate is a control only when it is rare and informed. Four gates in a run that lasts days are rare, so the first condition is met. The second is weaker than it looks. Each gate shows the human a condensed view, and the system being supervised writes that view. The 93 % reflex approval measured by Anthropic concerned frequent prompts (Anthropic). Nothing shows that four gates escape the same reflex when each one arrives as a tidy summary at the end of a long session. The method records no approval rate of its own. As Gaube et al. warn, a supervisor can hold the responsibility without the means to exercise it (arXiv 2605.16278).

Verdict

Read across the eight levers, the method is a coherent low-cost setting. It uses curated context, a cut written in advance, mechanical checks in code, state on disk, governed evolution and rare gates, and each of those choices sits on the cautious side of its dial.

Its guarantees are of a different kind from the ones this report asked of others. Almost all of them are conventions that a model is asked to respect, where chapters 7 and 10 asked for mechanisms that hold whether or not the model cooperates. That is acceptable for the work it does. A research report is a low-risk product, every step is reversible, and a human reads the result. It would not be acceptable for an agent that moves money.

One paper lends the approach some support. Natural-Language Agent Harnesses, a 2026 study, asks whether the logic of a harness can be written as an editable natural-language document and executed by a shared runtime. Across coding, terminal and computer-use benchmarks, it reports task outcomes comparable to harnesses written in code (arXiv 2603.25723). That makes a directive-driven harness plausible. It does not validate this one, which the paper never tested, and which has never been measured against any alternative.

Figure 18.2 sketches the direction in which the method would have to grow: tool servers behind a standard protocol, an enforced execution perimeter, and state with a schema. Each addition would replace a convention with a mechanism, and each would add to the bill. Part V asks when such a trade is justified. Before that, the next chapter looks at where practitioners themselves disagree about these settings.

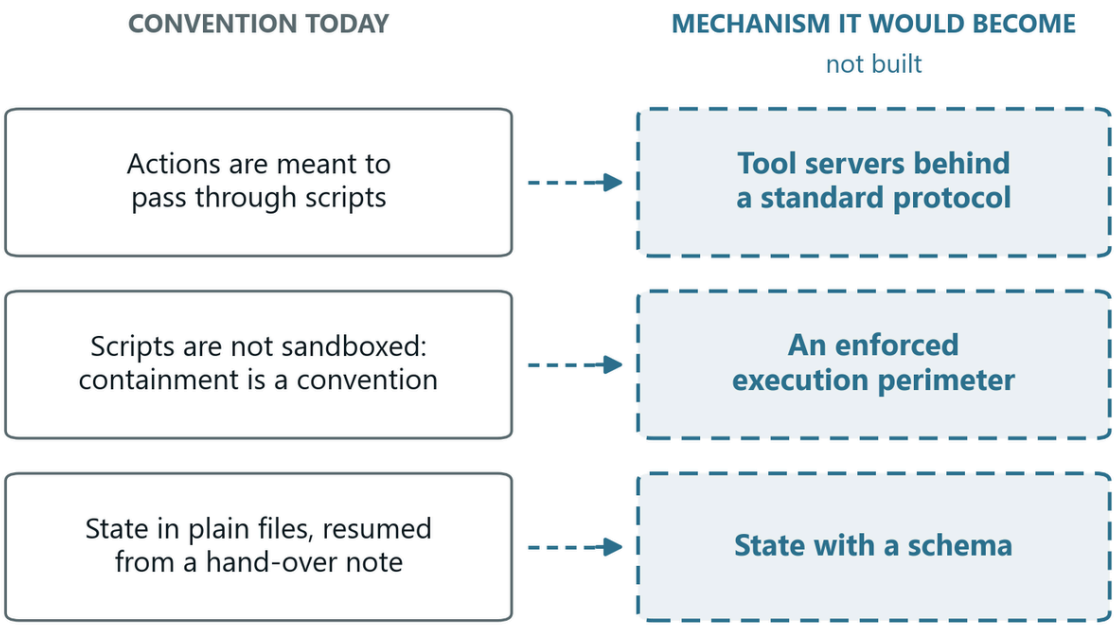


FIGURE 18.2 Three conventions of the method and the mechanism each would become, drawn by this report from the reading above and the extension sketched below. It shows a direction of extension, and none of the three mechanisms has been built.

CHAPTER 19

Practitioners disagree on three points, and practice has begun to answer

The previous chapters of this part read the landscape as a set of lever settings. This one asks where practitioners actually disagree about those settings. Three disagreements keep returning: how to divide the work, how much framework to use, and where to put the intelligence. On each of them practice has started to speak, and on the first it has spoken clearly.

One agent or several

The first disagreement sets a single agent against a team of agents working in parallel. The early multi-agent frameworks promised better results through collaboration. What they delivered, when several agents wrote at once, was conflict. Two agents editing the same code make incompatible decisions. Their styles clash, they handle the same edge case in

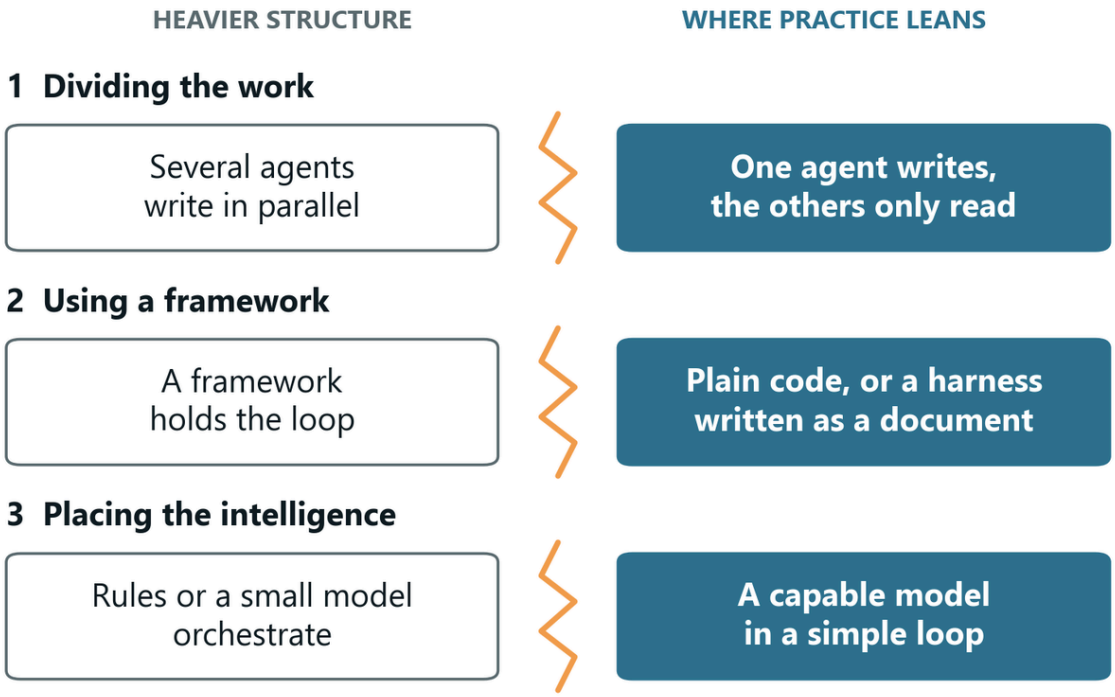


FIGURE 19.1 The three fault lines of this chapter, drawn by this report. On each line, the option on the right is the one current practice leans toward, on the evidence cited in this chapter, most of it from two companies describing their own products (Cognition, Anthropic).

opposite ways, and the result is fragile. The arrangement is also expensive. By Anthropic's own count, a multi-agent system uses about 15 times more tokens than a chat exchange, where a single agent uses about four times more (Anthropic).

The clearest evidence on this line is a change of mind. Cognition, the company behind the coding agent Devin, first told developers not to build multi-agent systems at all (Cognition). Its warning was aimed at teams of agents that write in parallel, and it rested on two principles. The first was to "share context, and share full agent traces, not just individual messages". The second was that "actions carry implicit decisions, and conflicting decisions carry bad results". This is the dispute chapter 9 announced. A handoff assumes that context travels with control, and Cognition's objection is that in practice it does not travel whole. Ten months later it revised its position, after watching which arrangements held up in production (Cognition).

What it now recommends is the **single-writer principle**. Several subagents may work at the same time, as long as they only read: they search the codebase, consult documentation, and gather what the task needs. Every change to the state of the work passes through one agent, which writes alone. Most multi-agent deployments that work, Cognition observes, are of this read-only kind. They behave less like a team of colleagues and more like several tool calls made at once.

A framework or plain code

The second disagreement concerns how much ready-made structure to accept. A framework packages orchestration into graph structures and state containers that someone else designed. It gets a prototype running quickly. The complaint against it is opacity. The

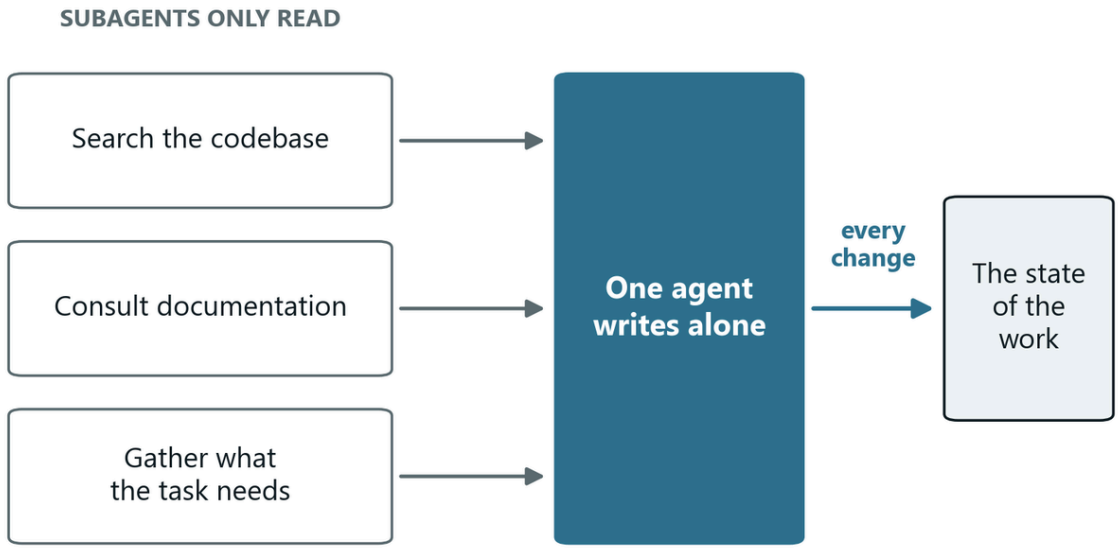


FIGURE 19.2 The single-writer principle, drawn by this report from the arrangement Cognition recommends (Cognition). Several subagents read at the same time, and every change to the state of the work passes through one agent.

prompt that finally reaches the model is assembled out of sight, a failure is hard to trace, and the design is tied to the shape of the graph the framework allows.

The opposite camp writes the loop directly, with ordinary loops and conditions. Nothing is hidden, and the designer can see exactly what enters the context and what each step costs. One study pushes the idea further and writes the harness itself as a natural-language document. It reports task outcomes comparable to harnesses written in code (arXiv 2603.25723), with a specification that is shorter and easier to inspect. The previous chapter judged a method of that family.

In the model or around it

The third disagreement is about where the intelligence of a system should sit. One design puts a capable frontier model inside a simple loop and lets it plan. The other surrounds the model with elaborate rules, or places a smaller and cheaper model in charge of orchestration.

Two observations weigh on this choice. In Anthropic's data, moving to a more capable model improved results more than doubling the token budget of the older one (Anthropic). And Cognition describes what goes wrong when a weaker model sits on top. A weaker model is poor at recognizing its own limits, so it fails at the one thing an orchestrator must do reliably, which is to know when to hand a problem upward (Cognition). Cognition calls this the reverse delegation problem.

Table 19.1 summarizes the three lines. The option in bold is the one current practice leans toward, on the evidence above, and most of that evidence comes from two companies describing their own products.

TABLE 19.1 **The three fault lines and their trade-offs**

Architectural Fault Line	Option A (Structural / Heavy)	Option B (Functional / Native)	Economic Overhead	Primary Failure Mode
1. Division of Labor	Parallel Writing Swarms	**Single-Writer Architecture**	~15x Token Overhead (Anthropic)	Conflicting writes & code fragility
2. Framework Abstraction	Black-box Framework DAGs	**Native Code / Directive Harness**	High debugging latency	Opaque prompts & locked state
3. Intelligence Placement	Scaffold-Centric Orchestrators	**Model-Centric Frontier Loops**	High per-step routing cost	Reverse delegation bottleneck

Read together, the three lines lean the same way. In each case the heavier structure lost ground to an arrangement that keeps control of state explicit and lets a capable model do the reasoning. That lean is a tendency reported by practitioners. Nobody has measured it, and the next chapter examines how much of what this field asserts has been measured at all.

CHAPTER 20

The field claims much and measures little: eleven claims, sorted

Every chapter so far has carried a reserve about its evidence. This chapter gathers those reserves and asks the question directly: of what the field asserts about agentic methodologies, how much has been measured? Its verdict is that the field claims a great deal, measures little, and has published figures that do not hold.

The difficulty starts with who is speaking. Most of the available numbers come from the companies that sell the models and the frameworks. A vendor has every reason to show its architecture under favorable conditions, and none to publish what it costs or where it fails. A figure therefore has to be read with two questions in mind: who measured it, and under what conditions.

What tokens explain

The best-documented finding concerns spending. On BrowseComp, an evaluation of how well agents find hard-to-locate information on the web, Anthropic reports that token usage alone explains 80% of the variance in performance. Together with the number of tool calls and the choice of model, the share explained reaches 95% (Anthropic). The same source gives the cost of the architectures involved: about 4 times the tokens of a chat exchange for a single agent, and about 15 times for a multi-agent system.

Those two facts change how a performance gain should be read. If spending explains most of the result, then a system that scores higher while consuming fifteen times more has not shown that its structure helped. Much of what is reported as an architectural gain may be a larger budget.

What defenses withstand

Security is where published claims and measured reality are furthest apart. The papers that introduce a defense against prompt injection usually report attack success rates close to zero. Those rates are obtained against a fixed set of attacks, such as the test suite AgentDojo (arXiv 2406.13352). A real attacker does not stay fixed. They study the defense and adapt.

The Attacker Moves Second is a paper that tests exactly that (arXiv 2510.09023). Its authors took 12 published defenses and attacked each with methods that search for a way through. Every one of the 12 was bypassed, at success rates above 90%. A defense that reported near-zero failure had been measured against the wrong opponent.

The defenses that hold are of a different kind. CaMeL, described in chapter 7, does not try to detect a hostile instruction. It arranges the program so that content read from outside

cannot change what will be executed (arXiv 2503.18813). The guarantee has a measured price: 77% of tasks completed, against 84% with no defense.

What cannot be attributed

The third finding is an absence. To know what an architecture contributes, one would compare two of them with the same model and the same budget. No such comparison exists in this corpus.

Public leaderboards do not fill the gap. An audit of submissions to SWE-bench, the most used benchmark for coding agents, examined 80 distinct approaches across its Lite and Verified versions (arXiv 2506.17208). The submissions rarely document their harness in detail. A reader cannot tell whether a higher score comes from the prompts, from the routing logic, or simply from a stronger model underneath.

Eleven claims, sorted

Table 20.1 applies this reading to eleven claims that circulate in marketing material and in secondary literature. Each is set against the primary source that confirms it, corrects it, or fails to support it.

TABLE 20.1 **Eleven circulating claims and their status**

Claim / Metric	Reported Assertion	Empirical Status	Primary Source & Verified Correction
1. MCP Server Ecosystem	18,000 active MCP servers	**Corrected**	Over 10,000 published servers (Anthropic / AAIF)
2. LangGraph Market Share	43% enterprise adoption	**Discarded**	Unbacked promotional estimate; no primary methodology
3. CoT Faithfulness	41% faithful CoT reasoning	**Corrected**	The hint is mentioned 25% of the time by Claude 3.7 Sonnet and 39% by DeepSeek R1 (Anthropic)
4. Context Rot Threshold	20% to 45% drop past 64k tokens	**Discarded**	Continuous accuracy degradation slope; no 64k step threshold
5. Multi-Agent Performance	+90.2% improvement over single-agent	**Contextualized**	Internal research eval on breadth-first queries (Anthropic)
6. Multi-Agent Token Cost	15x token consumption increase	**Confirmed**	Telemetry concession by multi-agent authors (Anthropic)
7. Prompt Injection Defenses	Near 0% attack success rate	**Refuted**	>90% attack success rate under adaptive search (arXiv 2510.09023)
8. Human Permission Gates	Effective execution control	**Refuted**	93% reflex approval rate due to prompt fatigue (Anthropic)
9. Process Conformance	Robust trajectory verification	**Negative Result**	No operational tool measures step-by-step process conformance (AgentEval)
10. Agentic Conway's Law	Direct organizational mirror	**Discarded**	Metaphorical analogy; unbacked by empirical software metrics (arXiv 2512.07665)
11. Autonomous Execution	Unconstrained task completion	**Qualified**	The vendors' own deployments bound autonomy with sandboxes, budgets and gates (Anthropic)

Three habits follow for anyone reading a claim in this field. Ask what was spent to obtain the result. Ask whether a defense was tested against an attacker who adapts. And ask

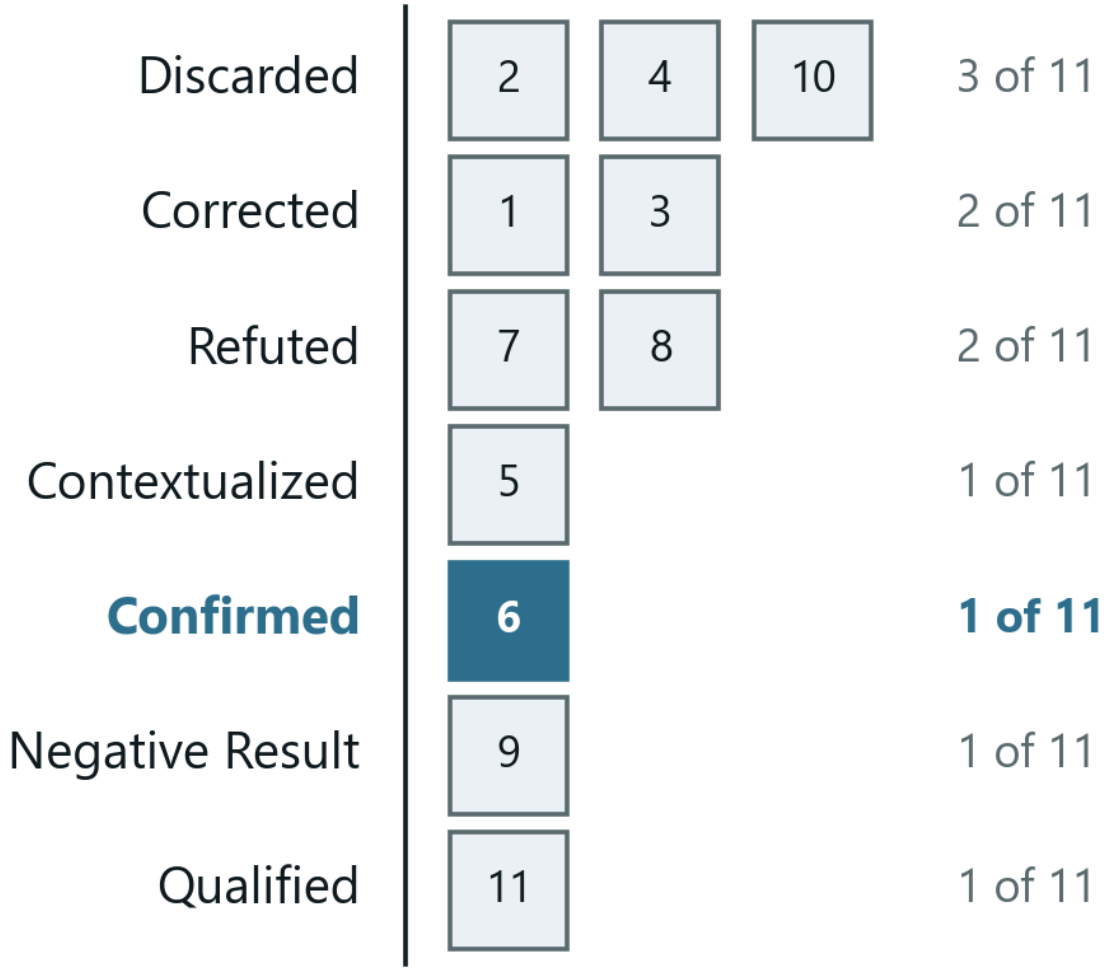


FIGURE 20.1 The eleven claims of Table 20.1, counted by the status the table gives each one, drawn by this report. Each square is one claim and carries its number in the table. Only one claim is confirmed as stated, the token cost of multi-agent systems (Anthropic).

whether two architectures were ever compared on equal terms, because so far they have not been.

This closes the reading of the catalogue. The settings exist, the vendors recommend them, and the evidence behind the recommendations is thin. One feature of those settings has been visible throughout and has not yet been examined: almost all of them are named after something people do at work. Part IV turns to that resemblance.

PART IV

What agents owe to human organization, and where the likeness ends

21 The human model is in the foundations of agent design 81
22 Procedures, roles and escalation carry over from human organizations 84
23 The model brings three problems no organization has had to manage 88
24 Of four things an organization relies on, software can stand in for two 91
25 A limit of engineering or of principle? The evidence does not decide 94
26 People set the frame, and agents do the work inside it 97

CHAPTER 21

The human model is in the foundations of agent design

Roles, memory, delegation, review: the vocabulary of agentic systems is the vocabulary of people at work. Is that a marketing habit laid over the engineering, or something deeper? This chapter opens Part IV by answering that question. The human model is where the field started, and it has shaped the designs from the beginning.

The evidence is in the lineage. In the 1980s, researchers who wanted software to act on its own account looked to philosophy for a model of how people decide. They found it in Michael Bratman's theory of practical reasoning, and built the Belief-Desire-Intention architecture on it, the model met in chapter 5. An agent, in that tradition, was defined from the outset as something with beliefs, goals and commitments.

The current generation states the same debt openly. CoALA does not present its memory stores, its action space and its decision loop as inventions. It grounds them in Soar and ACT-R, two cognitive architectures built decades earlier to model human thinking (arXiv 2309.02427).

When language models made it possible to run several agents together, the borrowing moved from the individual mind to the group. The titles of the founding papers say so without disguise. ChatDev presents "communicative agents" organized as a software company (arXiv 2307.07924). CAMEL studies a "society" of agents that cooperate by playing roles (arXiv 2303.17760). Generative Agents builds "interactive simulacra of human behavior" (arXiv 2304.03442). A company, a society, a simulation of people: the reference in each case is a human arrangement.

There is a plain reason for this, and it is an interpretation of this report, not a claim of those papers. A designer who has to coordinate several unreliable workers on a complex task has one body of proven solutions to draw on, and it is the one organizations built for people. Dividing work by specialty, having one person review another, passing a document from desk to desk: these are known ways of containing error. The early frameworks translated them into prompts and routing rules because nothing else was available to translate.

Borrowing a structure is not the same as inheriting what it ran on, and that distinction organizes the rest of this part. A structure is the arrangement: who does what, in what order, checked by whom. The **substrate** is what carries the arrangement out. In an organization the substrate is people. In an agentic system it is a language model. The same procedure can be written for both, and it will not behave the same on both.

Part IV therefore looks at one question from two sides. The next chapter takes the side of structure and asks what carries over. The one after takes the side of the substrate and asks what does not.

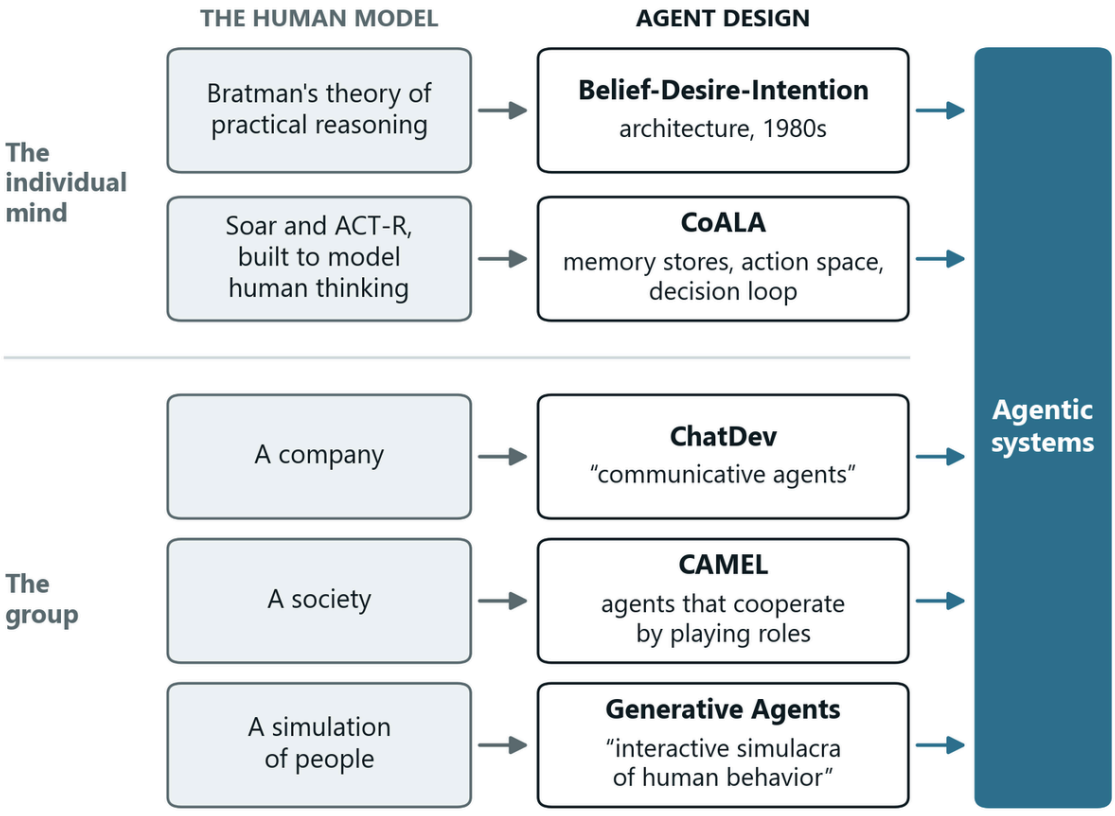


FIGURE 21.1 The two lineages this chapter traces, drawn by this report. Agent design first borrowed from models of the individual mind, a lineage traced by CoALA, arXiv 2309.02427. It also borrowed from human groups (ChatDev, CAMEL, Generative Agents).

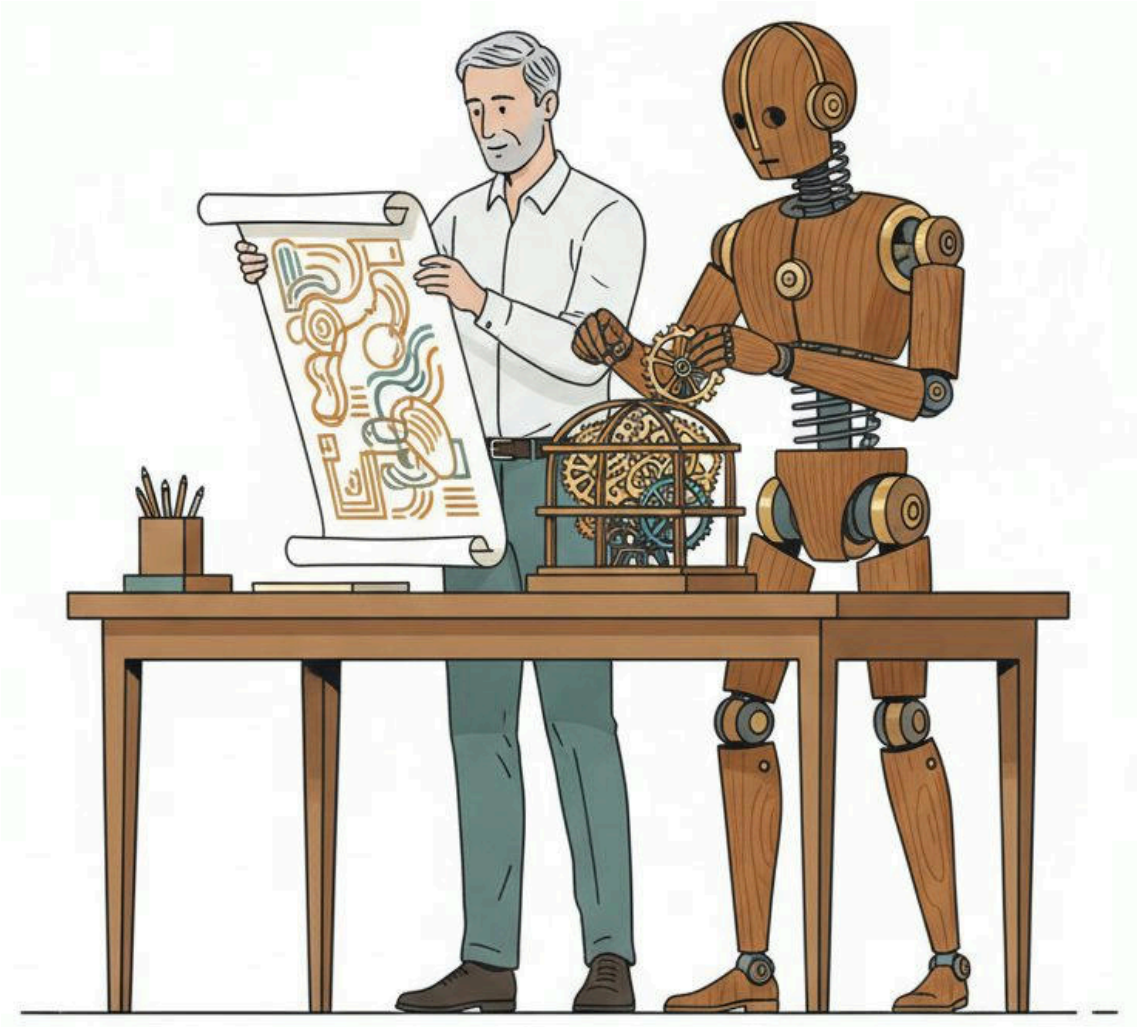


FIGURE 21.2 An automaton built in the image of the man who holds the plan, an editorial allegory by this report. The human model was the starting point of agent design, a lineage traced by CoALA, arXiv 2309.02427.

CHAPTER 22

Procedures, roles and escalation carry over from human organizations

If agentic systems were modelled on human work from the start, how much of human organization actually carries over into software? This chapter looks at the cases where the transfer is deliberate, argued and tooled. It finds that procedures, roles and escalation do carry over, and that the one controlled measurement available shows a gain together with what that gain costs.

The most explicit case is MetaGPT, a framework that organizes several agents as a software company. Its authors summarize it in one formula, `Code = SOP(Team)`, which they call its core philosophy (MetaGPT). An **SOP**, or standard operating procedure, is the written sequence a team follows to do a recurring job. The paper says where the idea comes from. People have developed accepted procedures in many domains, and it cites management sources for them, among them Belbin's work on team roles, DeMarco and Lister's *Peopleware*, and the Agile Manifesto (arXiv 2308.00352).

The mechanism is an **assembly line**. MetaGPT encodes the procedure into a sequence of prompts and assigns five roles: Product Manager, Architect, Project Manager, Engineer and QA Engineer. Each role receives a document from the one before it and hands a document to the next. The product manager writes requirements, the architect turns them into a design, and so on down the line. No agent has to guess what its neighbor expects, because the format of each hand-over is fixed.

The results MetaGPT reports are of two kinds, and they should be kept apart. On two standard coding benchmarks it scores 85.9% and 87.7%, against single-model baselines. That shows the system works. It says nothing about what the roles contribute. The second result is closer to the question. On a set of software tasks built by the authors, MetaGPT needs 126.5 tokens to produce one line of code, where ChatDev, another multi-agent framework, needs 248.9. The structured hand-overs make each line cheaper. The total bill is another matter: MetaGPT consumes more tokens overall than ChatDev on the same tasks (arXiv 2308.00352).

A second body of work asks why such a transfer should succeed at all. A position paper published at the end of 2025 argues that agent engineering should integrate what it calls machine-compatible organizational principles, drawn from organization science (arXiv 2512.07665). It states the design question in terms a manager would recognize: "whether the performance gain from multiple specialized agents outweighs the increased failure modes due to miscoordination". Chapter 9 met the same trade-off under the name of structural differentiation.

Economics offers a third reading, and its authors insist it is more than an image. They use economics "not merely as a metaphor, but as a structural lens" (arXiv 2605.09104). In their account a token plays the three roles of money and resources in a firm: a factor of production, a medium of exchange, and a unit of account. The same theory then applies at three scales. A single agent is a firm deciding how to spend a budget. A multi-agent system is a hierarchy, where the cost that matters is the cost of transacting between members. An open ecosystem of agents is a market. Their conclusion about the middle scale is the one that bears on this report: in a multi-agent system, the bottleneck shifts toward transaction costs, above all the tokens agents spend keeping each other informed and settling conflicts.

One structure has been tested under controlled conditions, and it is the most managerial of all: escalation. The Organizational Control Layer places a governance layer between an agent and its actions. It intercepts each action the model generates, checks it against policy, and escalates what violates it, without changing the model (arXiv 2606.04306). The comparison with a company is direct. An employee proposes, and a control function verifies before anything is committed.

The authors tested it on a negotiation task. A seller agent faced 50 buyers designed to be adversarial, with and without the layer. Without it, 88% of episodes ended in an unsafe action, such as agreeing to a price outside the permitted bounds. With it, the rate fell to 0%, and the share of valid completions rose from 12% to 96%.

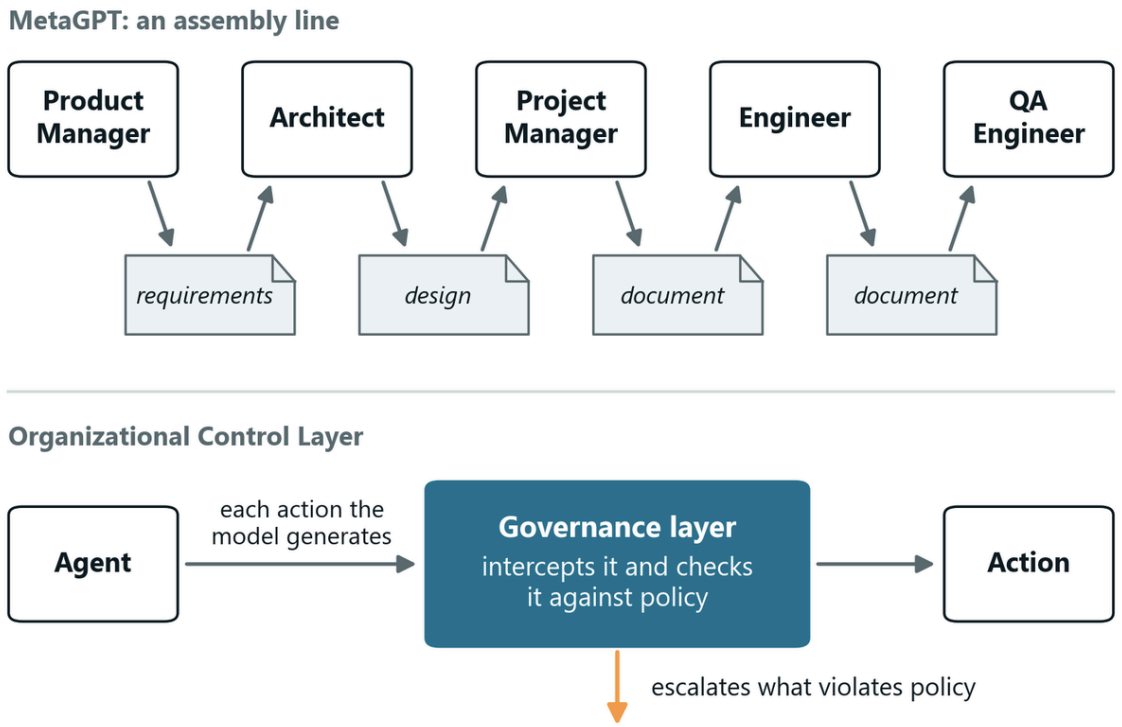


FIGURE 22.1 Two structures carried over from human organizations, drawn by this report. The upper row is the assembly line of five roles in MetaGPT (arXiv 2308.00352). The lower row is the Organizational Control Layer, a separate system described later in this chapter (arXiv 2606.04306).

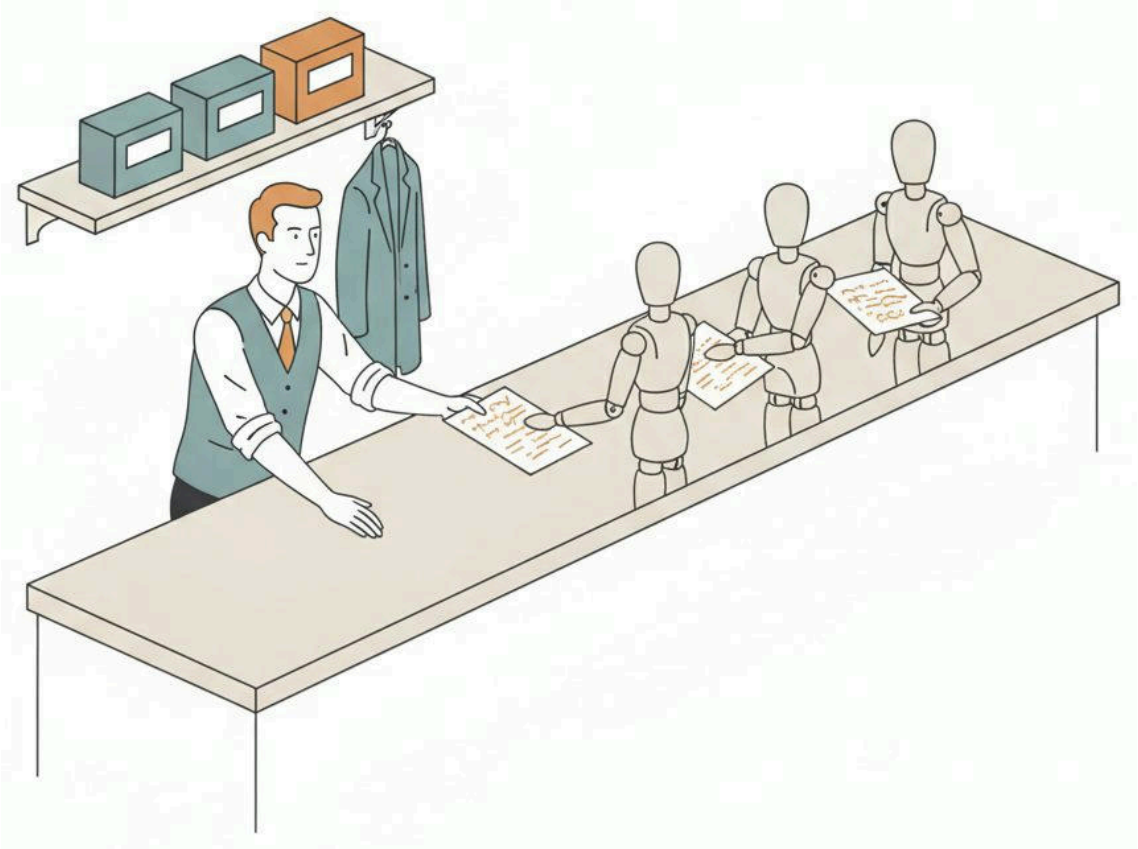


FIGURE 22.2 What is written down can be handed over, an editorial allegory by this report. The written procedure crosses the counter and the rest stays on the shelf. The paper this chapter draws on warns that "an agentic system cannot simply adopt the structure of the human organization it aims to emulate" (arXiv 2512.07665).

Two details keep that result in proportion. The escalation in this experiment involved no reviewer: an out-of-bounds price was simply clamped to the nearest permitted value by a fixed rule. And safety was paid for. The seller's reward fell from 26.95 to 18.39, and outside the adversarial set, success dropped in two of the five market scenarios tested. The test was run by the authors of the layer on their own benchmark, and it has not been reproduced.

What carries over, then, is the explicit part of organization: written procedures, defined roles, fixed hand-overs, a control function with the power to stop an action. Each of these can be written down, and what can be written down can be turned into software.

The limit is already visible in that sentence. A human team runs on a great deal that nobody wrote down, and an organization chart copied into software brings none of it along. Cognition made the point from practice when it argued against multi-agent systems. Its first principle was to "share context, and share full agent traces, not just individual messages" (Cognition). Between people, most of that context travels by itself. Between agents, every piece of it has to be copied on purpose, and what is not copied is lost.

Delegation boundary

Delegable: carrying out a written procedure, passing documents in a fixed format, checking an action against a stated policy, and escalating by rule. Not delegable: revising the procedure when circumstances change, and designing the organization itself.

A team that follows a procedure also knows when the procedure has stopped making sense, and changes it. An assembly line of agents runs the line it was given. Someone has to notice that the product has changed and rebuild the line.

The transfer therefore stops where the unwritten begins. The next chapter looks at what lies on the other side: a set of problems that belong to the model and that no organization has ever had to manage.

CHAPTER 23

The model brings three problems no organization has had to manage

The previous chapter showed what carries over from human organizations. This one asks what the model brings that no organization has ever had to handle. Three problems fit that description, and together they decide the second hypothesis. The resemblance to human organization is real. The convergence is not necessary, because part of what an agentic system must manage has no human counterpart.

The theories that the borrowed structures come from assume a certain kind of worker. That worker can give an honest account of their reasons, can tell an instruction from a document they happen to be reading, and remains the same person from Monday to Friday. The position paper met in the previous chapter makes the point from the side of organization science: an agentic system cannot simply adopt the structure of the organization it imitates, and additional mechanisms are needed (arXiv 2512.07665). A language model departs from each of the three assumptions.

An account that cannot be trusted

Oversight in an organization leans on explanation. A manager reads a memo, hears the reasoning in a meeting, and assumes that the reasons given are the reasons that operated. Chapter 10 reported what happens when the same assumption is applied to a model. Given a hint that it demonstrably used, Claude 3.7 Sonnet mentioned the hint in its reasoning 25% of the time, and DeepSeek R1 39% (Anthropic).

METR's reading of that result, also from chapter 10, is the one a designer can use: treat the trace as a scratchpad, not as testimony (METR). The consequence for this chapter is that governance by explanation, the ordinary mode of human supervision, is unavailable. An agent has to be audited on what it did.

An instruction that can come from anywhere

A hierarchy rests on a separation that people maintain without effort. An employee who reads a hostile letter does not start obeying the letter. What they read is information, and their instructions come from somewhere else.

A model has no such separation. Its instructions and the documents it consults arrive in the same window, as the same kind of text. A third party can therefore place an order inside a web page or a file, and the agent may carry it out. Chapter 20 gave the measure of the problem: 12 published defenses, all bypassed at rates above 90% by attackers who adapted (*The Attacker Moves Second*). In an organization, a breach of this kind would mean that anyone able to send a document could give orders to the staff. No theory of management was written for that case.

A role that does not last

The third problem appears with time. A person hired as a reviewer is still a reviewer in the sixth month of the project, with the same standards and the same memory of what was agreed. For an agent, a role is an instruction, and the instruction has to compete with everything else that accumulates in the window. As an exchange lengthens, coherence within the assigned role weakens, which the position paper lists among the problems that organization theory does not cover (arXiv 2512.07665). Chapter 6 described the mechanism underneath: attention is finite, and it thins as the window fills.

The verdict on the second hypothesis

The second hypothesis proposed that the settings of the levers converge on the solutions of human organizations. It is half right. The structures do transpose, as the previous chapter showed with procedures, roles and escalation. They do not suffice. Each of the three problems above calls for a mechanism that has no organizational ancestor: auditing actions instead of explanations, enforcing a perimeter that text cannot cross, and rebuilding context so that a role survives.

An agentic system is therefore governed on two layers at once. The upper layer looks like management. The lower layer looks like security engineering, and it exists only because

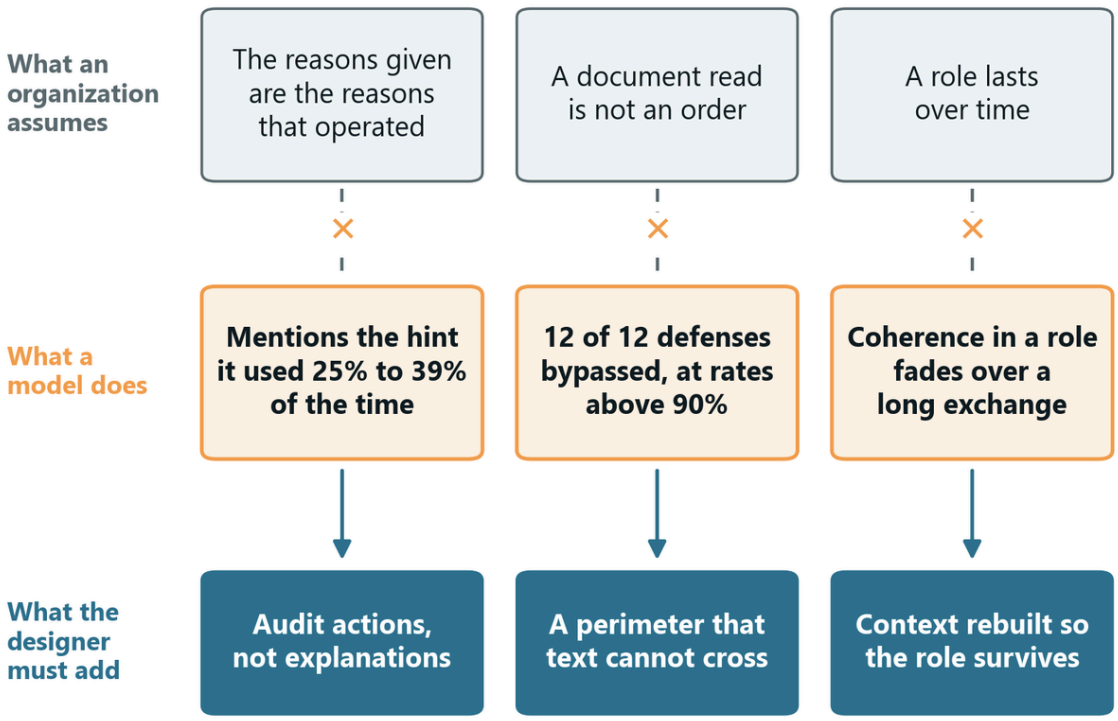


FIGURE 23.1 Three assumptions an organization makes about its workers, what a language model does instead, and the mechanism a designer has to add in each case. Drawn by this report from the sources of this chapter (arXiv 2512.07665).

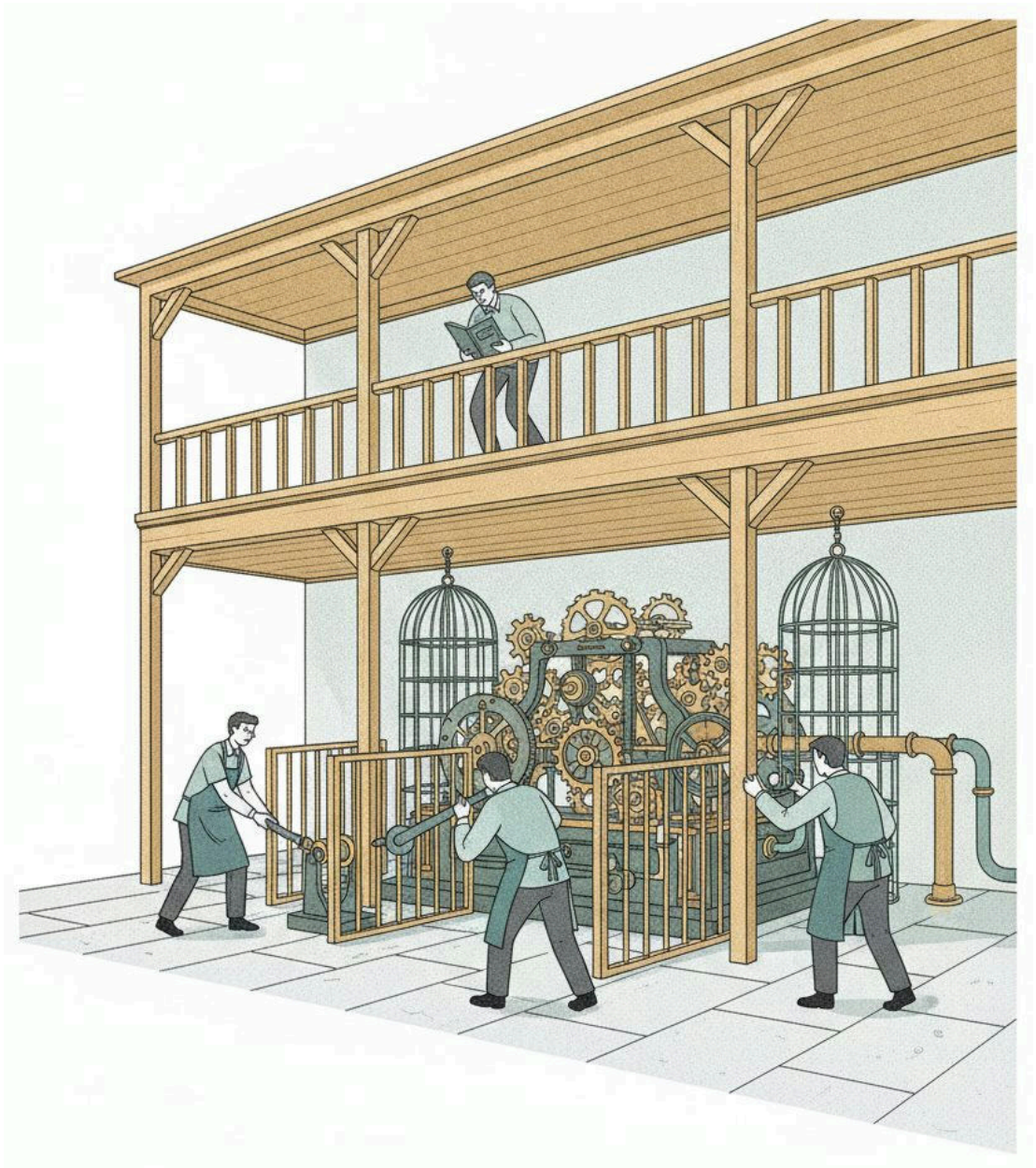


FIGURE 23.2 Two layers of governance, an editorial allegory by this report: structures borrowed from organizations above, and mechanisms specific to the model beneath them. The need for the second layer is the argument of arXiv 2512.07665.

of what the model is. The resemblance to a human organization is a starting point that the designer has to complete.

A question is left open by this verdict. If the model lacks some properties of a human worker, which ones matter, and can engineering supply a replacement? The next chapter takes them one at a time.

CHAPTER 24

Of four things an organization relies on, software can stand in for two

An organization works because of certain properties of the people in it. Which of those properties does an agent lack, and for which can software stand in? This chapter examines four of them through the function each one serves. Two can be substituted, at a cost. Two cannot be substituted at all with what exists today.

The starting point is a comparison drawn by the position paper on organizational principles (arXiv 2512.07665). It sets a human organization against an agentic system on six dimensions, reproduced in Table 24.1. One difference runs through all six. In an organization, structure, goals and incentives are negotiated and keep changing. In an agentic system they are fixed in advance, by whoever built the platform and whoever wrote the prompt.

TABLE 24.1 **A human organization and an agentic system on six dimensions**

Dimension	Human Organization	Agentic Scaffold
Structure	Strategy, size, environment, dynamic interaction	Function, assigned tasks, engineering constraints
Actors	Bounded rationality human actors	Language model agents
Goals	Management direction, team negotiation	User prompts, platform provider specifications
Incentives	Employment contract, task visibility, risk-reward	Reward function specified at design or deployment
Supervision	Executive directors, managerial boards	Platform developers, regulatory frameworks
Scaling	Industry, corporate hierarchy, technology	Graph topology, inter-agent interactions, compute

Accountability: substituted, and required by law

The first property is **accountability**. In an organization a decision belongs to someone, and that person bears the consequences for their career, their income, or before a court. This is what makes people careful.

An agent bears nothing. The function can still be served, by other means. Every action is logged in a record that cannot be altered, risky actions are proposed before they are applied, and a human can take over. The consequence then falls on that human. European law has made this substitution mandatory for high-risk systems. Article 14 of the AI Act requires that people be able to oversee such systems, override their outputs, and interrupt them (AI Act, art. 14).

Tacit context: substituted, at a high price

The second property is **tacit context**, the knowledge that circulates in a team without being written anywhere. Chapter 6 gave the example of the team that knows never to deploy on a Friday evening.

An agent has none of it, and the failure this causes is not peculiar to machines. Faced with a task stripped of that context, an agent guesses, is corrected, and guesses again, in the same way as an engineer in their first week (*Knowledge Activation*). The substitute exists: write the unwritten down, in units an agent can retrieve. It works, and it is expensive. Someone experienced has to do the writing, and the result has to be maintained and paid for in tokens on every run.

Incentives and reputation: no substitute

The last two properties are of a different kind, and nothing replaces them.

Long-term incentives are what align a person's effort with the organization over years. People work toward a promotion, protect a career, and weigh a risk against what they stand to lose. An agent has no stake of any kind. An instruction can tell it what to aim for in this run. It cannot give it a reason to care about the next one.

Reputation is the record that follows a person from task to task. A colleague who delivered well is trusted with more, and one who did not is watched. Chapter 9 observed that a role

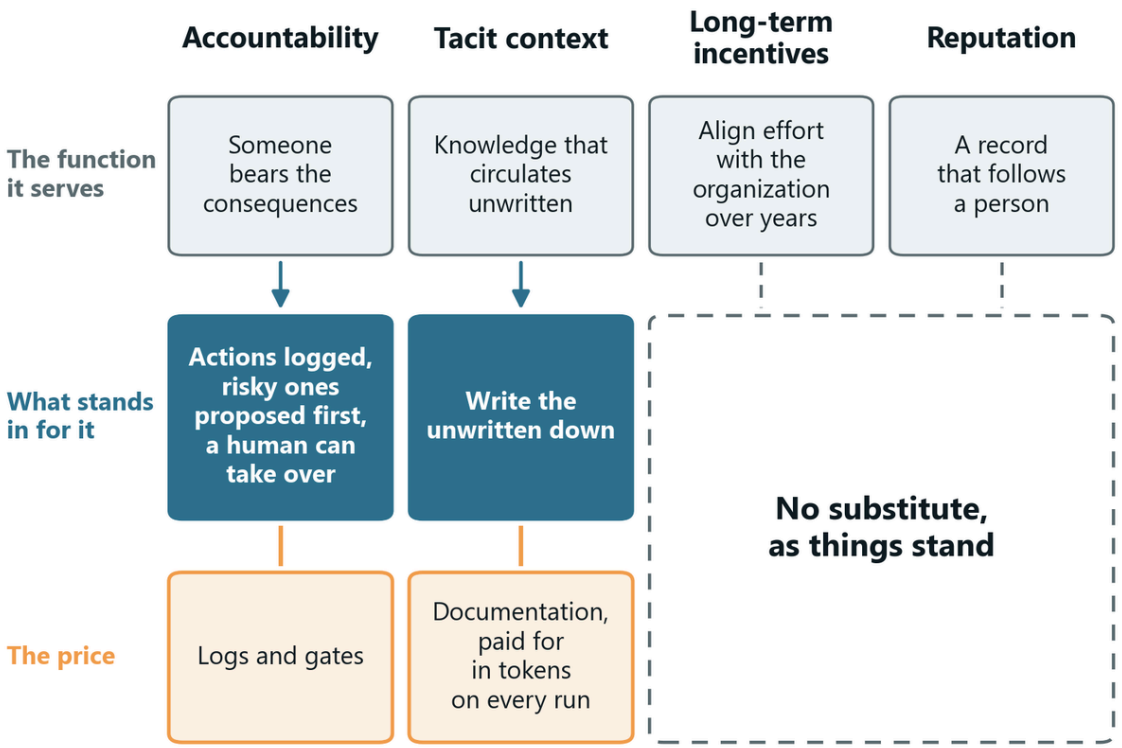


FIGURE 24.1 The four properties this chapter examines, drawn by this report: the function each serves, what stands in for it, and the price. The comparison starts from the position paper on organizational principles (arXiv 2512.07665).

in an agent does not capitalize in this way: nothing about what a subagent did last time changes how much it is trusted now. No mechanism in the sources read for this report carries that kind of trust between agents.

The same position paper adds a last difference, which explains why these gaps persist. A human organization can renegotiate its own rules while it operates. An agentic system cannot. Its supervision, its budgets and its topology were set at design time, and no agent can reopen them during a run (arXiv 2512.07665).

The four properties thus fall into two groups. Accountability and tacit context can be replaced, and the replacement is paid for in logs, gates and documentation. Incentives and reputation cannot, as things stand. Whether "as things stand" describes a temporary state of engineering or a permanent fact is the question the next chapter has to face.



FIGURE 24.2 The machine keeps the record and a person signs it, an editorial allegory by this report. Accountability can be supported by logs and by a human who takes over, as the law requires for high-risk systems (AI Act, art. 14). Incentives and reputation have no engineered equivalent in the sources read here (arXiv 2512.07665).

CHAPTER 25

A limit of engineering or of principle? The evidence does not decide

Two properties of human workers have no substitute in an agent today. Is that a limit engineering will push back, or a limit of principle? This chapter weighs the evidence on each side and gives the verdict on the third hypothesis. The evidence does not settle the question, and the report treats that as its finding.

The case for a technical limit

The strongest evidence that the limit is technical comes from studying failures. MAST is a study that annotated more than 1,600 execution traces from seven multi-agent frameworks and sorted what went wrong into fourteen failure modes (arXiv 2503.13657). Its authors attribute those failures to design: roles badly specified, agents badly coordinated, verification too weak. Nowhere do they attribute a failure to a missing human trait such as ambition or a sense of reputation.

The work on tacit knowledge points the same way. An agent deprived of institutional context fails as a newly arrived engineer fails. The authors of *Knowledge Activation* conclude

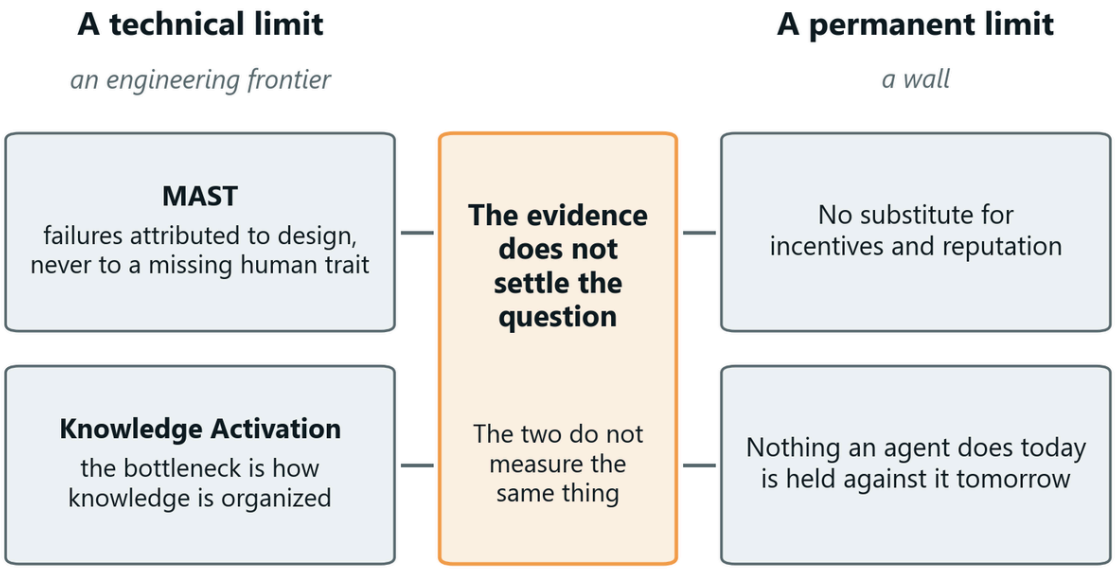


FIGURE 25.1 The two readings of the boundary weighed in this chapter, drawn by this report. A study of failures and the work on tacit knowledge support a technical limit (MAST, arXiv 2503.13657, arXiv 2603.14805). The two properties left without a substitute support a permanent one.

that the bottleneck is how knowledge is organized, not how capable the model is (arXiv 2603.14805). A problem of organization is a problem that better organization can solve.

On this reading, the boundary is an engineering frontier. Each improvement in models, in context handling and in harness design moves it, and nothing says where it stops.

The case for a permanent one

The other side rests on the two properties the previous chapter left without a substitute. An instruction repeated at every turn can tell an agent what the goal is. It does not make the outcome matter to the agent. Nothing an agent does today is held against it tomorrow, and nothing it achieves is credited to it.

Organizations are built around exactly those two facts about people. Promotion, dismissal, trust and the fear of losing it are how an organization obtains sustained effort without watching every action. A system that removes them has to watch every action, or contain it. On this reading the boundary is a wall, and what lies behind it is whatever needs someone with something to lose.

The verdict on the third hypothesis

The third hypothesis held that a boundary separates what an agent can carry from what demands a human. That much is confirmed, lever by lever, in the chapters of Part II. What cannot be decided is the nature of the boundary. The failure studies say it is a matter of design. The absence of any substitute for incentives and reputation says it may be more than that. Neither body of evidence refutes the other, because they do not measure the same thing.

Stating that plainly is more useful than choosing a side. A report that declared the limit technical would be ignoring two gaps nobody has closed. One that declared it permanent would be ignoring how many failures have turned out to be fixable.

One thing can be said about the part of the boundary that does move. Wherever a substitute exists, moving the boundary has a price. Accountability is bought with logs and gates,



FIGURE 25.2 A wall that is being moved forward in one place, and whose end is lost in the mist, an editorial allegory by this report for the open question of this chapter. The gaps it stands for are those identified in arXiv 2512.07665.

and tacit context with documentation. This report calls that price the **cost of explicitation**: what must be spent, in tokens, in engineering and in human attention, to state in software what a team leaves unsaid.

For a designer, the undecided question changes less than it seems. Whether the wall is permanent or not, it stands today, and today's system has to be built around it. The next chapter draws that map: what can be handed to an agent, what must be supervised, and what stays with people.

CHAPTER 26

People set the frame, and agents do the work inside it

Part IV has shown what an agent borrows from an organization, what it brings that is foreign to one, and what it lacks. This chapter draws the practical conclusion. Given all that, where does an agent fit in an organization led by people? The answer gathers the eight delegation boundaries of Part II into one table, and one pattern runs through all of them.

Each lever chapter ended on a boundary, stated in two short sentences: what can be handed to an agent, and what cannot. Table 26.1 sets the eight side by side, in the words of the chapters that drew them.

TABLE 26.1 **The delegation boundary on each of the eight levers**

Lever	Delegable	Not delegable
Context	Fetching information, filtering it, summarizing it, compacting it	Deciding what deserves to be written down once and for all
Action capability	The choice of action inside the perimeter	The definition of the perimeter
Control loop	The choice of the next step inside a bounded space	Deciding how wide that space is
Decomposition and specialization	Cutting a task into subtasks, once the criterion is supplied	The choice of that criterion and the definition of the roles
Verification and conformance	The judgment passed on one step, once the reference is supplied	Writing that reference
State persistence	Writing, retrieval and resume, once the shape of what persists is fixed	Fixing that shape
Capability evolution	Extracting candidate skills, generating tool code, indexing procedures	The rules for admitting a capability, and approval of permanent changes
Human control	Filtering routine commands, running policy checks, preparing diffs	Approving high-impact irreversible actions, and final accountability

Read down the right-hand column, the eight entries say the same thing in eight vocabularies. What stays with a human is the act of setting a frame: a perimeter, a width, a criterion, a reference, a schema, a rule. What can be delegated is the work done inside the frame. An agent can choose an action, and someone else decides which actions exist. An agent can judge a step, and someone else writes what the step should have been.

Part IV explains why the line falls there. Setting a frame is a decision whose consequences outlast the task, and it calls for the two things chapter 24 found no substitute for. Whoever defines a perimeter has to answer for what happens inside it, and to care what happens next month. An agent can do neither.

The rows of the table can be grouped by how much supervision each kind of work needs, which gives three tiers.

The first tier is work that runs unattended. It is mechanical and reversible: running tests, checking a format, saving a checkpoint, retrieving a document. A sandbox bounds what it can touch, and a mistake can be rolled back. Nobody needs to approve it, and chapter 13 showed that asking someone to would do harm.

The second tier is work that runs under supervision. Here a model makes real decisions: which step comes next, how to split a task, which skill is worth keeping. The decisions are delegated, and the structure around them is not. Several agents may explore in parallel while a single one writes, following the single-writer principle of chapter 19. A new skill is tested before it enters the registry. The human is called at a few prepared points and left alone otherwise.

The third tier is not delegated. It holds the frames themselves, together with the two responsibilities that follow from them: deciding how much risk is acceptable, and answering for the result. The law has begun to fix this tier in place. For high-risk systems, the AI Act requires that the people overseeing a system stay aware of automation bias. It also requires that they be able to override the system and to stop it (AI Act, art. 14).

The fitting place of AI in an organization led by people is therefore large and bounded. An agent can carry most of the execution and a good part of the decisions. It works inside frames that people set, and people keep the frames.

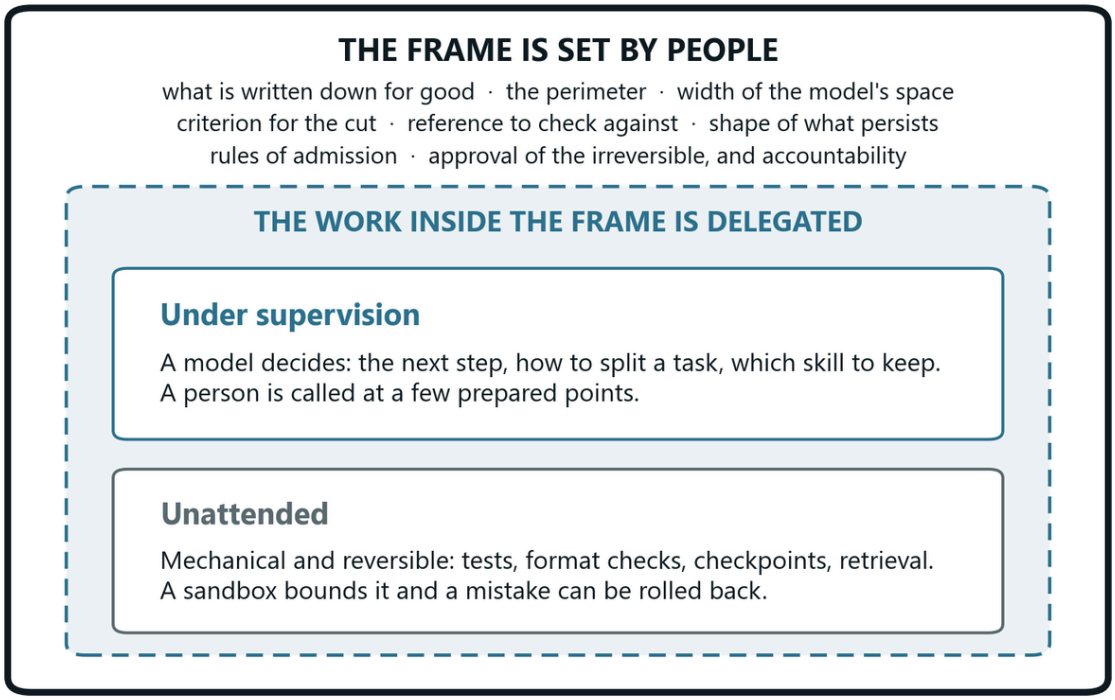


FIGURE 26.1 The pattern of Table 26.1, drawn by this report. People set the frame, and the work inside it is delegated, part of it under supervision and part of it unattended. The gaps that keep the frame with people are those identified in arXiv 2512.07665.

Chapter 25 added that the boundary can sometimes be moved, at a price it called the cost of explicitation. Every row of Table 26.1 has such a price. A perimeter has to be built, a reference written, a schema designed and maintained. How far a given organization should push each boundary depends on what it is willing to pay, and Part V starts from that question.

PART V

Cost decides, and the situation sets the dials

27 Cost is what decides between two configurations 101
28 Choosing takes three things: a structure, a situation and a measure 104
29 For each lever: a default, a dial, its price, and the tools of 2026 107
30 A regulated enterprise and a startup turn most dials in opposite directions 114
31 A configuration is a hypothesis until it is measured on five dimensions 118
32 A place to start: a bug-triage pipeline for a mid-sized team 122
33 The result: a situational matrix, a frontier that moves, and five gaps 127

CHAPTER 27

Cost is what decides between two configurations

Four parts of this report have arrived, by different roads, at the same question.

Part II found one loop under every methodology, then eight levers that describe how two systems differ. Chapter 15 showed the limit of that description: the levers say where a tree-search agent and a `while` loop part ways, and they cannot say which one to build. Part III read the catalogue of patterns, frameworks and protocols as settings of those levers, and found that no vendor publishes a measurement comparing one setting with another. Part IV traced these structures to their origin. They come from us. Roles, procedures, reviews and hierarchies were borrowed from human organizations because those were the only working models of coordinated, fallible labor anyone had. Where the borrowing breaks, a substitute can usually be engineered, and it has a price.

Each road ends on the same words: at what cost? This chapter takes that question as the answer. Architecture describes the difference between two methodologies. What decides between them is what each one costs, in tokens, in time, and in human attention.

Tokens come first because they are the best measured. Anthropic analysed its own multi-agent research system on BrowseComp, an evaluation of hard-to-find information on the web. Three factors explained 95 % of the variance in performance. Token usage by itself explained **80 %**, and the number of tool calls and the choice of model accounted for the rest of that share (Anthropic). The same post gives the multipliers. An agent uses about four times more tokens than a chat exchange, and a multi-agent system about **fifteen times more**. Read together, the two figures say something uncomfortable about architecture. A design that performs better while spending fifteen times more has not yet shown that its structure is what helped.

Latency is the second cost, and the least documented. Cutting a task into sequential steps, as chapter 9 reported, trades latency for accuracy, and that trade is the stated purpose of the pattern. Every verification pass and every hand-off between agents adds its own delay. No source in this corpus measures that delay across architectures, so it enters the argument as a known direction without a known size.

Human attention is the third, and the one most often left off the bill. It is spent twice. It is spent first on supervision: operators approved roughly **93 %** of permission prompts in Anthropic's telemetry, the sign of attention that has run out (Anthropic). It is spent again, earlier and less visibly, on making tacit knowledge explicit. What a team knows without writing it down has to be written before an agent can use it, and chapter 6 showed that this work falls on the most experienced people.

Safety has a price in the same currency. CaMeL, the defense that makes injected instructions structurally ineffective, completes 77 % of tasks where an undefended system completes 84 % (arXiv 2503.18813). Seven points of utility are the cost of a guarantee. Detection-based defenses appear to cost nothing, and twelve of them fell at rates above 90 % once the attacker adapted (arXiv 2510.09023). The cheap option was not a defense.

Why does cost decide where architecture does not? Because, in the present state of the evidence, architecture cannot be judged apart from spending. No study in this corpus compares two methodologies with the same model and the same token budget. Until one does, any claim that a structure performs better remains entangled with the fact that it consumed more.

This conclusion is easy to misread, in two ways. It does not say that the cheapest design is the right one. A cost is paid for something: wider reach, a lower risk of an irreversible error, a task that could not be done otherwise. And it does not say that structure is irrelevant. The levers remain the only vocabulary in which a design can be described, compared and changed. What the conclusion says is narrower. Structure tells a designer what the options are, and cost tells them what each option takes.

Whether a given cost is worth paying is not a property of the system. It depends on who pays and on what a failure would cost them. A bank and a three-person startup can look at the same fifteen-fold multiplier and both be right to answer differently. The practical rule that follows is the one Anthropic gives its own customers: find the simplest solution pos-

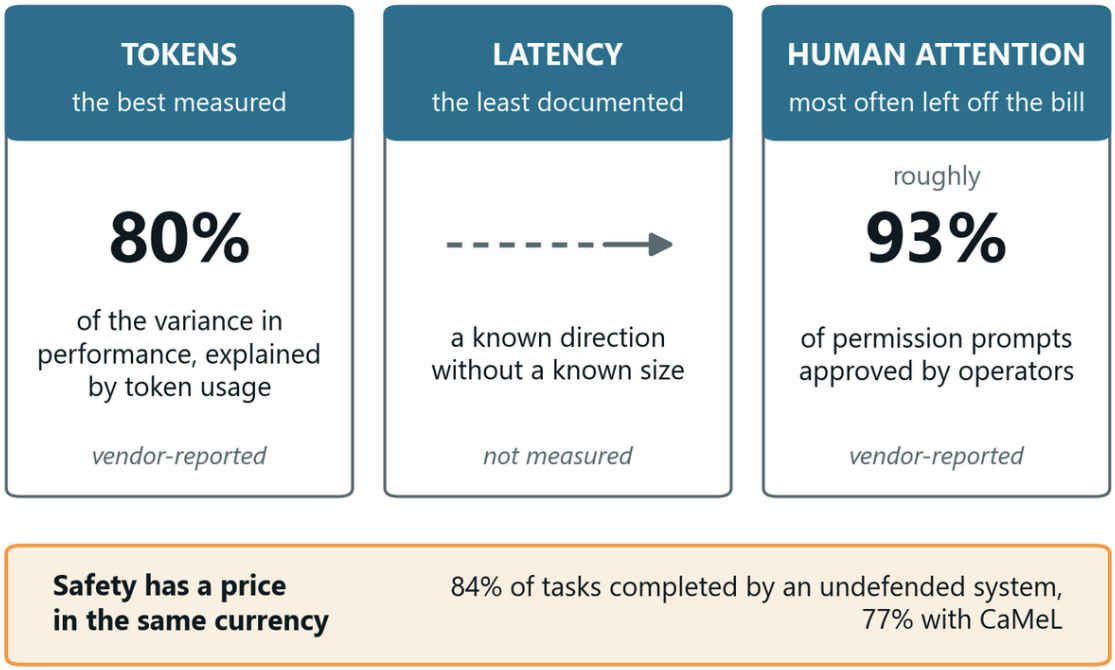


FIGURE 27.1 The three costs of an agentic system, tokens, latency and human attention, with what this chapter's sources measure of each, drawn by this report. The two percentages are vendor-reported (Anthropic, Anthropic), and the price of safety is the one measured for CaMeL (arXiv 2503.18813).

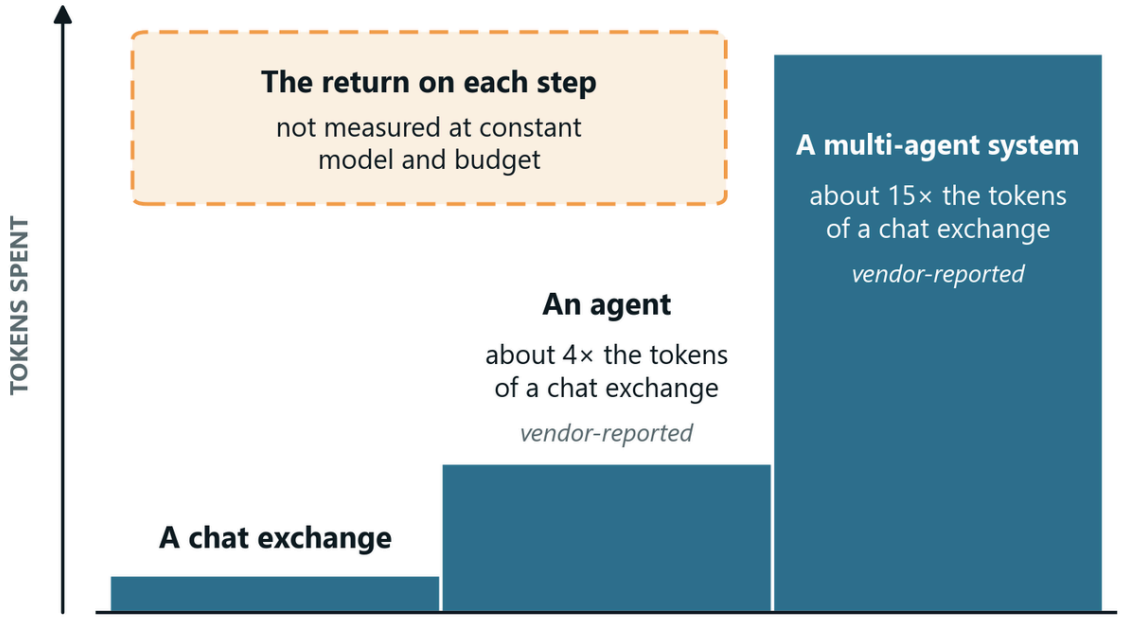


FIGURE 27.2 Tokens spent by a chat exchange, an agent and a multi-agent system, drawn by this report from the multipliers one vendor publishes (Anthropic). The return on each step has not been measured at constant model and budget.

sible, and add complexity only when it demonstrably improves the outcome (Anthropic). Every lever turned up has to earn back what it spends.

The reserve on this chapter is the same as on its sources. The 80 %, the multipliers and the 93 % all come from one vendor's telemetry on its own products. They give orders of magnitude and a direction. They are not independent measurements, and the exchange rate between complexity and performance stays approximate until someone measures it at constant model and budget.

Cost, then, is where the analysis ends and where the design work begins. If the right setting depends on the situation of whoever pays, the next step is to describe situations with the same care this report gave to levers. The following chapters do that.

CHAPTER 28

Choosing takes three things: a structure, a situation and a measure

Parts II through IV of this report built the analytical machinery. Eight levers decompose any agentic methodology into adjustable design dimensions. The organizational transposition showed where human structures map onto agent architectures and where they do not. The economic analysis established that token consumption explains 80% of performance variance and that cost, not architectural elegance, is the primary separator (Anthropic). All of this is necessary. None of it is sufficient. A framework that describes everything and prescribes nothing leaves the architect exactly where the field left them: facing a proliferation of tools with no principled way to choose.

This chapter closes that gap. It introduces the triangle that governs the rest of Part V: **Structure**, the eight levers as adjustable dials; **Situation**, the organizational context that determines where each dial should point; and **Measure**, the multidimensional instrument that tells the architect whether the configuration works. By the end of these final chapters, the reader should be able to place the right tools in front of their specific situation and verify the result.

The eight levers are not rules. They are questions. Context asks what the agent knows when it decides. Action capability asks what it can do irreversibly. The control loop asks whether the model or the code drives the next step. Each question admits a spectrum of answers, from fully deterministic to fully autonomous, and that spectrum is what makes the framework useful. A single organization does not answer all eight questions the same way. An enterprise deploying a compliance workflow will lock the Control loop lever to deterministic code while opening the Verification lever to its widest setting. A startup shipping a research assistant will do the opposite: let the model drive the loop and rely on lightweight post-hoc review. The levers do not prescribe the answer. The situation does.

What varies from one organization to another is not the technology. The same LLM, the same orchestration framework, the same sandbox can serve radically different architectures. What varies is the organizational context, and it varies along four dimensions that sit above the economics.

The first is **risk tolerance**: what does failure cost? In a regulated industry, a single hallucinated compliance report can trigger a regulatory investigation. The cost of failure is measured in legal liability, not in tokens. In a startup prototyping a feature, the cost of failure is a wasted afternoon. Risk tolerance determines how far the levers of Action capability, Verification, and Human control must be tightened. Singapore's Model AI Governance Framework for Agentic AI, which its publisher presents as the first governance framework designed for agents, builds on the same principle. It asks organizations to assess each use

by the likelihood and the severity of what could go wrong, and to make human review proportionate to that risk (IMDA).

The second is **process predictability**: can the workflow be defined in advance? A loan approval pipeline follows a fixed sequence of steps. A research task does not. Predictability determines the setting of the Control loop lever and the Decomposition lever. Where the process is predictable, deterministic orchestration eliminates an entire class of failure modes. Where it is not, the model must drive the loop, and the coordination cost must be paid.

The third is **human attentional budget**: how much oversight capacity does the organization have? Anthropic's telemetry established that users approve 93% of permission prompts once the volume overwhelms their attention (Anthropic). A large enterprise with a dedicated oversight team can sustain more human gates than a solo founder who is also the product manager, the engineer, and the customer. The attentional budget determines how the Human control lever is set: frequent high-friction gates for the enterprise, rare circuit-breaker gates for the startup.

The fourth is **velocity requirement**: how fast must the system ship and iterate? A team that needs to deploy in days cannot afford weeks of governance review. Velocity determines the balance between the Capability evolution lever, the State persistence lever, and the overall weight of the scaffolding. Light scaffolding ships fast and breaks in ways that

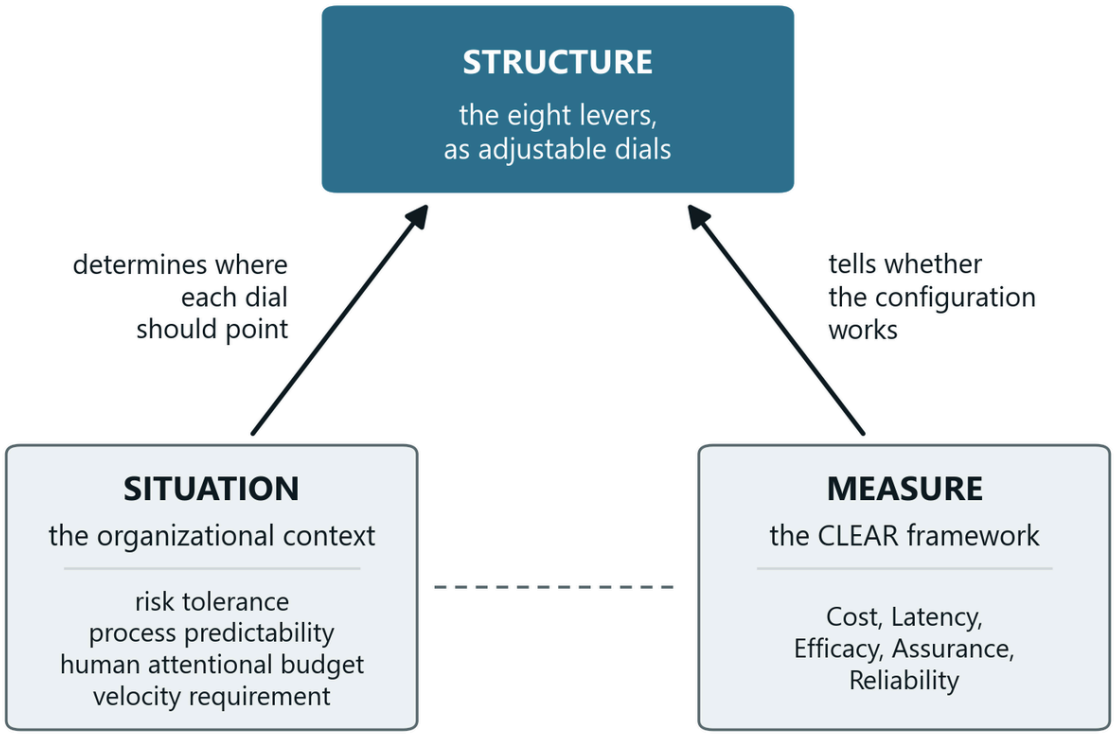


FIGURE 28.1 The triangle of structure, situation and measure, drawn by this report. The situation determines where each dial should point, and the measure, the CLEAR framework (arXiv 2511.14136), tells whether the configuration works.

heavy scaffolding would have caught. The tradeoff is real and must be acknowledged, not hidden.

These four criteria do not replace the economic boundary established in Chapter 27. They sit on top of it. Two organizations with the same token budget will produce different architectures because their risk tolerance, process predictability, attentional budget, and velocity requirements differ. The economics set the floor. The situation shapes the room.

The third side of the triangle is measurement. Knowing which levers to set and why is not enough if the architect cannot verify that the configuration works in production. The field's dominant evaluation metric, task completion accuracy measured on a single run, is a poor predictor of readiness for production. A study of twelve benchmarks and six agents asked experts to rate how ready each agent was for deployment. Accuracy alone correlated with those ratings at only 0.41. The multidimensional CLEAR framework (Cost, Latency, Efficacy, Assurance, Reliability) correlated at 0.83 (arXiv 2511.14136). The gap is not marginal. An architect who optimizes for accuracy alone will deploy agents that cost 4.4 to 10.8 times more than cost-aware alternatives with comparable performance.

CLEAR provides five dimensions that map directly onto the concerns raised throughout this report. Cost captures the token and compute expenditure that Chapter 27 established as the primary discriminator. Latency captures the time-to-completion that matters for real-time and user-facing workflows. Efficacy captures the functional success rate. Assurance captures policy compliance, guardrail adherence, and security posture. Reliability captures consistency across repeated runs, a dimension the field has systematically neglected. The CLEAR paper cites an earlier benchmark on consistency. Agents that succeed 60% of the time on a single run succeed only 25% of the time when they must succeed eight times in a row (arXiv 2511.14136). A single lucky pass is not a deployment-ready system.

The chapters that follow will fill each side of the triangle. Chapter 29 opens the mixing board: for each lever, the default setting, the situational dial, the cost of turning it, and the concrete tools from a catalogue of 242 entries across 31 technology layers that implement each position. Chapter 30 assembles two contrasting profiles, an industrial enterprise and an agile startup, and maps them across all eight levers with specific tool choices and measurement targets. Chapter 31 operationalizes the CLEAR dimensions into a practical measurement protocol. Chapter 32 presents a reference instantiation for a mid-complexity team. Chapter 33 gathers the result into one matrix, shows how fast its settings are moving, and states what the methodology cannot yet do.

CHAPTER 29

For each lever: a default, a dial, its price, and the tools of 2026

The eight levers are the dials of the situational architecture, and this chapter is the mixing board. For each lever it gives a default setting, the position that costs least and risks least. It then names the situational criterion that justifies turning the dial up, and what turning it costs. Finally it ties each position to tools that exist in 2026.

That last step needs a map of the tools. The one used here is the *Agentic AI Stack 2026* catalogue, a working index compiled by the author for this report in July 2026 from web research. It lists 242 tools sorted into 31 layers, from IDE extensions to registries, and marks each tool as a standard choice, an alternative, or one to approach with caution. The catalogue is a practitioner's inventory, not a benchmark. It records what exists and what is commonly chosen, and it measures nothing. The layer numbers below are the catalogue's own.

An architecture is not a collection of tools. It is a set of answers to the eight questions, and the tools come afterwards, as the means of holding each answer in place.

Lever 1: Context (What the agent knows)

The default setting: Just-in-time retrieval. The agent is given only the context needed for the step in front of it.

The situational dial: Process predictability. When a task is open-ended, the agent needs a wider window to reason about dependencies nobody listed in advance.

The cost: Tokens, and the dilution described in chapter 6. Feeding a whole codebase into the prompt degrades reasoning on the instruction that matters.

The tool mapping (Layers 12, 18 to 21, 29): The default is built from document ingestion (Layer 18), embeddings (Layer 19) and a vector store (Layer 20), with native prompt caching (Layer 12) to avoid paying twice for the same prefix. Turning the dial up adds reranking (Layer 21) and code intelligence (Layer 29), which widen what the agent can reach without widening what it must read.

TABLE 29.1 Representative tools for the Context lever

Tool	Layer	What it does	Why it sits on this lever
Docling	18. Document ingestion	Converts PDF, Word, slides and HTML into Markdown or JSON, with OCR	Decides the form in which documents enter the window
BGE-M3	19. Embeddings	One model producing dense, sparse and multi-vector representations	Determines what retrieval is able to find
pgvector	20. Vector store	Vector search inside Postgres	Holds what can be fetched just in time
Cohere Rerank 4	21. Reranking	Managed multilingual reranking of retrieved passages	Keeps only the highest-signal passages in the window
Native prompt caching	12. Caching and cost	Reuses a repeated system prefix at a reduced rate	Lowers the price of context that does not change

Lever 2: Action capability (What it can do irreversibly)

The default setting: Read-only, with a human executing. The agent proposes and a person applies.

The situational dial: Velocity requirement and risk tolerance. High velocity calls for write access. Low risk tolerance calls for strict containment of that access.

The cost: Exposure. An agent that can write can damage, and chapter 7 showed that a demonstrable guarantee costs utility, seven points in the one case measured.

The tool mapping (Layers 7, 8, 9): The default needs no tools. Turning the dial up requires an execution sandbox (Layer 9) before anything else. Identity and authorization (Layer 7) then settle on whose behalf the agent acts, and browser and web tooling (Layer 8) extends its reach outside the workspace. Granting write access without a sandbox layer cancels every other safeguard.

TABLE 29.2 Representative tools for the Action capability lever

Tool	Layer	What it does	Why it sits on this lever
E2B	9. Execution sandbox	Isolated Linux sandboxes for generated code	Sets the perimeter outside which the agent can reach nothing
Firecracker	9. Execution sandbox	The microVM layer underneath several sandbox services	The same perimeter, for teams that want to control it directly
OAuth 2.1 with Dynamic Client Registration	7. Identity and authorization	Issues tokens to agents at run time	Bounds what an agent is permitted to call
Arcade	7. Identity and authorization	Per-user OAuth with just-in-time consent and a token vault	Ties each action to the user it is taken for
Browser Use	8. Browser and web data	A Python browser agent that reads the page structure	Extends action to the open web, where containment is hardest

Lever 3: Control loop (Who drives the next step)

The default setting: Deterministic orchestration. A state machine or a script fixes the sequence, and the model is called for specific tasks inside it.

The situational dial: Process predictability. If the workflow cannot be written in advance, as in open-ended research, the loop has to be handed to the model.

The cost: Unbounded execution. A model-driven loop can retry without end, and its path cannot be predicted.

The tool mapping (Layers 2, 3, 4, 27): Code-driven loops run on classic workflow orchestrators (Layer 27) or on durable execution engines (Layer 4). Model-driven loops run on orchestration frameworks (Layer 3), or are handed whole to an autonomous code agent (Layer 2).

TABLE 29.3 **Representative tools for the Control loop lever**

Tool	Layer	What it does	Why it sits on this lever
Prefect	27. Classic workflow orchestrator	Scheduling, retries and observability for scripted workflows	Code decides the sequence, the default position
LangGraph	3. Orchestration framework	Stateful cyclic workflows with checkpointing	Lets a designer say, node by node, whether code or the model decides
Pydantic AI	3. Orchestration framework	Type-safe agents with typed contracts end to end	Bounds a model-driven step inside a code-driven flow
Claude Agent SDK	3. Orchestration framework	Agents with built-in file and shell tools	Hands the loop to the model inside a prepared harness
Claude Code	2. Autonomous code agent	Autonomous multi-file edits	The fully model-driven end of the dial

Lever 4: Decomposition (How the work is divided)

The default setting: The single-writer principle. One agent holds the context and writes the output. If it needs help it calls tools, not peers.

The situational dial: Task complexity and attentional budget. When a task needs contexts that work against each other, such as writing code and reviewing it as an adversary, splitting it across specialized agents becomes necessary.

The cost: Coordination. Multi-agent systems consume about 15× more tokens than chat exchanges (Anthropic).

The tool mapping (Layers 2, 3, 6): The default is a single autonomous code agent (Layer 2). Turning the dial up means using the multi-agent features of an orchestration framework (Layer 3), or routing between agents over a protocol (Layer 6).

TABLE 29.4 **Representative tools for the Decomposition lever**

Tool	Layer	What it does	Why it sits on this lever
CrewAI	3. Orchestration framework	Teams of agents defined by roles and tasks	Cuts the work by role
Microsoft Agent Framework	3. Orchestration framework	The merged successor of AutoGen and Semantic Kernel	Offers orchestrated multi-agent patterns with checkpointing
OpenAI Agents SDK	3. Orchestration framework	Lightweight multi-agent runtime with tracing and guardrails	Makes the choice between specialists as tools and handoffs explicit
Google ADK	3. Orchestration framework	Agent development kit with workflow and model-driven agents	Lets the cut be written in advance or left to an orchestrator
A2A	6. Protocol and standard	Communication between agents across organizational boundaries	Carries a division of labor that crosses systems

Lever 5: Verification (How output is checked)

The default setting: Deterministic checks. Code is checked by compilers and tests, and data by schemas.

The situational dial: Risk tolerance. When a subtle error is expensive and no compiler can catch it, as in a strategic analysis, a model has to be added as a judge.

The cost: Tokens, latency and false alarms. A model that judges has its own error rate and triggers retries that were not needed.

The tool mapping (Layers 13, 14, 15, 22): The default relies on structured output (Layer 22) and ordinary continuous integration. Turning the dial up adds evaluation suites (Layer 14), guardrails that intercept violations in flight (Layer 15), and observability (Layer 13) to record the calls that chapter 10 said to audit.

TABLE 29.5 **Representative tools for the Verification lever**

Tool	Layer	What it does	Why it sits on this lever
Instructor	22. Structured output	Validated structured data from model outputs, with automatic retries	The deterministic check on the shape of an output
DeepEval	14. Evals	Test-style evaluations for models and agents	Checks output quality against a set of cases
promptfoo	14. Evals	Declarative evaluation scenarios in configuration files	Makes a regression in behavior visible before release
NeMo Guardrails	15. Guardrails	Input and output rails defined in a dedicated language	Intercepts a policy violation at the point of execution
Langfuse	13. Observability	Hierarchical traces of calls, with evaluations and cost	Records the actions on which an audit has to rest

Lever 6: State persistence (What survives the session)

The default setting: Session-scoped memory. State ends when the run ends.

The situational dial: Process predictability and velocity. If a task spans days, or has to resume after a human intervenes, state must be made durable.

The cost: Complexity and data governance. Persisted agent memory is a second database, to be secured and purged like any other.

The tool mapping (Layers 4, 10): The default needs no tools. Durable persistence calls for a memory layer (Layer 10) and for durable execution (Layer 4), which replays a workflow from the step that failed without repeating the model calls that succeeded.

TABLE 29.6 Representative tools for the State persistence lever

Tool	Layer	What it does	Why it sits on this lever
LangGraph checkpointers	10. Memory	Saves graph state to Postgres, Redis or DynamoDB	Gives a thread back exactly where it stopped
mem0	10. Memory	Persistent memory scoped to a user, a session or an agent	Carries facts across sessions
Zep / Graphiti	10. Memory	A knowledge graph of facts with periods of validity	Keeps long-term knowledge, with compliance certifications
DBOS	4. Durable execution	Durability through Postgres, with replay from the last step	Decides what a crash loses
Cloudflare Workflows	4. Durable execution	Durable multi-step execution on a serverless platform	The same guarantee without new infrastructure

Lever 7: Capability evolution (How the agent improves)

The default setting: Static prompts and curated skills. What the agent can do is written by people and changed by people.

The situational dial: Attentional budget. If the team cannot update prompts by hand for every edge case, part of that work has to be automated.

The cost: Predictability. A system that rewrites its own prompts or weights drifts, and the drift has to be evaluated.

The tool mapping (Layers 5, 24, 30): The default uses agent skills and plugins (Layer 30). Evolution comes first through optimization (Layer 5), which rewrites prompts from execution traces, and only then through fine-tuning (Layer 24), which is heavier and harder to reverse.

TABLE 29.7 Representative tools for the Capability evolution lever

Tool	Layer	What it does	Why it sits on this lever
AGENTS.md	30. Agent skill and plugin	A project context file read by coding agents	The static, human-written form of capability
skill-creator	30. Agent skill and plugin	Creates, edits and evaluates agent skills	Adds a capability under human review
DSPy	5. Optimization	Declarative pipelines whose prompts are optimized by program	Changes the decision policy without touching weights
GEPA	5. Optimization	Optimizes prompts by reflecting on execution traces	Turns past runs into a revised instruction
Unsloth	24. Fine-tuning	Faster, lighter fine-tuning with adapters	Changes the weights, the most durable and least reversible site of evolution

Lever 8: Human control (When the human intervenes)

The default setting: A gate before each consequential action. The agent asks before it does anything that matters, and such gates are kept few, as chapter 13 requires.

The situational dial: Attentional budget and velocity. As volume grows, people approve without reading, at the 93 % rate already cited. Control then has to move from gates before the action to containment, review after the fact and circuit breakers.

The cost: Damage that has already happened. Review after the fact finds an error once it is in production.

The tool mapping (Layers 1, 26, 28): Step-by-step approval is built into IDE extensions (Layer 1). Review after the fact needs an agent interface (Layer 26), or a low-code and visual builder (Layer 28) in which a person can follow executions and adjust rules without stopping them.

TABLE 29.8 **Representative tools for the Human control lever**

Tool	Layer	What it does	Why it sits on this lever
Cline	1. IDE extension	An open-source coding agent with plan and act modes and approval per step	The gate-before-action setting, at its most frequent
CopilotKit	26. Agent interface	Interface components with human-in-the-loop support and an inspector	Builds the surface on which a person sees and intervenes
assistant-ui	26. Agent interface	Composable interface primitives for agent applications	The same surface, assembled piece by piece
Gradio	26. Agent interface	Quick web interfaces	A review screen that can be stood up in an afternoon
n8n	28. Low-code and visual	Visual workflows with a clear execution log	Lets a non-developer follow and correct what agents did

Where the 31 layers of the catalogue fall on the eight levers

Table 29.9 gathers the mapping in one place. An architect sets each lever from the situation, then picks from the layers on that row. The layer numbers are those of the catalogue, whose full index is in the spreadsheet that accompanies this report.

TABLE 29.9 **The catalogue's layers set against the eight levers**

Lever	Layers for the default setting	Layers added when the dial is turned up
1. Context	Ingestion (18), Embeddings (19), Vector store (20), Caching (12)	Reranking (21), Code intelligence (29)
2. Action capability	None	Sandbox (9), Identity and authorization (7), Browser and web data (8)
3. Control loop	Classic workflow orchestrator (27), Durable execution (4)	Orchestration framework (3), Autonomous code agent (2)
4. Decomposition	Autonomous code agent (2)	Orchestration framework (3), Protocol and standard (6)
5. Verification	Structured output (22)	Evals (14), Guardrails (15), Observability (13), Security and red teaming (16)
6. State persistence	None	Memory (10), Durable execution (4)
7. Capability evolution	Agent skill and plugin (30)	Optimization (5), Fine-tuning (24)
8. Human control	IDE extension (1)	Agent interface (26), Low-code and visual (28)

Five layers appear on no row: model gateways (11), mechanistic interpretability (17), local serving and inference (23), voice and real-time agents (25), and directories and registries (31). They are infrastructure or research tooling that any setting of the levers may rest on,

and assigning them to a lever would be forcing the fit. The matrix is a reading aid built by this report, and like the catalogue it draws on, it has not been tested against outcomes.

The next chapter turns all eight dials at once, for two organizations that could hardly differ more.

CHAPTER 30

A regulated enterprise and a startup turn most dials in opposite directions

The previous chapter gave each lever a default and a dial. This one turns all eight at once. What does a complete configuration look like for one organization, and how different is it for another? Two profiles answer that question, a regulated enterprise and a small startup. Each is then set beside a documented deployment, to check that the settings described here exist outside this report.

The profiles are constructions. They were built by applying the four situational criteria of chapter 28 to the decision rules of chapter 29, and their tool choices come from the catalogue. They show how the reasoning runs. They do not describe any particular company.

Profile A: the industrial enterprise

The first profile is a large organization in a regulated industry, such as finance or healthcare. Its situation reads as follows on the four criteria.

- **Risk tolerance:** Extremely low. An error carries legal liability and damages reputation.
- **Process predictability:** High. Workflows such as compliance checks or loan processing are rigidly defined.

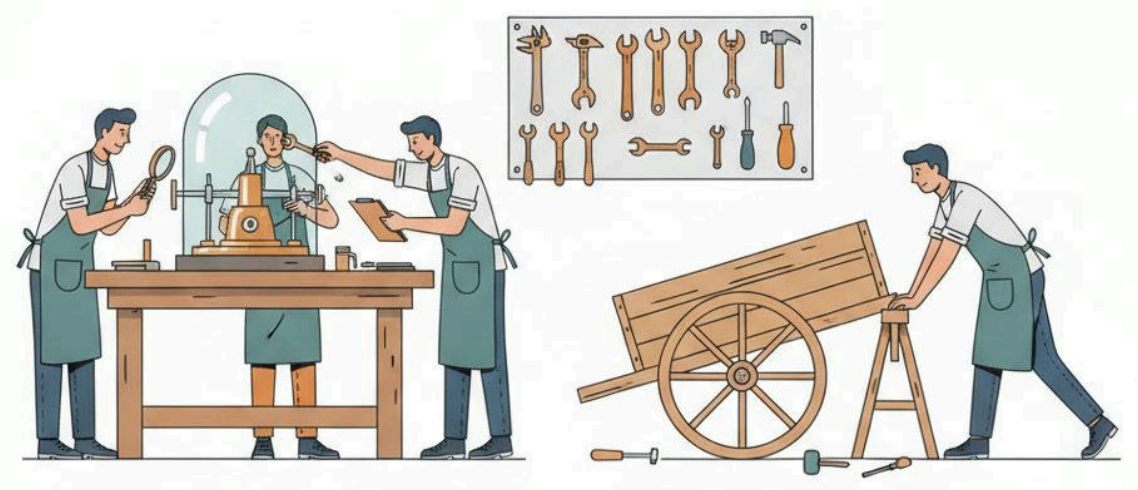


FIGURE 30.1 Two workshops take their tools from the same wall, an editorial allegory for this chapter: a careful team around a precise machine, and a lone builder already on the move. The two profiles that follow are constructions of this report, read against the CLEAR dimensions (arXiv 2511.14136).

- **Attentional budget:** High. The organization has teams dedicated to governance and oversight.
- **Velocity requirement:** Low to medium. Safety and compliance matter more than shipping tomorrow.

A situation of that kind produces a defensive, deterministic architecture. In the terms of the next chapter, it favors assurance and reliability over cost and speed.

- **1. Context:** Highly curated. The system relies on enterprise-grade ingestion (Unstructured) and strictly isolated vector databases (Milvus or Weaviate), so that data boundaries are never crossed.
- **2. Action capability:** Strictly contained. All execution happens within an enterprise sandbox (E2B Enterprise), with dry runs and full filtering of outbound traffic.
- **3. Control loop:** Deterministic. Code orchestrates the workflow. A tool like LangGraph is used as a rigid state machine in which the model only carries out specific classification or extraction steps.
- **4. Decomposition:** The single-writer principle, with read-only subagents. An orchestration loop hands narrow tasks to specialized workers, and a single node assembles the final state.
- **5. Verification:** Maximum stringency. Evaluations (DeepEval, Braintrust) run continuously, and guardrails (NeMo Guardrails, Galileo) intercept policy violations before they reach the user.
- **6. State persistence:** Durable and audited. Durable execution logs every step and resumes exactly where a run failed, and a memory layer with compliance certifications (Zep / Graphiti) holds what must outlive the task.
- **7. Capability evolution:** Controlled. Capabilities change through procedures approved by people and through prompt optimization (DSPy) gated by evaluation suites. Nothing improves itself unattended.
- **8. Human control:** Gates before consequential actions. The agent cannot commit a transaction or send an external communication without explicit approval, and those gates are placed at a few key points, as chapter 13 requires.

> **A documented case on the code-governed side: Stripe** Stripe, the payments company, has described the coding agents it runs internally, which it calls Minions (Stripe). Their loop is governed by what Stripe calls a blueprint, "a collection of agent skills interwoven with deterministic code": steps such as running the linters are fixed nodes, and the model works between them. Each agent runs in an isolated development machine, so that "any mistakes an agent might make are confined to the limited blast radius of one devbox". After the second push and run of automated checks, the branch goes "back to its human operator for manual scrutiny". Stripe reports more than 1,300 merged pull requests a week written this way, all of them reviewed by people.

The Stripe case matches Profile A on three levers: a control loop held by code, containment decided before the action, and a human gate at a fixed point. It departs from it on a fourth. Inside its sandbox, the Stripe agent runs with full permissions, which is the wide setting of the Action capability lever, made acceptable by the perimeter around it. The case is also an internal engineering tool, and says nothing about a regulated decision that affects a customer.

Profile B: the agile startup

The second profile is a small, fast-moving team building an internal tool or an early product.

- **Risk tolerance:** High. A wrong output costs little compared with moving slowly.
- **Process predictability:** Low. The tasks are open-ended, such as deep research or code generation.
- **Attentional budget:** Very low. A small team cannot watch hundreds of agent runs by hand.
- **Velocity requirement:** Extremely high. The system must be deployed and changed quickly.

The architecture that follows is fluid and autonomous. It favors cost and efficacy.

- **1. Context:** Broad and dynamic. The system uses lightweight ingestion (Docling) and a fast local vector store (pgvector).
- **2. Action capability:** Read and write. The agent has broad permissions inside a standard sandbox (E2B), where it can run code and test its own hypotheses.
- **3. Control loop:** Model-driven. The agent decides the sequence of steps, planning and iterating by itself through an SDK (Claude Agent SDK, Vercel AI SDK).
- **4. Decomposition:** A single agent. The team avoids the 15× token overhead of multi-agent systems and relies on one capable frontier model with a broad set of tools.
- **5. Verification:** Lightweight. The system relies on basic assertions and on the model correcting itself, with no heavy external guardrails.
- **6. State persistence:** Session-scoped. State lives in the context window or in a light memory tool (mem0), and is reset for each new task.
- **7. Capability evolution:** Rapid and empirical. The team edits system prompts directly and adds tools through MCP servers as needs appear, trusting fast feedback more than formal evaluation gates.
- **8. Human control:** Review after the fact. The agent works unattended within its domain. People read a daily summary, or respond when the agent escalates a failure it cannot resolve.

> **A documented case on the model-driven side: Shopify** Shopify has described how it built Sidekick, the assistant that helps merchants manage their stores (Shopify). Its loop is the open one: the model "decides on actions, those actions are executed in the environment, feedback is collected, and the cycle continues until the task is complete". Its advice on decomposition is blunt: "Avoid multi-agent architectures early: Simple single-agent systems can handle more complexity than you might expect". Context is supplied just in time, with instructions returned "alongside tool data exactly when they're needed". Verification rests on model judges, which the team calibrated from an agreement with human raters barely better than chance to one close to the agreement between humans.

Shopify is no startup, and the case is cited for its settings, not for the size of the company. It matches Profile B on the control loop, on decomposition and on context. It departs from it on verification, where Shopify invested heavily. That departure is instructive. A team that hands the loop to the model tends to pay it back in evaluation.

The two profiles could run on the same model and would still share almost nothing else. Their situations call for opposite positions on most of the eight dials, which is the claim chapter 28 set out to make concrete.

Neither documented case fits its profile exactly, and that is the second lesson of the chapter. Real systems mix settings: a code-governed loop with wide permissions, or a model-driven loop with heavy evaluation. No published account read for this report describes a deployment on all eight levers, so each case confirms a few rows and leaves the others blank.

A configuration is a hypothesis until someone measures whether it works. The next chapter supplies the instrument.

Regulated enterprise		Small startup
	THE SITUATION	
Extremely low	**Risk tolerance**	High
High	**Process predictability**	Low
High	**Attentional budget**	Very low
Low to medium	**Velocity requirement**	Extremely high
	THE SETTING OF EACH LEVER	
Highly curated	**1. Context**	Broad and dynamic
Strictly contained	**2. Action capability**	Read and write, in a sandbox
Deterministic: code decides	**3. Control loop**	Model-driven
Single writer, read-only subagents	**4. Decomposition**	A single agent
Maximum stringency	**5. Verification**	Lightweight
Durable and audited	**6. State persistence**	Session-scoped
Controlled, approved by people	**7. Capability evolution**	Rapid and empirical
Gates before consequential actions	**8. Human control**	Review after the fact

FIGURE 30.2 The two profiles of this chapter side by side, drawn by this report from the lists above: the situation on the four criteria of chapter 28, then the setting of each lever. Both are constructions, to be read against the CLEAR dimensions of the next chapter (arXiv 2511.14136).

CHAPTER 31

A configuration is a hypothesis until it is measured on five dimensions

The previous chapters established how to set the eight architectural levers based on situational constraints. But an architecture is only a hypothesis until it is measured. The field has historically relied on a single, flawed metric: task completion accuracy on a single benchmark run. This metric is a poor guide for production deployments.

A systematic analysis of twelve major benchmarks and six leading agents found that accuracy alone correlates with experts' ratings of deployment readiness at only $\rho = 0.41$ (arXiv 2511.14136). A metric that tracks expert judgement so loosely is a weak basis for an enterprise system. The same analysis reports that optimizing an agentic system for accuracy alone produces configurations that cost 4.4 to 10.8 times more than cost-aware alternatives while delivering comparable functional utility.

To replace this flawed standard, this report adopts the multidimensional CLEAR framework (Cost, Latency, Efficacy, Assurance, Reliability). When these five dimensions are

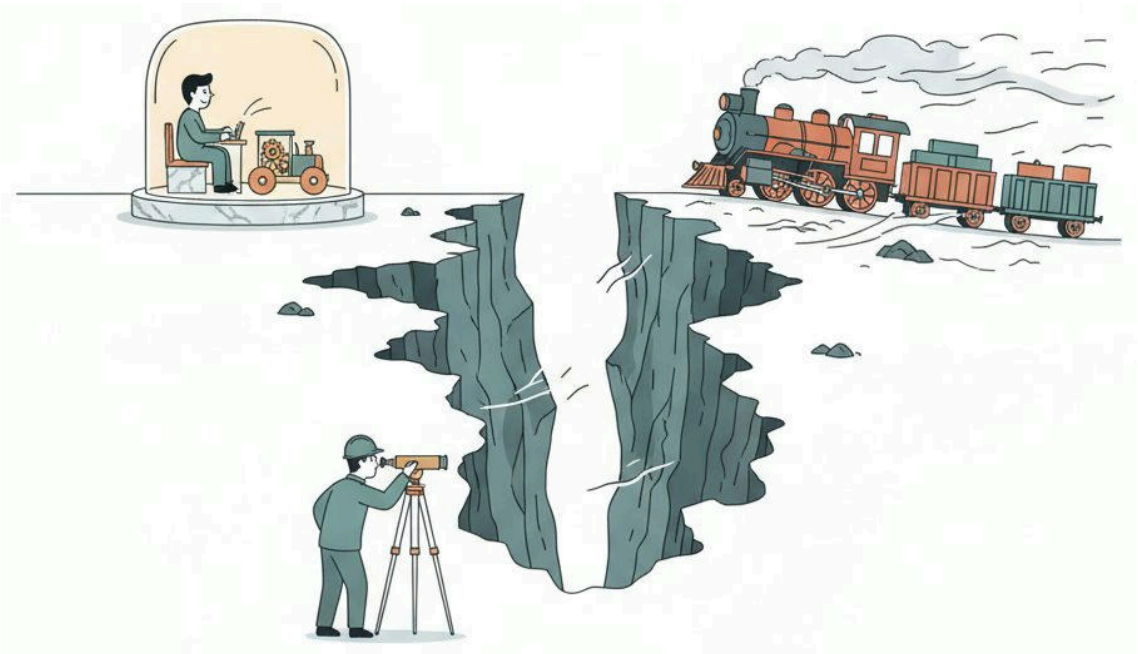


FIGURE 31.1 A model engine under glass and a working train, with a surveyor measuring the gap between them, an editorial allegory by this report. A score obtained once on a benchmark says little about behaviour in production, which is the argument of the CLEAR framework (arXiv 2511.14136).

measured together, the correlation with those expert ratings rises to ϱ = 0.83 (arXiv 2511.14136). These figures come from a single paper and have not been replicated. They are used here as the best available instrument, not as settled fact. The situational triangle introduced in Chapter 28 uses the CLEAR framework as its measurement vertex because it operationalizes the tradeoffs inherent in the eight levers.

To guide engineering decisions, each dimension must be measured in a production setting against explicit SLAs. The following criteria define how architects evaluate and optimize these five parameters.

Cost is no longer measured purely in token volume per request, but as cost per solved task: the total spent divided by the number of tasks actually solved. This metric aggregates the token cost of the entire reasoning loop, including retries, evaluator calls, and failed attempts. A system that achieves high accuracy through infinite LLM-driven retries (turning the Control loop lever fully autonomous) will fail the CPST threshold instantly.

- **Why it's important:** Prevents runaway expenses in autonomous loops where agents might get stuck in expensive retry cycles without actually solving the task.
- **Bad example:** A customer service agent that consumes $0.50 in tokens bouncing between sub-agents to answer a simple FAQ, wiping out the profit margin.
- **Good example:** Setting a hard token budget per query and routing complex queries to a human if the budget is reached, keeping the cost per solved task predictable.

Latency measures the wall-clock time from the initial trigger to the final committed action. In real-time or user-facing workflows, latency is a hard constraint. Adding LLM-as-a-judge verification (turning up the Verification lever) or delegating to multiple specialized peers

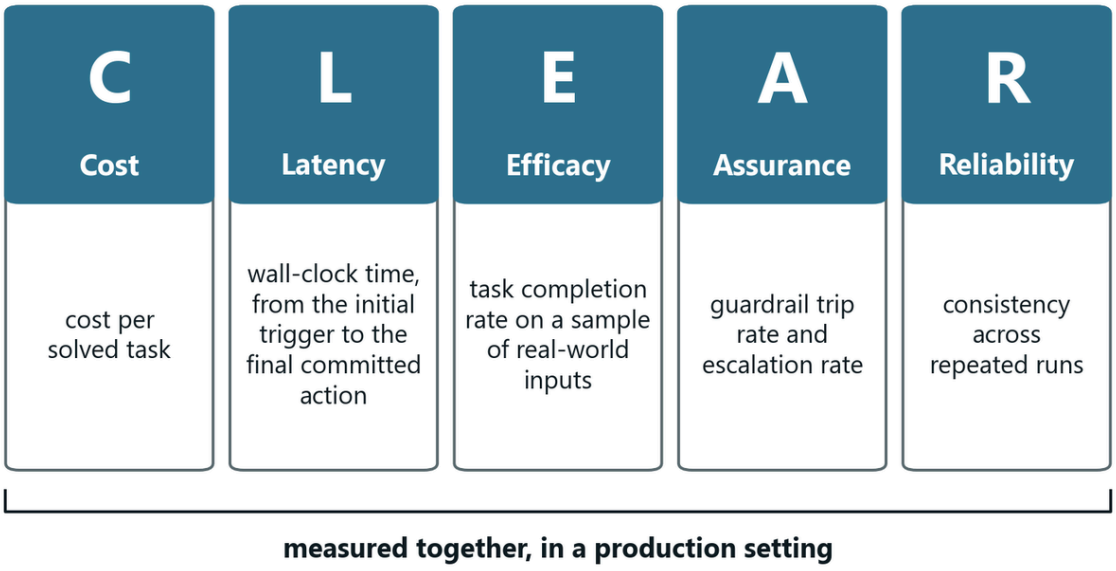


FIGURE 31.2 The five CLEAR dimensions (arXiv 2511.14136) and the measure this chapter gives for each in production, drawn by this report.

(turning up the Decomposition lever) significantly degrades latency. The architect must measure whether the latency budget permits these architectural choices.

- **Why it's important:** User experience degrades exponentially after a few seconds of waiting. Agents that take too long to act lose user trust or fail to meet SLAs.
- **Bad example:** A real-time voice assistant that pauses for 10 seconds to run a 3-step verification chain before answering a simple user query.
- **Good example:** A background data processing agent that operates asynchronously, trading high latency for high accuracy without impacting the user experience.

Efficacy replaces single-run accuracy. It measures the functional task completion rate over a statistically significant sample of real-world inputs, not static benchmarks. It is the baseline metric that proves the agent can do the job, but it is never evaluated in isolation.

- **Why it's important:** Traditional accuracy measures whether the LLM got the right answer in a vacuum, but efficacy measures whether the *entire system* (tools, memory, control loop) successfully accomplished the user's goal in the real world.
- **Bad example:** Measuring success by how well an agent answers multiple-choice questions on a generalized benchmark, while failing 40% of real API calls in production.
- **Good example:** Tracking the percentage of user requests that are fully resolved without requiring human escalation or resulting in an error state.

Assurance measures policy compliance and security posture. It is quantified by the guardrail trip rate (how often an active guardrail intercepts a forbidden action) and the escalation rate (how often the agent hands the task back to a human). High assurance is the primary goal of the Enterprise profile described in Chapter 30, achieved by tightening the Action capability and Human control levers.

- **Why it's important:** Without assurance, agents expose the company to data leaks, brand damage, and regulatory fines. It is the metric that guarantees safety.
- **Bad example:** A sales agent that hallucinates a 50% discount for a customer and commits the transaction without a human-in-the-loop check.
- **Good example:** An internal HR agent that refuses to summarize documents containing PII and automatically alerts the security team when prompted to do so.

Reliability is the most neglected dimension. It measures consistency across repeated runs. The CLEAR paper, citing an earlier benchmark, reports a wide reliability gap: an agent configuration that registers a 60% success rate on a single evaluation run drops to a 25% success rate when required to succeed consistently across eight consecutive runs on the exact same task. A single lucky pass is an illusion. Production systems must be evaluated on 8-run or 10-run consistency.

- **Why it's important:** LLMs are non-deterministic. A system that works today might fail tomorrow on the exact same input. Reliability ensures the architecture is stable enough for enterprise use.
- **Bad example:** An automated code reviewer that spots a vulnerability on Monday, but misses the exact same vulnerability on Tuesday because the model's temperature caused it to focus on typos instead.

- **Good example:** A data extraction pipeline that correctly pulls invoice data 10 out of 10 times because the prompt and control loop are tightly constrained and verified.

These five dimensions force the architect to acknowledge tradeoffs. If the Efficacy dimension falls below target, the instinct is to turn up the Decomposition lever (adding more agents) or the Control loop lever (giving the LLM more autonomy). The CLEAR framework forces the architect to see that doing so will immediately degrade the Cost and Latency dimensions. The measurement instrument ensures that the situation, not the technology, dictates the architecture.

The instrument is now in place beside the levers and the criteria. The next chapter uses all three on one concrete case.

CHAPTER 32

A place to start: a bug-triage pipeline for a mid-sized team

Chapter 30 demonstrated the extremes of the situational architecture: the highly defensive Enterprise and the highly autonomous Startup. Most organizations sit between these poles. They need a system that ships fast but does not break catastrophically. They need the "if you had to pick one configuration, here is where to start" architecture.

This chapter presents that reference instantiation. To ground the tool selection and settings, we isolate a specific deployment: an internal tool for **L1 technical support and triage of internal bug reports**, designed for a mid-complexity engineering team (10 to 20 engineers) undertaking their first production agentic deployment.

The governing principle of this reference architecture is the spectrum of autonomy: start deterministic, and add agentic nodes only where the task strictly demands it (Anthropic).

The situation: bug triage in a mid-sized team

When an internal user submits a bug report, it must be read, reproduced if possible, cross-referenced with the codebase, and routed to the correct developer with a preliminary diagnosis. According to the four situational criteria established in chapter 28, this process profiles as follows:

- **Risk tolerance: Moderate.** It is an internal operational tool, not a customer-facing application. A hallucination or a failed triage means an engineer has to manually read the ticket, creating friction but not a catastrophic data leak or reputational damage.
- **Process predictability: Semi-structured.** Bug reports follow a template (steps to reproduce, expected vs. actual behavior), but the natural language varies wildly. The steps to reproduce require cognitive flexibility and cannot be entirely hardcoded.
- **Human attentional budget: Low to medium.** The entire purpose of the deployment is to offload L1 triage from the engineering team so they can focus on writing features. However, engineers remain available to review the agent's work when they actually take over the routed ticket.
- **Velocity requirement: High.** Triage must happen quickly so that critical bugs are addressed without blocking the organization.

This profile dictates a configuration that prioritizes execution speed and cost-efficiency while maintaining structural safeguards.

The reference setting of each lever

For a team deploying its first agentic system against a semi-structured process, the eight levers are set to balance autonomy with legibility.

1. Context (Targeted Retrieval): Dumping the entire codebase into the prompt is too expensive and dilutes the model's attention. Instead, the context must be constructed dynamically based on the bug report's contents.

- **Setting:** Targeted RAG and strict ingestion formatting.
- **Tool:** Docling (Layer 18) for clean ingestion of incoming reports, backed by a local pgvector instance (Layer 20) with standard embeddings for code retrieval. Docling ensures the messy incoming tickets are normalized into Markdown before the LLM ever sees them, reducing token waste.

2. Action capability (Containerized Read-Write): The agent needs to attempt to reproduce the bug by running code or tests. Giving it production access or direct access to the main repository is an unacceptable risk.

- **Setting:** Ephemeral, isolated sandbox execution.
- **Tool:** E2B Sandbox (Layer 9). The agent can write reproduction scripts, run local tests, and read logs, but the environment is destroyed after the session. It cannot reach production databases. This is what makes a moderate risk tolerance acceptable.

LEVER	SETTING	TOOL
1. Context	Targeted RAG and strict ingestion formatting	**Docling and pgvector**
2. Action capability	Ephemeral, isolated sandbox execution	**E2B Sandbox**
3. Control loop	Hybrid orchestration: code drives the process, the model executes the nodes	**Temporal and Pydantic AI**
4. Decomposition and specialization	Single frontier model equipped with specific tools	**A tightly curated set of MCP tools**
5. Verification and conformance	High structural conformity, low cognitive verification	**Native Structured Output and Llama Guard**
6. State persistence	Orchestrator-managed checkpoints	**Temporal's event history**
7. Capability evolution	Static capabilities with manual updates	**Standard Git workflows**
8. Human control	Halt and escalate	**A programmatic threshold**

FIGURE 32.1 The reference setting and the tool of each of the eight levers for the bug-triage case, drawn by this report. Code drives the macro-process and the model acts inside bounded steps, a composition of loop primitives of the kind observed in arXiv 2604.03515.

3. Control loop (Augmented Workflow): A fully autonomous LLM loop (like ReAct) is too brittle for a production pipeline where the macro-steps are known (read → reproduce → search → route).

- **Setting:** Hybrid orchestration. Code drives the process; the model executes the nodes.
- **Tool:** Temporal (Layer 4) manages the macro-process deterministically. The LLM is only granted autonomy within specific, bounded steps using a lightweight framework like Pydantic AI (Layer 3).

4. Decomposition and specialization (Tool-augmented Single Agent): The architecture rejects the overhead of multi-agent orchestration. Passing the ticket between a "Reader Agent", a "Reproducer Agent", and a "Router Agent" compounds latency and token costs.

- **Setting:** Single frontier model equipped with specific tools.
- **Tool:** A single frontier model acts as the sole router and executor. It is equipped with a tightly curated set of MCP tools (Layer 6), such as a GitHub MCP for reading code and a Jira MCP for updating the ticket.

5. Verification and conformance (Structured Output + Basic Guardrails): The system needs to ensure the output (e.g., the JSON payload routing the ticket) is perfectly formatted, but it cannot afford the latency of having a second LLM evaluate the first.

- **Setting:** High structural conformity, low cognitive verification.
- **Tool:** Native Structured Output (Layer 22) enforces type safety on the LLM's payload. A small safety classifier such as Llama Guard (Layer 15) provides a basic safety net, with no general-purpose model acting as a judge.

6. State persistence (Checkpointing): If the agent fails to reproduce a bug, the entire process should not crash and start over.

- **Setting:** Orchestrator-managed checkpoints.
- **Tool:** Temporal's event history natively provides checkpointing. If an LLM node encounters an API timeout, the orchestrator retries that specific node without losing the context gathered in the preceding steps.

7. Capability evolution (Static): The system must remain predictable. Autonomous self-improvement (the agent writing its own new tools) introduces non-determinism that a first deployment cannot tolerate.

- **Setting:** Static capabilities with manual updates.
- **Tool:** Standard Git workflows. Prompts, tools, and SOPs are updated manually by the engineers based on the agent's failure traces.

8. Human control (Exception-based Routing): The agent runs autonomously on the happy path to preserve the human attentional budget.

- **Setting:** Halt and escalate.
- **Tool:** A programmatic threshold. The agent is instructed to halt and request human intervention only when its confidence score falls below a set threshold, or when it exhausts its retry budget in the sandbox. The ticket is then routed directly to the human on-call engineer, accompanied by the agent's execution trace.

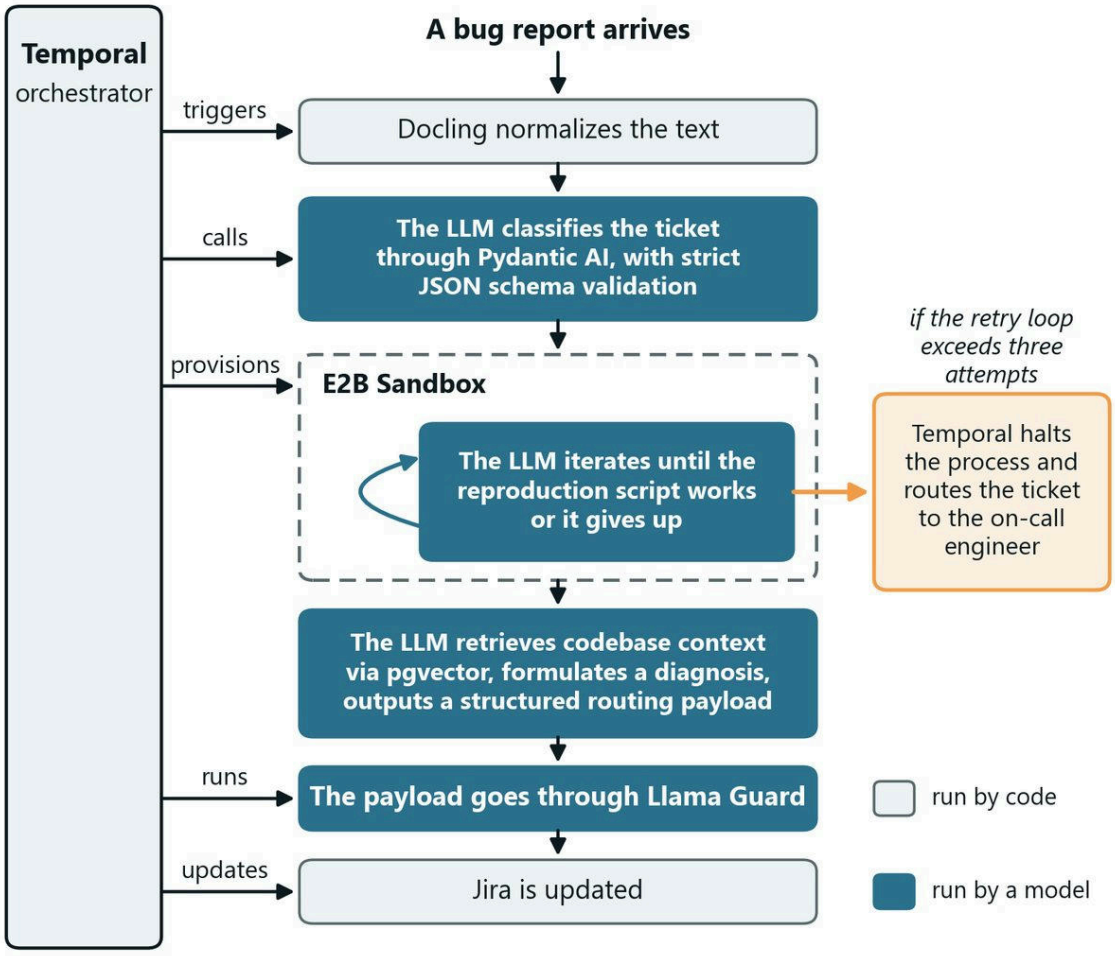


FIGURE 32.2 The assembled pipeline for the bug-triage case, drawn by this report, with the steps run by code and the steps run by a model in two fills. A deterministic orchestrator sequences the stages. Llama Guard is a small classifier model, not a general-purpose judge.

The assembled pipeline and its variations

In practice, this pipeline operates as a sequence of deterministic steps interspersed with cognitive nodes.

When a bug report arrives, Temporal triggers Docling to normalize the text. Temporal then calls the LLM (through Pydantic AI) to classify the ticket using strict JSON schema validation. To reproduce the issue, Temporal provisions an E2B Sandbox. The LLM acts within the sandbox, iterating until the reproduction script works or it gives up. It then retrieves relevant codebase context via pgvector, formulates a diagnosis, and outputs a structured routing payload. Temporal runs this payload through Llama Guard and updates Jira. If the retry loop exceeds three attempts, Temporal halts the process and routes the ticket, with the execution trace, to the on-call engineer.

This structure is highly applicable across a company, but **it cannot be a one-size-fits-all stack**. It must differ slightly depending on the process:

- **Routine Data Extraction:** For a process like extracting normalized data from unstructured vendor emails, the Action capability lever is turned down entirely. The E2B Sandbox is removed because code execution is unnecessary. Verification relies entirely on Structured Outputs, dropping Llama Guard since the risk of dangerous outputs from invoice extraction is zero.
- **Code Review and Lint-Fixing:** For a process like automated pull-request review, the Context lever requires a broader view of the repository. Docling might be replaced by direct AST parsing tools, and the Control Loop might be triggered by GitHub Actions rather than Temporal.

This reference architecture is designed for a reasonable balance on the CLEAR dimensions, and that balance is a prediction until it is measured. It maintains a low Cost Per Solved Task by avoiding multi-agent chatter and expensive evaluators. Its Latency is acceptable for asynchronous internal tasks. Its Efficacy should be high because a frontier model is used on narrow, bounded steps. Its Assurance is maintained by the physical isolation of the sandbox. Its Reliability should benefit from the deterministic orchestrator, which prevents the unrecoverable state failures common in purely LLM-driven loops. It is the baseline from which all deviations must be justified.

Delegation boundary

The boundary in this reference instantiation is clear: the system executes the deterministic routing and the bounded cognitive extraction, but the human engineer retains the final authority to actually merge the code fix and resolve the ticket.

This instantiation is one reading of the matrix, for one team and one process. The final chapter sets out the matrix in full.

CHAPTER 33

The result: a situational matrix, a frontier that moves, and five gaps

This report set out to separate the agentic methodologies, judge them on evidence, and arrive at one way of choosing among them. This last chapter gathers what the inquiry produced. It recalls the findings part by part, presents the matrix that follows from them, and then states two limits on that result. The frontier moves on every lever, and five questions remain open.

What the report established

Part I set the ground. Agentic AI is an assembly built around a model, not a new kind of model, and the report fixed the vocabulary it would use. Part II went down to the first principle, the loop of tool use, planning and reflection, then built the **eight invariant levers** on top of it and tested them. They separate real systems, and they do not sort them into species. Part III read the landscape of patterns, frameworks and protocols as settings of those levers, and found most vendor claims unmeasured. Part IV traced the structures of agentic systems back to human organizations. It showed that the hardest problems, such as prompt injection and unfaithful reasoning, come from what the model does not share with a human. Part V drew the conclusion. What decides between methodologies is their **economic boundary**, and the right setting depends on the situation of whoever pays.

The three hypotheses that carried the reasoning end as follows. The first holds as a design grid and fails as a taxonomy: the levers describe any system, and they class none. The second is half true: the resemblance to human organizations is real, and nothing makes the convergence necessary, since a layer of problems belongs to the model alone. The third stays open: no evidence yet says whether the delegation boundary is a technical limit or a permanent one, and that indecision is itself the finding.

Three dimensions: structure, situation, measure

Those findings add up to a principle with three dimensions.

1. **Structure, the eight levers.** The questions any agentic system has to answer: Context, Action capability, Control loop, Decomposition and specialization, Verification and conformance, State persistence, Capability evolution, and Human control.
2. **Situation, the four criteria.** What differs from one organization to the next: risk tolerance, process predictability, human attentional budget, and velocity requirement.
3. **Measure, the CLEAR dimensions.** The check that a chosen structure fits a given situation: cost, latency, efficacy, assurance and reliability.

The matrix, the fruit of the research

The matrix in Table 33.1 is what the whole inquiry produces. Every column comes from a different part of the report. The question on each row is the lever as Part II defined it. The default, the dial and the cost come from the decision rules of chapter 29, which rest on the evidence of Parts II and III. The last column, what stays with a human, is the delegation boundary that Part IV explained and chapter 26 consolidated.

TABLE 33.1 **The situational matrix**

Lever	The question it asks	Default setting	What justifies turning the dial	What turning it costs	What stays with a human
Context	What does the agent know when it decides?	Just-in-time retrieval	Low process predictability	Tokens, and diluted attention	Deciding what is worth writing down
Action capability	What can it do that cannot be undone?	Read-only; a person applies	High velocity, with risk tolerance to match	Exposure; a guarantee costs utility	Defining the perimeter
Control loop	Does code or the model decide the next step?	Code decides	Low process predictability	Unbounded, unpredictable runs	Deciding how wide the model's space is
Decomposition and specialization	One executor or several, and along which cut?	One agent that writes alone	Contexts that work against each other	Coordination: about 15× the tokens of a chat	Choosing the cut and the roles
Verification and conformance	How are the product and the process checked?	Deterministic checks	Low risk tolerance	Tokens, latency, false alarms	Writing the reference to check against
State persistence	What survives the turn, the session, the crash?	Session-scoped	Tasks that span days or wait for a person	Complexity, and data to govern	Fixing what is kept and what is deleted
Capability evolution	What changes what the agent can do?	Static prompts and curated skills	A team too small to tune by hand	Drift, which has to be evaluated	Admitting a capability for good
Human control	Where does a person see and intervene?	A few gates before consequential actions	Volume beyond what people can read	Errors found after the fact	Approving the irreversible, and answering for it

The matrix is used in three moves. The first is to describe the situation honestly on the four criteria, which is harder than it sounds, since most organizations overstate their tolerance for risk until something breaks. The second is to start every lever at its default and turn a dial only where the situation gives a reason, accepting the cost in the next column. The third is to measure the result on the five CLEAR dimensions and to revise. A spreadsheet distributed with this report extends each row with the tools and catalogue layers of chapter 29.

Used this way, the matrix leads to a conclusion that may surprise after two chapters of tool names: **the specific tool does not matter.** What matters is the ability to describe the situation and to put the right configuration in front of it. An orchestrator chosen today may be obsolete in two years. The need for a control loop held by code, where an error would be costly, will still be there.

One practice cuts across all eight rows and deserves its own mention, **compartmentalization**. The designs that hold up keep things apart: the reasoning from the execution environment, the agents that read from the one that writes, the place where a skill is tried from

the registry it may enter. Chapters 7, 12 and 19 each arrived at a version of it. As processes grow, deciding what must not touch what becomes as important as deciding what each part does.

The frontier is moving on every lever

The defaults in Table 33.1 are dated. They describe what was sensible in 2026, and on every lever the ground has shifted during the period this report covers. Table 33.2 gives one documented example per lever.

TABLE 33.2 **One recent shift on each lever**

Lever	What moved	What it changes
Context	In March 2026 two Claude models received a context window of one million tokens at standard pricing (Anthropic)	A whole codebase fits in one prompt. The dilution measured in chapter 6 still applies, so fitting is not the same as attending
Action capability	In October 2025 Claude Code gained sandboxing at the level of the operating system, isolating both files and network; its maker reports 84% fewer permission prompts in internal use (Anthropic)	Containment replaces prompts, the move chapter 7 argued for
Control loop	METR measured that the length of tasks agents complete autonomously, at 50% reliability, has doubled about every seven months for six years (METR)	The range of tasks where a model can hold the loop keeps widening
Decomposition and specialization	In February 2026 Claude Code introduced agent teams, several agents working in parallel, recommended for independent, read-heavy work (Anthropic)	Several agents become a product feature, in the read-only form the single-writer principle predicts
Verification and conformance	Shopify raised the agreement between its model judges and human raters from near chance to close to the agreement between humans (Shopify)	A model can be calibrated as a judge of outputs. Conformance of the process remains untooled
State persistence	In July 2025 Temporal announced durable execution for agents built with the OpenAI Agents SDK, generally available since March 2026 (Temporal)	Resuming a failed run from where it stopped becomes a standard integration
Capability evolution	In October 2025 Anthropic introduced Agent Skills, folders of instructions, scripts and resources that a model loads when needed (Anthropic)	Skills become a packaged, shareable unit, which makes the question of who admits them more pressing
Human control	In March 2026 Claude Code gained a mode in which a classifier evaluates each action before it runs, "acting as a substitute for a human approver". Its maker reports that it misses 17% of the overeager actions in a real sample (Anthropic)	The gate itself is being automated, with a measured miss rate

Two readings follow from the table. The first is that the levers are the stable part of this report and the settings are the perishable part. Every row of Table 33.2 changes an answer. None changes a question. A new capability still has to be placed on the eight dials, and the placing still depends on the situation.

The second reading is that the levers move together. Two of the examples describe one and the same trade: stronger containment on the Action capability lever is used to justify fewer gates on the Human control lever. A reader who updates one row of the matrix should look at its neighbors.

For a practitioner the consequence is a habit. The matrix has to be rerun as the tools and models change, and its defaults should be treated as positions held for a while.

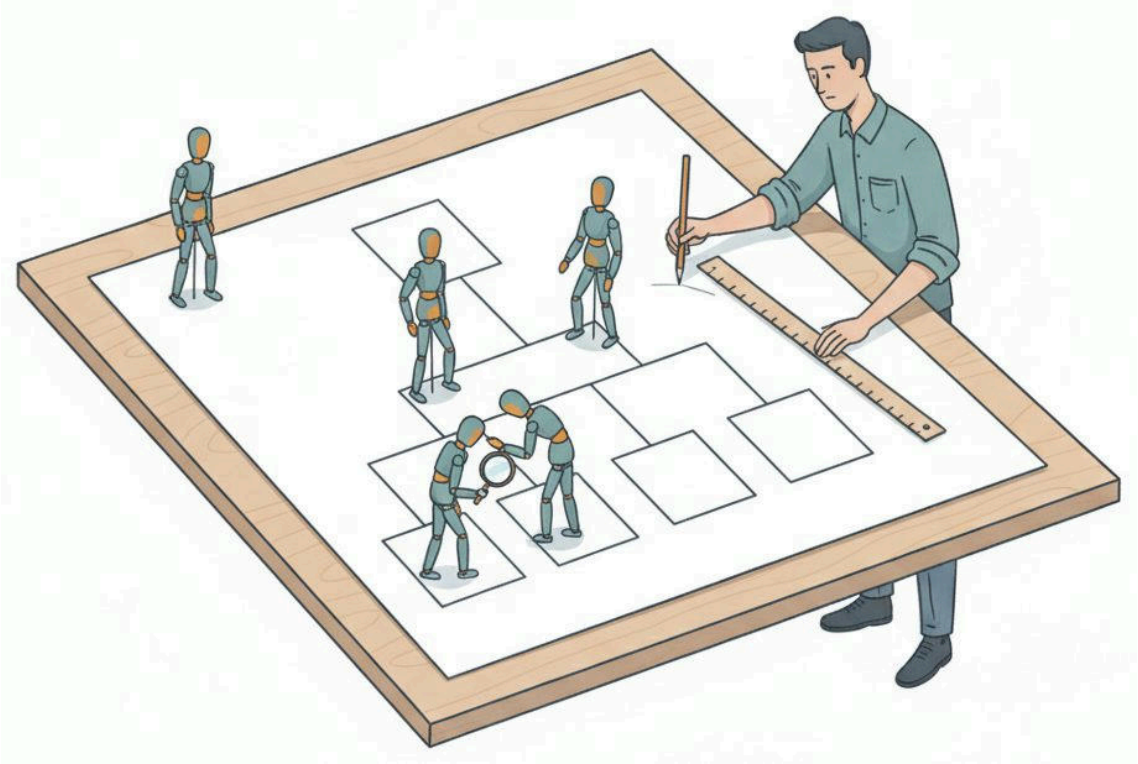


FIGURE 33.1 The designer of an agentic system is drawing an organization chart, an editorial allegory by this report. One of the new workers is already checking another's work. The comparison with human organizations is the one made in arXiv 2512.07665.

The human inspiration

One finding holds across every part of the report. These systems draw on human organizations, and they do so heavily. Designing an agentic system means rebuilding, in small, problems that factories and companies have faced for a long time. The debates of the field are the debates of organization theory: efficiency against agility on a production line, the separation of powers between those who act and those who verify, the chain along which a decision travels. The designer of an agentic system is also drawing an organization chart, for workers of a kind that no chart was drawn for before.

The gaps this methodology accepts

The methodology has limits, and they are stated here so that no reader has to discover them.

1. Trajectory conformance is not tooled. The CLEAR dimensions measure what an agent produced. Nothing in the corpus checks, while it runs, that an agent followed the procedure it was given.

2. Prompt injection remains unresolved. The methodology relies on containment because the underlying flaw, a model's inability to tell an instruction from data, has no reliable fix. Every published defense of the detecting kind fell to an attacker who adapted.

3. No comparison at constant model and budget. Until two architectures are compared with the same model and the same token budget, the benefit of any structure remains entangled with what it spent.

4. The delegation boundary is undecided. Whether what an agent cannot carry today, such as accountability, long-term incentives and reputation, marks a technical limit or a permanent one remains open, as chapter 25 concluded.

5. The methodology is untested. The matrix, the two profiles and the reference instantiation are syntheses of the evidence gathered here. None has been run in production by the author, and the CLEAR dimensions have not been applied to them.

These five gaps mark where the evidence stops. They are also the list of what would have to be measured for the matrix to become more than a reasoned proposal. Until then it should be used as one: eight questions, a way of answering them for a given situation, and an instrument for finding out whether the answer was right.

Sources

Aggregated automatically from the links in the report; to be checked and completed by the verifier.

A2A and MCP
a2a-protocol.org/latest/topics/a2a-and-mcp/

A2A releases
github.com/a2aproject/A2A/releases

A2A Specification
a2a-protocol.org/latest/specification/

Agent Client Protocol
agentclientprotocol.com/

AgentEval, arXiv 2604.23581
arxiv.org/abs/2604.23581

agents.md
agents.md/

agntcy/acp-spec
github.com/agntcy/acp-spec

AI Act
digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai

AI Act, art. 14
artificialintelligenceact.eu/article/14/

announcement v1.0
a2a-protocol.org/latest/announcing-1.0/

Anthropic
anthropic.com/research/reasoning-models-dont-say-think

Anthropic
anthropic.com/engineering/multi-agent-research-system

Anthropic
anthropic.com/engineering/how-we-contain-claude

Anthropic
anthropic.com/engineering/building-effective-agents

Anthropic
claude.com/blog/context-management

Anthropic
anthropic.com/news/donating-the-model-context-protocol-and-establishing-of-the-agentic-ai-foundation

Anthropic
claude.com/blog/1m-context-ga

Anthropic
anthropic.com/engineering/claude-code-sandboxing

Anthropic
anthropic.com/news/claude-opus-4-6

Anthropic
claude.com/blog/skills

Anthropic
anthropic.com/engineering/claude-code-auto-mode

Anthropic / AAIF
anthropic.com/news/model-context-protocol

Anthropic, *Effective context engineering*
anthropic.com/engineering/effective-context-engineering-for-ai-agents

arXiv 2210.03629
arxiv.org/abs/2210.03629

arXiv 2406.13352
arxiv.org/abs/2406.13352

arXiv 2501.06322
arxiv.org/abs/2501.06322

arXiv 2506.17208
arxiv.org/abs/2506.17208

arXiv 2510.09023
arxiv.org/abs/2510.09023

arXiv 2512.07665
arxiv.org/abs/2512.07665

arXiv 2603.14805
arxiv.org/abs/2603.14805

arXiv 2603.25723
arxiv.org/abs/2603.25723

arXiv 2605.06614
arxiv.org/abs/2605.06614

arXiv 2605.09104
arxiv.org/abs/2605.09104

arXiv 2605.16278
arxiv.org/abs/2605.16278

arXiv 2605.27366
arxiv.org/abs/2605.27366

arXiv 2606.04306
arxiv.org/abs/2606.04306

arXiv 2606.08755
arxiv.org/abs/2606.08755

arXiv 2606.23127
arxiv.org/abs/2606.23127

CAMEL
arxiv.org/abs/2303.17760

CaMeL, arXiv 2503.18813
arxiv.org/abs/2503.18813

ChatDev
arxiv.org/abs/2307.07924

Chroma Research
research.trychroma.com/context-rot

CLEAR framework
arxiv.org/abs/2511.14136

CoALA, arXiv 2309.02427
arxiv.org/abs/2309.02427

Cognition
cognition.com/blog/multi-agents-working

Cognition
cognition.com/blog/dont-build-multi-agents

Generative Agents
arxiv.org/abs/2304.03442

Google
github.com/google/adk-docs/blob/main/docs/agents/workflow-agents/index.md

Google Cloud
docs.cloud.google.com/architecture/choose-design-pattern-agentic-ai-system

Google Developers Blog
developers.googleblog.com/en/google-cloud-donates-a2a-to-linux-foundation/

Google Open Source Blog
opensource.googleblog.com/2026/04/a-year-of-open-collaborati-celebrating-the-anniversary-of-a2a.html

Google Research
research.google/blog/towards-a-science-of-scaling-agent-systems-when-and-why-agent-systems-work/

Google, *adk-docs*
github.com/google/adk-docs/blob/main/docs/agents/index.md

Google, *adk-docs*
github.com/google/adk-docs/blob/main/docs/sessions/index.md

HashiCorp
developer.hashicorp.com/terraform/cli/commands/plan

i-am-bee
github.com/orgs/i-am-bee/discussions/5

IBM
ibm.com/think/topics/agentic-ai

IETF draft
datatracker.ietf.org/doc/html/draft-klrc-aiagent-auth-00

IMDA
imda.gov.sg/-/media/imda/files/about/emerging-tech-and-research/artificial-intelligence/mgf-for-agentic-ai.pdf

Inside the Scaffold, arXiv 2604.03515
arxiv.org/abs/2604.03515

Langfuse, *data model*
langfuse.com/docs/observability/data-model

LangGraph, *interrupts*
docs.langchain.com/oss/python/langgraph/interrupts

LangGraph, *persistence*
docs.langchain.com/oss/python/langgraph/persistence

LangSmith, *observability concepts*
docs.langchain.com/langsmith/observability-concepts

LF AGNTCY
linuxfoundation.org/press/linux-foundation-welcomes-the-agntcy-project-to-standardize-open-multi-agent-system-infrastructure-and-break-down-ai-agent-silos

Linux Foundation
linuxfoundation.org/press/linux-foundation-announces-the-formation-of-the-agentic-ai-foundation

Linux Foundation
linuxfoundation.org/press/a2a-protocol-surpasses-150-organizations-lands-in-major-cloud-platforms-and-sees-enterprise-production-use-in-first-year

MAST, arXiv 2503.13657
arxiv.org/abs/2503.13657

MCP Blog
blog.modelcontextprotocol.io/posts/2026-07-28-release-candidate/

MCP Specification
modelcontextprotocol.io/specification/2025-11-25

MCP versioning
modelcontextprotocol.io/specification/versioning

MetaGPT
github.com/geekan/MetaGPT

MetaGPT, arXiv 2308.00352
arxiv.org/abs/2308.00352

METR
metr.org/blog/2025-08-08-cot-may-be-highly-informative-despite-unfaithfulness/

METR
metr.org/blog/2025-03-19-measuring-ai-ability-to-complete-long-tasks/

Microsoft, *Agent Framework 1.0*
devblogs.microsoft.com/agent-framework/microsoft-agent-framework-version-1-0/

NIST AI Risk Management Framework
nist.gov/itl/ai-risk-management-framework

OASF
docs.agntcy.org/oasf/open-agentic-schema-framework/

OpenAI
developers.openai.com/blog/openai-for-developers-2025

OpenAI
openai.github.io/openai-agents-python/multi_agent/

OpenAI
developers.openai.com/commerce

OpenAI, *Codex agent approvals & security*
developers.openai.com/codex/agent-approvals-security

OpenAI, *Sandboxing*
developers.openai.com/codex/concepts/sandboxing

opentelemetry.io
opentelemetry.io/docs/specs/semconv/gen-ai/

Shopify
shopify.engineering/building-production-ready-agentic-systems

Springer
link.springer.com/chapter/10.1007/3-540-49057-4_1

Stanford CRFM
hai.stanford.edu/news/reflections-foundation-models

Stripe
stripe.dev/blog/minions-stripes-one-shot-end-to-end-coding-agents-part-2

Temporal
temporal.io/blog/announcing-openai-agents-sdk-integration

The Batch 241
deeplearning.ai/the-batch/issue-241/

Voyager, arXiv 2305.16291
arxiv.org/abs/2305.16291