Architecting Agentic Systems

Engineering Dependable AI Agents

Damian Beresford

Edition 1.1.0 · July 5, 2026

Contents

Edition 1.1.0 · Generated July 5, 2026.

Preface

The organizational mandate echoing through enterprise boardrooms is absolute: build autonomy. Automate the workflows. Deploy the agents.

The economic imperative behind this mandate is real. For the first time, software can take on open-ended, judgment-laden work that resisted automation precisely because it could not be reduced to fixed rules — triaging a messy ticket, drafting a considered reply, finding its way around an unfamiliar API. Done well, this collapses the cost of work that used to require a human in the loop. The pressure to capture that value is not just hype; it is a justified operational urgency.

But under the weight of this urgency, the software industry is sleepwalking into a structural mess.

Sound software engineering has always rested on a simple triad: build the right thing, build the thing right, and assume all things fail. In the rush to accelerate business impact, we are routinely failing on all three fronts. We are building the wrong things — prescribing highly complex, unmanageable multi-agent swarms for problems that require a simple router. The point-and-click agent builders only sharpen the temptation: when conjuring an autonomous agent takes a few clicks, every problem starts to look like one. We are building them wrong — treating conversational logs as databases and system prompts as security boundaries. And we are operating with reckless optimism, deploying open-ended reasoning loops into production with the hope that the model will behave like a careful junior engineer. It will not.

The danger is not that foundation models do not work. The danger is that they work *just well enough* in a stage-managed demo (*AI theater*) to be trusted with things they cannot structurally guarantee once that demo meets production.

I wrote this book because I watched technical leaders struggle without a treatment that integrated bounded autonomy, governance, memory, the trace, the harness, and the skills boundary as one subject rather than scattered across pattern catalogs and position papers. Where Gulli, Anthropic, CSIRO, and Andrew Ng catalog patterns and demonstrate frameworks, this book integrates the architectural discipline end to end — bounded autonomy as substrate, governance as load-bearing structure, failure-mode taxonomies, trace discipline, the harness, and the skills boundary. Governance-as-architecture is now consensus the book documents; the integration is the contribution. Judge the work on whether that integration holds, not on whether every position is novel.

Unless a case is named and sourced, incidents described in this book are composites of documented public cases, altered to remove identifying detail. Claims about the state of the field — adoption figures, dominant transports, model capabilities — are as of mid-2026; the architectural arguments are written to survive their revision (Chapter 1 makes the same shelf-life bet).

Many seasoned engineers lack vocabulary for probabilistic components; architects who see the risk struggle to hold the line without a formal framework to defend against hype and deadline pressure. This book is an attempt to provide that blueprint — for the engineer turning prototypes into enterprise-grade software and the architect who will carry the pager when these systems meet production.

The premise of the following chapters is uncompromising: an agentic system is not an AI project; it is a distributed systems engineering problem. The foundation models at the center of these systems are highly useful, highly volatile probabilistic components. They cannot be trusted to police their own behavior, adhere to corporate policy, or respect cost constraints. They are valuable *exactly insofar as* their behavior can be bounded, governed, observed, and recovered from by the deterministic infrastructure built around them.

If you are looking for a book on how to write the perfect prompt or how to fine-tune a model, this is not it. The frameworks will be obsolete in six months; the prompt-engineering hacks will be patched out in the next model weights update. This book assumes you already know how to build standard software. It does not teach you how to write a web server; it teaches you how to safely embed a non-deterministic agent into one. The harness that embeds the agent — the deterministic loop the model runs inside — is designed directly in Chapter 19.

The book is divided into four parts:

- **Part I (Foundations, Chapters 1–4)** defines what agentic systems are, fixes the agent definition in the context of such systems, names the structural forces every architecture must balance, and closes with a compressed map of the cognitive layer — including the harness concept — before architecture begins.
- **Part II (Architecture, Chapters 5–10)** develops the deterministic shell: bounded autonomy, governance pipelines, tiered memory, data ingestion, control and coordination, and the skills layer.
- **Part III (Production, Chapters 11–15)** applies production discipline: failure modes and anti-patterns, trace-driven testing, the glass layer, enterprise integration, and model routing.
- **Part IV (Synthesis, Chapters 16–19)** composes the whole: system-architecture vignettes, the Concord worked example, operational discipline — including retrofitting ungoverned agents (Chapter 18) — and the harness capstone (Chapter 19).

Autonomy is not a feature you turn on. It is an operational state that must be heavily guarded. To truly accelerate the business, we must engineer these systems not with the optimistic hope that they will succeed, but with the architectural certainty that they will fail, and that when they do, our infrastructure will catch them.

For the engineers, architects, and technical leaders tasked with delivering the future while protecting the present, I hope this book provides the vocabulary and the structure you need to do both.

Chapter 1: The nature of agentic systems

Software architecture is undergoing a structural shift. Traditional systems are built on deterministic control flow: explicit logic, predefined transitions, predictable state transformations. Agentic systems do not abandon any of this. They introduce a *probabilistic reasoning layer* inside the deterministic shell. Portions of the control flow are no longer fully encoded ahead of time, they are generated dynamically by reasoning components operating toward goals.

This is the architectural definition the rest of the book works from. It is not a marketing claim. It is a structural commitment with cost: the moment any control decision is made by a probabilistic component, the system has to be designed differently from a deterministic one. The book is about what those differences are and how to manage them.

The shift is comparable in scale to earlier transitions in software architecture, each of which produced its own pattern language for failure modes that had not existed in the prior world. The move from monolithic, transactional, single-machine applications to distributed systems brought partial failure, eventual consistency, and asynchronous communication; its pattern language, represented best by *Designing Data-Intensive Applications*, addressed split-brain failures, message reordering, retry storms, and consistency models. The move from desktop applications with full local state to web applications with stateless servers and rich clients produced its own literature on session management, caching, and idempotent design. The transitions could be multiplied — time-sharing, object orientation, virtualization, cloud — but the pattern repeats: a new class of failure appears, and the discipline takes a generation to catch up.

The transition now underway is of that kind. The novel architectural concern is not distribution or statelessness; it is that *some control decisions are made by a component whose behavior is not fully described at design time*. The component is correct most of the time, plausible nearly all of the time, and occasionally wrong in ways that are hard to anticipate. The literature on how to build reliable systems around such components is still being written. This book is one entry in it.

What "agentic" names

The word *agentic* has been used to describe anything from a chat interface with a function-calling capability to a multi-agent swarm composing services through emergent negotiation. In this book the term names systems with two defining properties:

1. They contain at least one **agent** as formally defined in Chapter 2.

2. The agent's decisions affect **control flow**, what runs, in what order, with what arguments, not only the *content* of an output.

To those two descriptive properties the book adds two architectural commitments, the claims it spends most of its length defending:

1. The agent operates inside **deterministic infrastructure** that provides durable state, tool adapters, policy enforcement, and observability.
2. The agent's loop is **bounded** by explicit limits on iterations, cost, and risk, enforced from outside the agent itself.

The split between the two pairs is deliberate. The properties are descriptive: they tell a reader whether the patterns in this book apply to their system at all. The commitments are normative: they are the book's thesis about what such a system needs to be reliable in production. A system with the two properties but without the two commitments is not a non-agentic system, it is an unbounded, ungoverned one, which is precisely the subject of the failure-mode catalog (Chapter 11) and the retrofitting guidance (Chapter 18), not a counterexample to the book.

The descriptive properties still do real work at the boundary. A pipeline that runs a model as a single deterministic step is not an agentic system, it is a workflow with a probabilistic component, because the model produces content, not control flow (the second property is absent). A model that calls tools but carries no state across turns is closer to a stateless router than to an agent (the first property is absent; see Chapter 2). Each is a useful construction, but the patterns that govern it are not the patterns of agentic systems, and applying them adds ceremony without benefit.

This narrowing matters because the patterns in the book are not free. Each pattern carries an implementation cost, an operational cost, and a cognitive load on the team. Applying them to systems that are not genuinely agentic produces complexity without payoff. Applying them late, after the system is already in production, produces incidents. The narrowing is also a service to the reader: a senior engineer reading the book should be able to ask, of their system, whether it has the two defining properties, and to receive a clear answer that determines whether the rest of the book applies.

The probabilistic layer inside the deterministic shell

The defining structural fact of an agentic system is that *one or more decision points in the control flow are made by a probabilistic component*: the model, not a static routing table, decides whether to call a tool or ask a human, whether to retry a failed step or route around it, and whether the task is finished. Everything else follows.

In a deterministic service, control flow is statically describable. Given an input, the path through the code can be enumerated. Errors are exceptional cases, situations the designer anticipated and handled. Tests assert exact outputs. Tracing is for diagnostics, not for design correctness. The hard parts of the system are about scale, latency, and correctness under concurrency; the patterns are about persistence, queuing, replication, and consistency.

In an agentic system, control flow is partially synthesized at runtime. Given an input, the path through the system is one of many possible paths, chosen by the reasoning component based on state, goal, tools, and prompt. The system can still be described, but the description is now over *distributions* of paths, not single paths. Errors are not exceptional, they are part of the operating

profile. Tests assert *properties* (boundedness, policy compliance, action safety) more often than exact outputs. Tracing is now structural, it is the only complete record of what the system actually did, because the system's behavior cannot be fully derived from its inputs and code.

This is the architectural reframing the book asks the reader to internalize. Once a probabilistic component is on the control-flow path, the surrounding system must:

- Treat **boundedness** as an explicit property rather than a default of the runtime (Chapter 5).
- Treat **governance**, validators, policy gates, oversight, as load-bearing structure rather than bolt-on safety (Chapter 6).
- Treat **state** as a first-class memory architecture rather than a side effect of the prompt (Chapter 7).
- Treat **failure modes** as part of the design surface rather than as bugs (Chapter 11).
- Treat **traces** as the system of record rather than as logs (Chapter 12).

Each of these is a deliberate engineering choice with a cost. None are optional in a reliable agentic system. The cost compounds, a small system that violates one or two of these can survive on luck and careful design; a system at scale violates them at its peril, and the operating bill is paid in incidents.

Why existing software patterns are insufficient

The Gang of Four, Fowler, and the data-intensive-systems literature describe patterns for deterministic systems. They remain valid for the deterministic substrate of an agentic system, the data stores, the queues, the API adapters, the transaction boundaries. They do not, however, cover the part of the system where the control flow is generated by a probabilistic component.

The shortfall has a specific shape:

Patterns for deterministic flow assume the decision-maker is known at design time. In an agentic system the decision-maker is a language model whose behavior can shift between releases. Architectural commitments must therefore be conservative about what the decision-maker is responsible for, and where the deterministic substrate is responsible for enforcing limits. The patterns of the deterministic literature assume "the application decides"; the patterns of the agentic literature assume "the model decides, within bounds the application enforces."

Patterns for deterministic state assume a finite, enumerable state machine, a fixed set of states with predefined transitions between them. In an agentic system the *transitions* are open: the model can synthesize novel actions, combinations, and sequences, producing paths through the system that the designer never encoded. A deterministic application's state space may itself be effectively infinite, any system that accepts free-form input has one, so the distinguishing problem is not the size of that space but the open set of transitions across it. Architectural commitments must therefore bound the *envelope* of permitted transitions rather than enumerate them. A bounded agent is not described by a finite state machine, it is described by a finite set of *constraints*, and the path it takes within them is open.

Patterns for deterministic error handling assume the failure modes are known. In an agentic system failure modes can emerge from the interaction between the model and the tools, and can vary by model version. A model upgrade can change which prompts are reliable and which fail subtly. The literature on testing such systems is younger than the literature on deterministic testing and

accommodates this reality with replay-driven and property-based techniques (Chapter 12). The book devotes Chapter 11 to the taxonomy of failure modes specific to agentic systems.

Patterns for deterministic auditability assume reconstruction is straightforward. A deterministic system's behavior can be reconstructed from its inputs and code; an agentic system's cannot. Even with a pinned model version and temperature set to zero, provider-side nondeterminism and silent model updates mean the same inputs need not reproduce the same behavior, so the output is not reliably a function of the visible inputs. The trace becomes the system of record, not an auxiliary log; without it, an incident is unreconstructable.

The contribution of *agentic* design patterns is not to replace deterministic patterns. It is to add the layer above them that turns a probabilistic component into a system property: bounded, governable, observable, replayable. The deterministic patterns are still required underneath; they are not sufficient on their own.

A note on reasoning models and pattern erosion

The patterns of agentic systems fall into two categories that are easy to conflate. *Cognitive patterns* govern how a model reasons inside a single agent, how it is prompted to plan, act, reflect, and self-critique. *Architectural patterns* govern the system around the model, how it bounds, governs, persists, and observes that reasoning. The distinction matters because the two are moving in opposite directions, and the rest of this section is about that divergence.

Between 2024 and 2026 the architectural relevance of several once-canonical agentic patterns shifted. Patterns originally proposed as scaffolding around weaker models — explicit ReAct loops, externalized Plan–Execute, hand-rolled reflection cycles — have been partially absorbed into the models themselves. A modern reasoning model emits intermediate thought structure, plans across long horizons, and self-critiques without external orchestration code.

This does not make those patterns wrong. It moves them. What was once an architectural concern (writing the loop) is now a model concern (the loop is internal). What remains architectural is everything around the loop: bounding it, governing it, persisting its state, observing it, and recovering from its failures. That deterministic envelope around the loop is the *harness*, the term this book uses for the program that turns a model into an agent. Chapter 4 introduces it; Chapter 19 designs it in full.

The implication for the rest of the book is that emphasis goes to the structural patterns, bounded autonomy, governance, memory, skills, anti-patterns, trace discipline, and the cognitive layer is treated briefly, with deference to canonical sources for the patterns themselves. This is a deliberate response to the field as it stands in 2026, not an editorial omission.

There is also a forward-looking reading of this erosion: as models continue to internalize reasoning patterns, the *architectural* surface around agents stabilizes while the *cognitive* surface inside them keeps changing. A book that cataloged cognitive patterns at full depth would date quickly; a book that develops the architectural discipline, what is around the model, has a longer shelf life. The book is structured accordingly.

What this book does, and what it does not

The book defines an agent (Chapter 2), names the forces that agentic patterns must balance (Chapter 3), maps the cognitive layer briefly in Chapter 4, then proceeds through architecture (Part II), production discipline (Part III), and synthesis (Part IV). It treats the cognitive layer briefly because canonical sources cover that ground. It treats bounded autonomy, governance, the ingestion pipeline, the glass layer, enterprise SaaS integration, model routing, anti-patterns, trace discipline, and the Skills layer at depth because those are the topics where the existing literature is thin.

The book does not:

- Prescribe a framework. The agent frameworks and vendor SDKs in common use are mentioned only where unavoidable, and none is endorsed. Patterns are architectural and outlive frameworks, naming the framework-of-the-month would date the book faster than anything else in it.
- Document prompts. The prompt engineering literature is large and moves fast. The book stays above the prompt layer.
- Enumerate every pattern. Gulli (2025) and the CSIRO catalog (2025) already do this well. The book's catalog is the subset that admits structural treatment; the rest is referenced.
- Defend autonomy as an end in itself. Autonomy is a means; the end is reliable execution. The book consistently argues for *less* autonomy than the team is tempted to grant, and for bounding the rest.
- Promise to make systems safe. Safety is a property of operating discipline as much as of architecture; the book provides the architectural substrate and points to the operational practice (Chapter 18), but no architecture is safety on its own.

The book *does*:

- Insist on a formal definition of *agent* that excludes a great many systems currently labeled as such.
- Treat the governance layer as structural architecture, not as a compliance afterthought.
- Make failure modes a first-class subject.
- Make trace discipline a first-class subject.
- Place the Skills layer, dynamic capability loading, of which the agentskills.io standard is one implementation (Chapter 10), in its architectural context, including the argument that skills are subordinate to architecture, not a way around it.
- Take editorial positions where the literature is unclear or split, and name them as positions rather than as defaults.

A worked example: Concord

The book accompanies its architectural development with a single worked example, threaded throughout: **Concord**, a coding-assistant agent. Concord helps engineers make changes to a codebase: given a task (implement a feature, fix a bug, refactor a module), it proposes a change, runs the project's tests against it, and submits the change for human review. It operates in a sandbox; it never commits, pushes, or deploys without explicit human approval.

Concord exists in this book for two reasons. First, the coding-assistant shape is the most widely-deployed class of agentic system in 2026, readers will recognize it. Second, Concord exercises the core architectural concerns the book develops: bounded autonomy on all six axes, a load-bearing

governance layer, tiered memory, control structure, the Skills layer, failure-mode defenses, trace discipline, and operational practice. By Chapter 17 the reader will have seen Concord developed end-to-end with concrete artifacts (bounding YAML, governance pseudocode, skill manifests, trace excerpts, policy tables) that they can adapt to their own systems.

Concord is fictional. The shape is real and the artifacts are realistic. Where a chapter benefits from a concrete example, Concord is named; the full walkthrough is in Chapter 17.

Concord is the book's single *threaded* example, chosen so that one system can be followed end to end as each layer is developed, but it is not the only concrete material. Chapter 16 sketches six other system shapes as vignettes (a research agent, a customer-support platform, an operations controller, and more), and Chapter 14 works through enterprise-integration scenarios in their own right. A reader whose system looks nothing like a coding assistant will find a closer match among those.

The shape of the argument

Chapter 2 fixes the term *agent* with a four-property definition and the boundary cases that the definition excludes. Chapter 3 names the architectural forces that agentic patterns must balance, the structural tensions that drive every later design decision.

Chapter 4 maps the cognitive layer in compressed form — deliberately brief, because erosion has moved much of it into the models — and introduces the harness concept. Chapters 5–10 develop the architectural discipline: bounded autonomy, governance, memory, the ingestion pipeline that feeds it, control and coordination, and the Skills layer. Chapter 5 and Chapter 6, in particular, are the load-bearing chapters of the book.

Chapters 11–15 turn to production discipline: failure modes, testing and trace discipline, the glass layer, enterprise SaaS integration, and model routing. Chapters 16–19 synthesize: system-architecture vignettes, the Concord worked example (Chapter 17), operationalization (Chapter 18), and the harness design (Chapter 19) that the earlier chapters attach to. The Glossary (Chapter 20) and Annotated Bibliography (Chapter 21) are reference; the bibliography is the single point of departure for readers who want the full hands-on or academic treatment of any pattern the book covers in compressed form.

Readers should expect to find themselves disagreeing with some of the book's editorial positions. The book argues that multi-agent coordination is over-prescribed; that prompt-based governance is unreliable; that several once-canonical cognitive patterns have eroded; that Skills are subordinate to architecture rather than parallel to it. These are positions, not consensus. They are stated as positions so the reader can decide whether to accept them; where they hold, they shape what good architecture looks like, and the rest of the book follows.

> **The book's core commitment** The architectural commitment underlying the entire book can be stated in one sentence: **probabilistic reasoning components are useful exactly insofar as their behavior can be bounded, governed, observed, and recovered from, by deterministic infrastructure around them.** Everything that follows is an elaboration of that sentence.

Chapter 2: The agent: Formal definition and properties

This chapter fixes the term *agent* for the remainder of the book. It is a narrowing chapter, not the center of gravity. The book's actual subject is the **agentic system**, defined in Chapter 1 as a system that embeds probabilistic reasoning components inside deterministic infrastructure; the thesis that reliability lives in that deterministic shell, not in the model, is what the rest of the book develops. The agent is the component whose definition this chapter fixes, for one reason: a reader needs to know whether the system in front of them is the kind of system this book's patterns address. The four properties below are the discriminator that answers that question. They are deliberately strict so that architectural decisions can be reasoned about without ambiguity, and so that applying the book's discipline to a system that is *not* an agent adds ceremony without benefit — a cost the preface warns against.

The definition here is intentionally structural. It is not tied to any particular model, framework, or prompting strategy. It is also not a claim about general intelligence; the book's concern is **bounded autonomy**: explicitly constrained reasoning and action loops with cost, iteration, and risk limits enforced by architecture (Chapter 5).

Formal definition

An **agent**, as defined in this book, is:

> **Definition: Agent** A stateful, goal-directed computational entity that reasons about possible actions, selects among them, executes actions in an environment, and updates internal state based on feedback.

This definition is intentionally compact. It excludes systems that merely *compute* outputs from inputs, and it excludes systems that can only *talk* but cannot act. It also excludes systems that can act but do not close the loop by incorporating feedback into subsequent decisions.

The definition is structural, not behavioral. It says nothing about whether the agent uses a large language model, a symbolic reasoner, a rule engine, or some combination. It says nothing about the model's size or capability. What it asserts is the *shape* of the component: a closed loop with state, goal-directed choice, action, and adaptation. Any system with this shape is an agent under this book's terminology; any system without all four properties is not.

The agent boundary

An agent is a logical boundary around a closed loop of (1) state, (2) goal-directed choice, (3) action, and (4) feedback-driven update.

A useful architectural test is whether you can draw an “agent box” and identify what crosses its boundary.

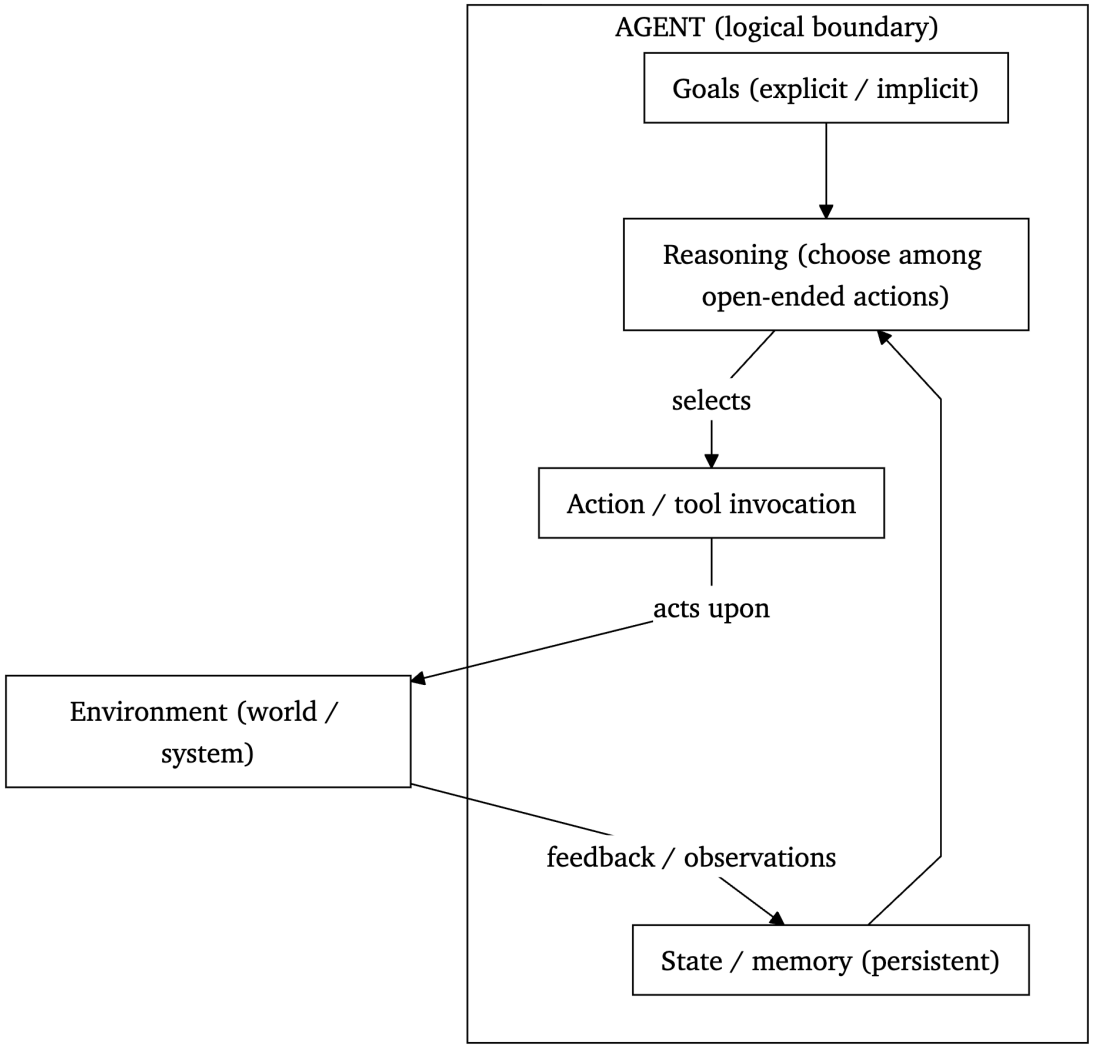

Figure 1: Figure 1. The agent boundary

Several later patterns (Control, Governance, Memory) can be understood as *externalizing* parts of this box into deterministic infrastructure while leaving the agent with bounded decision-making. This is the payoff the definition earns: once the agent is a closed loop of state, choice, action, and feedback, the rest of the book is about which parts of that loop the architecture pulls out and enforces deterministically, so that the probabilistic component is left to reason inside limits it cannot widen. The agent boundary is therefore not a hard line drawn around a process; it is a logical line drawn around the closed loop of choice and feedback, and most of what makes an agent reliable in production lives outside that line, not inside it.

Four distinguishing properties

The definition above can be operationalized as four properties. Remove any one of them and the system ceases to be an agent under this book's terminology.

1. Statefulness across steps

Statefulness means the agent carries information across reasoning/action iterations in a way that affects future choices. The state can be small (a handful of variables) or large (structured memory), but it must persist beyond a single function call.

Architecturally, this implies that the agent cannot be evaluated only as a pure mapping `input -> output`. The system must decide where state lives (in-process, external store, or a dedicated memory subsystem; see Chapter 7), how it is updated (append-only logs, snapshots, derived summaries), and how it is validated and governed (Chapter 6). Two confusions are worth disposing of quickly: having *access* to history is not the same as being stateful, since a stateless service can read a log and still behave as a pure function of the current request; and *caching* is not state, because it improves performance without representing commitments, hypotheses, or evolving beliefs. The practical test is whether a fact the agent established earlier can change what it does later. If the component's behavior depends only on the current input and freshly queried data, it may be automated, but it is not operating as an agent. Whether that state also survives a process restart is a separate, production-strength requirement, taken up in Chapter 7.

2. Explicit or implicit goals

Goal-directedness means the agent's behavior is evaluated against a notion of success that is not fully captured by the immediate input. The goal may be explicit (a declared objective and constraints) or implicit (policy-derived, role-derived, or inferred from a task description), but it must exist as something the agent can use to select among alternatives.

The selection is among *open-ended* alternatives, actions and sequences not reducible to a fixed, enumerable set of branches known at design time. This is the line between a goal-directed agent and a deterministic controller. A thermostat selects between on and off against a target temperature and adapts to feedback, yet no one would call it an agent: its choice set is fixed and its transitions are enumerable. When every action a system might take is known and encoded in advance, it is a workflow, however much state and feedback it carries (see the boundary cases below). The agent property requires that the space of choices be open.

Goals are architectural objects because they determine:

- Termination criteria (what counts as "done").
- Tradeoff policy (latency vs correctness, cost vs thoroughness).
- Acceptable risk (what can be attempted autonomously).

Note that the definition requires a goal but does not require termination. An agent that loops indefinitely toward an unsatisfiable goal satisfies all four properties; nothing in the definition stops it. That gap is precisely what bounded autonomy (Chapter 5) and the governance layer (Chapter 6) close from outside, by enforcing iteration, cost, and time limits the agent cannot widen. Termination is therefore not a property of agents but a responsibility of the system around them.

Goals also delimit responsibility. A well-designed agent has a goal that can be audited and bounded. "Be helpful" is too unconstrained to support reliable operation without substantial governance.

This leads to an architectural position the book defends throughout: implicit goals are an anti-pattern for reliable systems. A goal carried only as a role, "you are a helpful coding assistant", is convenient in a demo and nearly impossible to govern, because a deterministic policy gate cannot evaluate whether an implied objective was satisfied, only whether an explicit one was. Promoting the goal to an explicit, measurable objective, stated with success criteria and constraints, is the first step in turning an agent from a plausible demo into a governable component. The definition admits implicit goals; the architecture discourages them.

In later chapters, goals interact strongly with bounded autonomy (Chapter 5). If you cannot state constraints (budget, allowed actions, unacceptable outcomes), you cannot bound autonomy in a meaningful way.

3. Capability to act upon an environment

An agent must be able to cause effects outside itself, invoking tools, mutating state in external systems, initiating workflows, making commitments, or otherwise changing the environment.

This property matters because it is the transition from *analysis* to *operations*. The moment an agent can act, side effects must be made observable and attributable, actions must be validated (schema, policy, authorization), and high-risk actions must route to escalation (Chapter 6). Environments can be physical (robots), digital (APIs, databases), or social (ticketing systems, communications); the book treats the environment abstractly, as the substrate in which actions have consequences and from which feedback is observed.

A system that generates text but cannot trigger actions is best treated as a decision-support component rather than an agent. It can still be valuable, but many of the failure modes of agentic systems (cascading side effects, irreversible actions) do not apply.

4. Adaptation based on outcomes

Adaptation means the agent incorporates feedback from the environment into subsequent reasoning and behavior. Feedback can be immediate (tool result, error code) or delayed (evaluation outcome, user correction), but it must change the agent's future action selection. Adaptation is not necessarily learning in the ML sense; the book does not require gradient updates or fine-tuning, and it can be as simple as revising a plan after a tool call fails or recording a constraint and proceeding differently.

The distinction that earns its place here is architectural, between two time horizons. *Intra-task adaptation* happens within a single run; *inter-task adaptation* happens across runs, when the agent updates a persistent store so a mistake made today is not repeated in next week's session. Both satisfy this property, but they rest on different memory architectures, working memory for the former, episodic or semantic memory for the latter (Chapter 7), and they carry different governance risk, because inter-task adaptation lets one session's outcome change another session's behavior. Adaptation also introduces the problems that motivate much of the rest of the book: feedback can be noisy, adversarial, or incomplete; the loop can oscillate without convergence; and the agent may overfit to short-term signals. Cognitive structures that improve reasoning under uncertainty (Chapter 4), governance constraints that prevent runaway adaptation (Chapter 6), and evaluation that detects drift across runs (Chapter 12) are the responses.

What is not an agent (boundary cases)

The definition earns its keep by excluding systems that look agentic but lack one of the four properties. Most boundary cases reduce to a single missing property and need only be named. A stateless service that computes a response from a request (no state, no goal-directed choice) is infrastructure, often the deterministic shell an agent runs inside. A scripted workflow or RPA process (state and action, but a fixed, enumerable choice set rather than open-ended selection) is a workflow, and the book recommends it over an agent wherever the domain is stable (Anthropic's *workflows* sit on this side of the line, and their reliability comes precisely from omitting Properties 2 and 4). A planner that never executes (no action, no feedback) is a planning subsystem. A reactive tool router that maps inputs to tool calls with no goal-directed reasoning and no cross-step state is a router. And several "agents" packed into one prompt are one agent stylized as several unless each maintains independent state, goals, and adaptation. None of these is a value judgment; a workflow or a router is often the better tool. They are category statements: the patterns in this book address systems that have all four properties, and applying them elsewhere adds ceremony without benefit.

The one confusion worth developing, because it traps the most teams, is the chat interface. A chat surface that generates text but cannot execute actions, or whose actions are always carried out by a separate deterministic process the agent never selects, is not an agent under this definition; it is a decision-support component, however useful. Conversely, a chat interface can *host* an agent when the loop behind it includes tool invocation, feedback incorporation, and persistent state that influences subsequent steps. The interface is then a presentation layer over an agent, not the agent itself. The trap is to ship the first, call it an agent, and then apply agentic patterns to a system that has no loop to bound; the discipline is to ask, of the thing behind the chat box, whether it closes the loop, and to treat the patterns that follow as applicable only when it does.

Agents, tools, and deterministic infrastructure

The common architectural mistake is to treat the agent as the whole system. In production systems the agent is almost always a bounded component inside the deterministic shell — the tool adapters, schemas, policy gates, human oversight, memory stores, and trace that Chapter 1 named and that the rest of the book develops. The decomposition that matters is the one this definition sets up: the agent is the decision-maker inside a loop, and the surrounding infrastructure defines the permitted action surface. The full design of that loop, the envelope around the model, is the *harness*, introduced in Chapter 4 and designed in Chapter 19; this chapter's job is only to fix what sits inside it.

The Skills layer (Chapter 10) extends this decomposition with a runtime mechanism for loading capability descriptions on demand. Skills do not change the agent boundary; they change *what the agent can be asked to do* without re-deploying the system. The architectural implication of Skills is treated in Chapter 10.

Bounded autonomy as a property of agents-in-systems

Bounded autonomy is not a moral stance; it is an engineering requirement. Unbounded loops and unconstrained action surfaces are not compatible with production reliability. The agent that satisfies all four properties is, by that fact, a component capable of doing real damage: it iterates, it spends, it acts on the world, and it carries state forward that can be wrong. Those four capacities are exactly

what the definition admits, and they are exactly why the definition is strict enough to be worth bounding.

Bounding autonomy means imposing explicit limits on iterations, cost, time, action surface, data access, and reversibility. Some of these bounds can be internal to the agent (stop conditions); most must be enforced externally by the governance layer (Chapter 6), because an agent cannot be relied upon to police itself under uncertainty or adversarial conditions. Chapter 5 develops bounded autonomy as a discipline in its own right; the point here is only that the definition makes bounding necessary rather than optional, by admitting only systems capable of the loop-shaped damage the bounds exist to contain.

Summary

An agent, in this book, is defined by a closed loop: persistent state, goal-directed choice among actions, the capability to act on an environment, and adaptation based on feedback. The definition is deliberately strict so that architectural decisions can be reasoned about without ambiguity.

With the term fixed, Chapter 3 introduces the architectural forces that agentic patterns must balance, and the remainder of the book presents the patterns and the discipline that turns those balances into reliable systems.

Chapter 3: Architectural forces in agentic design

Every design pattern in this book, and most of the patterns in canonical sources such as Gulli (2025), Anthropic's *Building Effective Agents*, and the CSIRO catalog, exists to balance a competing pair of structural forces. The patterns are not arbitrary, they are responses to tensions that arise whenever a probabilistic reasoning component sits on the control-flow path of a system that has to be reliable, auditable, and economical to run.

This chapter names those forces. The list is short by intent. A small number of well-named forces is more useful than a long catalog of dimensions, because architects reason about systems pair-wise: in any given design decision, two of these forces dominate and the rest fall out as consequences.

These seven are a working vocabulary, not an orthogonal basis, and it is worth saying so plainly. Some overlap. Exploration versus cost and deliberation versus latency are close to the same tradeoff in two currencies, money and time, kept separate here because which currency binds depends on whether the system is batch or interactive. Autonomy versus control and generality versus governability both pull between a system's reach and the ease of constraining it. The point of the list is the clarity it buys, not a clean decomposition: naming the dominant force pair for a decision surfaces the tradeoff being made, even when a third force is also in play.

The forces named here function the way *forces* function in Christopher Alexander's *A Pattern Language* and in Gamma et al.'s *Design Patterns*. They are not solutions; they are the structural tensions a pattern resolves. A pattern is a *deliberate balance* among forces. Reading later chapters with the forces in mind clarifies what each pattern is giving up to obtain what it provides.

The forces are:

1. Autonomy vs. control
2. Exploration vs. cost
3. Adaptability vs. predictability
4. Centralization vs. emergence
5. Deliberation vs. latency
6. Memory depth vs. context constraints
7. Generality vs. governability

Each is treated below. The chapter closes with a short note on how forces compose, and how to read the rest of the book as a sequence of deliberate balances among them.

Autonomy vs. control

The tension. Unconstrained ability to handle novel tasks versus the deterministic enforcement of safety boundaries.

The first and load-bearing force. Greater autonomy increases an agent's ability to handle novel tasks, recover from unexpected failures, and choose among alternatives without prescriptive instructions. It also increases the variance of outcomes, the cost of testing, and the surface for unsafe action.

Greater control, expressed through explicit constraints, validators, narrowed tool surfaces, and deterministic orchestration, reduces variance and makes the system tractable to test and audit. It also reduces the system's ability to handle situations the designer did not anticipate.

There is no universal balance. There is, however, a default for production systems: less autonomy than the team is tempted to grant, and bounded enforcement of the rest (Chapter 5). Systems frequently fail because autonomy was granted on the assumption that the model would behave like a careful junior engineer. Models, even strong ones, are better understood as components with unbounded curiosity, no fear of irreversible action, and no internal sense of cost.

The autonomy/control tradeoff is the single force that most strongly determines the *shape* of an agentic system. It is the reason this book devotes Chapter 5 to bounded autonomy as a discipline rather than a feature. A system designed around the "right" autonomy point with the wrong control mechanism is brittle in production; a system designed with conservative autonomy and strong control mechanisms can be loosened later as the model and operating discipline prove themselves.

A common mistake is to imagine the choice as a single dial running from "fully manual" to "fully autonomous." It is closer to a set of independent dials, one for each axis of bounded autonomy (Chapter 5). A system can have high autonomy on which sequence of tools to invoke while having low autonomy on which data the tools may touch; high autonomy on iteration count while having low autonomy on cost spend. Treating the dials as independent prevents the failure mode in which all dials are turned up together.

Exploration vs. cost

The tension. Better answers from more exploration versus the tokens, time, and money each additional pass consumes.

The reasoning component can almost always improve its output by exploring more, generating more candidates, reflecting more times, calling more tools, retrieving more context. Each increment of exploration carries an increment of cost in tokens, time, external API spend, and downstream resource use.

The architectural question is not "should we explore" but "what budget governs exploration, and where is the budget enforced?" The budget can be:

- **Implicit** in the reasoning loop's stop conditions (poor; the model decides when it is done).
- **Bound to a single dimension** like iteration count (acceptable for some patterns; misses cost spikes from expensive tools).
- **Multi-dimensional and externally enforced** — iterations, tokens, wall-clock, spend, retries, with hard limits that abort the loop (the form recommended in Chapter 5).

The tradeoff has a non-obvious shape: the marginal return on exploration is steep and then flat. Two iterations are often dramatically better than one; 10 are rarely much better than eight. Most exploration budgets in production systems are set too high because the architect tested with the model converging in three iterations on easy cases and did not measure the long tail. The percentile distribution of useful exploration matters far more than the average.

Self-consistency, debate, and tree-of-thought patterns (cataloged in Chapter 4) are explicit explorations of this force. Each commits resources to additional reasoning passes in exchange for variance reduction. The architectural question is when the variance reduction justifies the cost. For a $0.05 query asked thousands of times per day, a self-consistency multiplier of five may be unjustifiable; for a once-per-week strategic analysis worth thousands of dollars in human time, a multiplier of ten may be cheap.

The deeper architectural point is that the cost dimension to optimize depends on what the system is for. Interactive systems optimize wall-clock; batch systems optimize spend; high-stakes one-shot systems optimize quality and accept high cost. Naming this explicitly in the design, "this system optimizes X subject to constraints on Y and Z", prevents the failure where the team applies exploration patterns appropriate for one optimization profile to a system that should have been on another.

Adaptability vs. predictability

The tension. Adapting to novel situations at runtime versus behavior that is predictable, testable, and certifiable across runs.

An agentic system is valuable in part because it can adapt, replan after a failure, switch tools, change strategy. The same adaptability makes the system's behavior harder to predict from one run to the next, harder to test, and harder to certify against compliance requirements.

The architectural moves that mitigate the tradeoff are:

- **Replay testing** (Chapter 12) — record traces and assert on them; if the trace stays within the envelope of an approved replay, behavior is treated as acceptable.
- **Bounded plan revision** — allow the agent to replan, but only a bounded number of times and only along approved branches.
- **Deterministic substrate** — keep the deterministic infrastructure deterministic; the only variability is in the reasoning component, which is itself bounded.
- **Behavioral envelopes** — describe the agent's *space of acceptable behaviors* rather than its exact behavior, and assert at deployment that the envelope holds.

The deepest cost of unmanaged adaptability is *organizational*. Stakeholders cannot certify a system whose behavior they cannot describe. The architecture's job is not to eliminate adaptability, that would defeat the purpose, but to make the *envelope* of adaptability describable.

Regulated industries pay this cost most visibly: a financial-services agent that adapts unpredictably cannot be deployed at all in jurisdictions that require behavioral certification. The architectural answer is the envelope: the agent has wide latitude inside a tightly described boundary, and the boundary is what is certified. The certification is then stable across model versions, prompt changes, and small architectural revisions; only changes to the envelope require re-certification.

For non-regulated systems, the same principle applies for different reasons. A customer-facing system whose behavior is unpredictable from week to week loses user trust independently of whether

any individual run was correct. Users build mental models of systems and rely on those models. An envelope that holds across runs lets users build a working model; behavior without an envelope prevents them from doing so.

Centralization vs. emergence

The tension. Centralized orchestration that is easy to reason about and govern versus decentralized coordination that scales and partitions responsibility.

Single-agent systems with centralized orchestration are easier to reason about, debug, and govern. Multi-agent systems with decentralized coordination scale better, partition responsibility, and can exhibit useful emergent behavior.

The honest reading of the 2024–2026 literature is that multi-agent is over-prescribed. A surprising fraction of production agentic systems are single agents with tools, plus possibly an orchestrator that delegates to specialized sub-agents under bounded authority. True multi-agent systems with peer-to-peer coordination, voting, debate, or auction-based allocation remain rare in production and disproportionately common in demos.

The pragmatic default: start with a single agent and tools, then add an orchestrator with workers if responsibility decomposes cleanly, then, and only with strong justification, move to peer multi-agent shapes. The CSIRO catalog and Gulli's chapter on multi-agent collaboration cover the shapes thoroughly, the architectural question is *which shape does the problem actually require*, not *how do we build the most interesting one*.

The tradeoff also has an observability cost. Multi-agent traces are harder to read, debug, and replay than single-agent traces. Chapter 12 develops trace discipline; Chapter 9 covers coordination shapes in compressed form. An incident in a single-agent system that an on-call engineer can debug in 20 minutes can take hours in a poorly-instrumented multi-agent system, because the engineer must reason about cross-agent interactions, inter-agent message channels, and timing artifacts.

The architectural commitment in multi-agent systems is to make the coordination layer first-class. Inter-agent channels are governed (Chapter 6); messages are structured; the orchestrator's view of fleet state is observable; the trace links events across agents through correlation identifiers. Without this, multi-agent systems decay into hard-to-debug systems whose value depends on no two agents disagreeing in a way the architecture cannot resolve.

Single-agent systems are dull architectures. They are also the ones that ship reliably and survive contact with production, which is the test that matters, and the reason to resist a more interesting shape when a simpler one will do.

Deliberation vs. latency

The tension. Deeper reasoning for better answers versus the response-time budget the system must meet.

Deeper reasoning produces better answers and slower responses. For interactive systems (chat, IDE assistants, voice) latency is a hard constraint; for batch systems (research agents, code migrations, document processing) latency is a budget, not a cliff.

The architectural moves here are:

- **Two-tier reasoning** — a fast tier handles common cases; a deliberate tier handles flagged ones. Anthropic's evaluator-optimizer pattern is one shape; Andrew Ng's reflection pattern is another.
- **Asynchronous deliberation** — for batch work, decouple the user interaction from the reasoning, with structured progress reporting (Chapter 18).
- **Streaming** — for interactive cases, surface intermediate reasoning to the user while the loop continues. Streaming improves *perceived* responsiveness without reducing actual computation.
- **Progressive disclosure** — load context (and skills, Chapter 10) only when the task warrants the deeper reasoning, and keep the fast tier light.

It helps to separate two latencies. *Information latency*, time to first token, is what a user feels, and streaming addresses it directly. *Action latency*, time until the agent actually acts on the environment, such as calling a database or committing a change, is untouched by streaming: if the agent must reason for 15 seconds before invoking the tool, the side effect still lands 15 seconds late. For systems that chat with users, information latency dominates; for systems that act on the world, action latency is the real constraint, and streaming is no remedy for it.

Note that reasoning models in 2026 reduce the visible latency penalty for deeper reasoning, because more of the reasoning is internal to a single model call. This shifts some patterns (reflection, plan-execute) from architectural to model-internal, see Chapter 4. The architectural question, however, persists: the reasoning model's internal deliberation still consumes time and tokens, and a system that calls the reasoning model on every interaction pays the cost on every interaction.

A neglected aspect of the latency tradeoff is *latency variance*. Users tolerate consistent latency better than variable latency. A system whose responses come in two seconds 99% of the time and 12 seconds 1% of the time feels less reliable than one whose responses always take five. The architectural commitment is to monitor percentiles, not averages, and to bound the long tail at the time-budget axis (Chapter 5).

Memory depth vs. context constraints

The tension. Richer memory for competence versus the finite, attention-limited context window the model reasons over each turn.

Agents benefit from richer memory: past tasks, learned strategies, domain knowledge, user preferences. Richer memory introduces retrieval complexity, increases the risk of leaking irrelevant context into the loop, and consumes the bounded window available to the reasoning model on every turn.

The force breaks into two sub-tensions:

- **Within-loop memory.** The reasoning model has a finite context window (even at the multi-million-token sizes of 2026 models, finite). Filling it with retrieval results, prior turns, and tool documentation reduces the space available for reasoning. The Skills layer (Chapter 10) and Anthropic's progressive disclosure mechanism explicitly target this: load only what is needed, when it is needed.

- **Cross-loop memory.** Episodic and semantic memory (Chapter 7) accumulate across runs. They improve the agent's competence over time and degrade the predictability of the system, because behavior now depends on history that may not be obvious from the current input.

The pragmatic balance: prefer rich tool-mediated retrieval to large static context. Prefer compact, summarized memory to verbatim memory. Treat memory as a managed resource with policies for what is captured, when it is consulted, and how it is invalidated. Chapter 7 develops this in depth.

The dominant cost of a bloated context is not financial; it is attentional. Models attend non-uniformly across long contexts: relevant content placed late in a long window is processed less effectively than the same content in a short one. A 100-token context with the right content beats a 100,000-token context with the right content buried in the middle. Even a multi-million-token window that is cheap to fill degrades the model's reasoning about the immediate task when most of it is noise. Memory architecture is, first, a signal-to-noise problem.

The financial argument is weaker than it once was. Provider-side prompt caching, introduced across the major APIs in late 2024, lets a large static prefix, system instructions, tool documentation, stable semantic memory, be resent each turn at a fraction of its first-call cost. Caching does nothing for content that changes every turn, and nothing for the attention problem above, but it removes the simplest reason to keep static context small. Cost discipline now matters most for the dynamic, per-turn portion of the context and at high scale, where the ratio between disciplined and undisciplined memory architectures can still reach an order of magnitude.

Generality vs. governability

The tension. One agent that can do anything versus a narrowly scoped agent that is straightforward to bound and own.

A general agent, one capable of arbitrary tool use, arbitrary tasks, arbitrary domains, is impressive in demos and hard to govern. A narrowly scoped agent, one tool surface, one task class, one domain, is dull to demonstrate and straightforward to govern.

The force shows up as a structural choice in nearly every project:

- Specialist agents with narrow tool surfaces (one for code, one for SQL, one for incident response). Easier to bound. Often combined under an orchestrator.
- Generalist agents with broad tool surfaces and a reasoning loop that decides which tool to use. Harder to bound. Failure modes are harder to enumerate.

The book's recommendation, consistent throughout, is the specialist direction: bounded scope first, generality only when the operating profile justifies it. Most production agentic systems work because they are doing something narrow well, with strong governance, rather than something broad mediocrely with weak governance.

The decision has organizational implications too, and Conway's Law is the right lens for them: systems come to mirror the communication structures of the organizations that build them. A specialist agent has a clear owner, the team for that domain, so the architecture and the org chart reinforce each other. A generalist agent has unclear ownership; everyone is responsible, which usually means no one is, and the system decays because no single team's structure maps to it. Specialist orchestrator-worker architectures align with Conway's Law; monolithic generalist agents work against it, and tend to be worse-maintained over time as a result.

There is a counterargument: an organization may not have the budget for many specialist agents and may prefer one generalist. The architectural answer is to *factor* the agent's responsibilities so that the specialist split is internal to the system even if it is not visible to the user. An orchestrator-worker architecture (Chapter 9) presents a single front to the user while internally being a small team of specialists; this preserves the governance benefits of specialization without multiplying user-facing complexity.

How forces compose

In most design decisions, two of the seven forces dominate and the rest follow.

- *Bounding autonomy* (Chapter 5) is primarily about autonomy vs. control and exploration vs. cost.
- *Governance architecture* (Chapter 6) is primarily about adaptability vs. predictability and generality vs. governability.
- *Memory design* (Chapter 7) is primarily about memory depth vs. context constraints and adaptability vs. predictability.
- *Coordination* (Chapter 9) is primarily about centralization vs. emergence and deliberation vs. latency.
- *Skills* (Chapter 10) is primarily about memory depth vs. context constraints and generality vs. governability.
- *Failure modes* (Chapter 11) is the consequence of failing to balance any of these forces.

When reading later chapters, identify the pair of forces dominating the pattern. The pattern's tradeoffs section is, in essence, a statement of which force was accepted as the cost and which was preserved as the benefit. A reader who can map any pattern in the book (or in the canonical sources) to a force pair has the analytical vocabulary the book is trying to provide; a reader who cannot has either missed something or has identified a pattern that does not deserve its place.

It is also useful to recognize that *good* patterns are clear about their force balance, while *bad* patterns are vague. A pattern that claims to optimize everything simultaneously is hiding the tradeoff it makes; the tradeoff is real and will appear in production. A clear statement of which force the pattern privileges is part of what makes a pattern usable.

Forces, patterns, and deliberate choice

Several existing pattern catalogs (Gulli 2025, the Augment Code 2026 guide) list forces alongside patterns. Where they differ, this book follows the principle that forces describe tensions and patterns describe responses to them. A pattern is good when it makes the architect's choice explicit; it is bad when it disguises the choice as a default. A useful test for any proposed pattern, in this book, a canonical source, or a team's design document, is to ask which forces it balances and at what cost. If the answer is unclear, the pattern is incompletely specified; if the answer is "all of them," the pattern is incoherent, because the forces are not simultaneously optimizable.

The forces themselves are stable; the *positions* architects should take on them shift as the field evolves. In 2024 the right point on the autonomy/control axis was further toward control than many architects realized. In 2026, with stronger reasoning models, the same axis can be carried

slightly toward autonomy in domains where the bounding layer is mature. In 2028 it may shift again.

The architectural commitment is invariant across these shifts: whatever the current position, it is a deliberate choice, expressible as a balance of forces, defended by the design's bounding and governance, and revisable as the field evolves. A team that builds with this discipline can move along the axes as conditions change; a team that does not is locked into the position it implicitly chose at the start. Chapter 4 and onward use the forces named here as the vocabulary for that discussion.

Summary

Seven structural forces, autonomy/control, exploration/cost, adaptability/predictability, centralization/emergence, deliberation/latency, memory depth/context, and generality/governability, drive nearly every architectural decision in agentic systems. The patterns in this book and in canonical sources can be read as deliberate balances among these forces. The next chapter maps the cognitive layer briefly, with attention to which patterns have become model-internal in 2026, from Chapter 5 onward, each architectural chapter is explicitly framed by the forces it balances.

Chapter 4: Cognitive patterns: A reference map

Cognitive patterns structure the *internal reasoning* of a single agent, how it converts a goal and a current state into a next action. They are the most heavily documented part of the agentic literature, and they are also the part most rapidly being absorbed into the reasoning models themselves.

This chapter closes Part I with a deliberately compressed treatment: the cognitive layer earns less space because erosion has moved much of it into the models, and the book's budget goes to what remains architectural.

The canonical patterns, ReAct, Plan–Execute, Reflection, Self-Consistency, Debate, Tree-of-Thought, Constraint-Guided Reasoning, and Tool Use, are each documented at length in the sources cited in Chapter 21: Gulli (2025) covers them with hands-on code; Anthropic's *Building Effective Agents* essay treats the workflow shapes; Andrew Ng's *Agentic Design Patterns* (DeepLearning.AI, 2024) introduces reflection, tool use, planning, and multi-agent collaboration as foundational; the CSIRO catalog formalizes them in the academic literature.

How to read this chapter

This chapter gives the architectural reader the vocabulary in one place and, more importantly, a candid assessment of pattern erosion — which patterns have moved into the model and which remain architectural. The reference table below carries each pattern's intent, the architectural residue it leaves in 2026 (what the surrounding system must still provide), and where to read the full treatment — compressed to a fraction of the length a per-pattern entry would require. The Erosion column marks how far the reasoning core has moved into 2026 models: `partial` for patterns whose reasoning has moved substantially into the model but which retain an architectural footprint; a dash (–) for patterns that remain fully architectural, requiring external coordination, isolation, or enforcement the model cannot provide for itself.

Chain-of-Thought, reasoning step by step within the model's own context, is the one cognitive pattern that has eroded *totally*, which is why it gets no row of its own. It is the substrate the others build on — ReAct adds action to it, self-consistency votes across several runs of it — and it now happens natively inside any reasoning model, leaving no architectural residue to manage. Its total erosion is precisely why none of the patterns below are tagged total: each earns its residue by reaching outside the model's context, for a tool, a second opinion, a validator, a vote, where Chain-of-Thought never had to.

Two patterns readers often expect here are deliberately absent. *Routing*, dispatching an input to one of several handlers, is a deterministic workflow, not an agentic cognitive loop (Chapter 2); when the routing decision is a fixed classifier, there is no goal-directed choice to erode. *Retrieval-augmented generation (RAG)* is not a standalone reasoning pattern either: it is a memory and retrieval architecture (Chapter 7) on the read path, and where it constrains the model to grounded sources it is an instance of Constraint-Guided Reasoning. Both matter enormously; neither is a cognitive pattern in the sense this chapter maps.

Cognitive patterns: reference table

Pattern	Intent	Architectural residue in 2026	Erosion	Where to read more
ReAct (reason–act loop)	Interleave reasoning and action in bounded iterations so the agent adapts to feedback without committing to a full plan upfront	The loop is increasingly model-internal; what remains is the *bounding* — iteration limit, cost ceiling, tool schemas, observability of the trace	partial	Yao et al., *ReAct* (2022); Gulli, Tool Use; Anthropic, *Building Effective Agents*
Plan–execute	Separate planning from execution: produce a structured plan, then execute, replanning on failure	Useful when the plan is an auditable deliverable (a step-by-step approval, a migration runbook); commitments are plan persistence, plan diffing on replan, and a governance gate between plan and execution	partial	Gulli, Planning; Augment Code 2026 catalog
Reflection (generate, critique, revise)	Improve correctness by inserting bounded self-evaluation between generation and commit	Whether the critic is the *same* model or a *different* one (separation matters for catching consistency failures) and whether the critique is *bounded* (cap on critique iterations to prevent thrashing); worthwhile when the critic has information the generator does not	partial	Andrew Ng, *Reflection*; Anthropic, *Evaluator-Optimizer*; Gulli, Reflection
Self-consistency	Run multiple independent reasoning traces over the same problem and aggregate by voting or selection to reduce stochastic error	A cost/quality tradeoff: how many traces, what aggregation rule (majority vote, weighted vote, judge model), whether traces share state	—	Wang et al., *Self-Consistency* (2022); Gulli, Self-Consistency

Pattern	Intent	Architectural residue in 2026	Erosion	Where to read more
Debate	Structured adversarial exchange between two or more agents (or two roles of the same agent) before a final judgment	Role isolation (each side cannot see the other's prior turns in a way that compromises the dialectic), a judge component, bounded rounds; costly, reserved for problems that benefit from *opposing* perspectives	—	Du et al., *Multi-Agent Debate* (2023); Gulli, Debate
Tree-of-thought	Explicit branching reasoning: generate candidate next-steps, score, prune, recurse, for problems with combinatorial structure	Heavy. Branch factor, depth limit, scoring function, pruning policy. Rarely justified in enterprise tasks, which lack the objective intermediate-state evaluator ToT needs; cost grows as the product of branch factor, depth, and scorer cost	—	Yao et al., *Tree of Thoughts* (2023); Gulli, Reasoning Techniques
Constraint-guided reasoning	Apply explicit domain or structural constraints during reasoning to reduce hallucination and ensure output conformance	Substantial and underappreciated. Structured-output constraints (JSON schema, regex, grammar-constrained decoding), retrieval-grounded prompts, and pre/post-validators (Chapter 6) are all instances. The *most* architectural of the cognitive patterns: the constraint typically lives outside the agent and is enforced by deterministic infrastructure	—	Anthropic, *Building Effective Agents* (validators); Gulli, Safety Patterns

Pattern	Intent	Architectural residue in 2026	Erosion	Where to read more
Tool use	Recognize a gap in the model's own knowledge or reach, formulate a query to close it, and suspend reasoning until the observation returns	Where the cognitive layer meets the architecture. The model decides *which* tool; the architecture decides *which tools are available*, *with what authorization*, *with what schema enforcement*, *with what side-effect logging*. The footprint has *grown*, not shrunk: the tool surface is where most production failures happen (Chapter 11)	—	Andrew Ng, *Tool Use*; Anthropic, *Tool Use* docs; Gulli, Tool Use

The pattern is clear: cognitive structures the model can run *inside its own context* have eroded; structures that require *external coordination, isolation, or enforcement* have not. The pattern would be falsified only if provider-hosted execution became the norm *and* customers lost the ability to enforce deterministic bounds and refusals on every proposed action. They have not: the execution seam remained customer-controlled even when tools ran on provider infrastructure. Chapter 19 treats the seam where that control is most at risk today. This is good news for architects. The patterns that erode are the ones with the most public attention but the smallest production payoff. The patterns that remain are the ones that decide whether a system holds up in production — bounding, governance, observability, the tool surface. The rest of the book is about those.

The harness: the envelope that did not migrate

The reference table just presented tells a story of migration: the cognitive loop has moved into the model, and what remains outside is the *envelope* around it. That envelope is the **harness**, the deterministic program that turns a model into an agent, and it is the structure the rest of this book attaches to. This section introduces the concept; Chapter 19 designs it in full.

The envelope, not the inner loop

What migrated into the model is the *inner cognitive loop*; what did not migrate is the *envelope* around it — assembling context, executing proposed actions, enforcing bounds, calling governance, persisting state, and recording the trace. That envelope is the **harness**, the deterministic infrastructure that turns a probabilistic model into a bounded, governable agent.

A single harness turn may wrap several model-internal steps: the harness makes one model call and sees only the boundary — what went in, what actions came out — deliberately blind to the deliberation in between. As the inner loop moves into the model, the envelope carries more, because more behavior is decided inside an opaque call only the envelope can constrain.

The model decides; the harness acts

The model is the reasoning organ; the harness owns every hand and eye. The model emits text proposing action; deterministic harness code executes tools with authorization, schema enforcement, and trace logging. Chapter 19 develops the execution seam — including provider-hosted execution of effectful tools, where the principle is most at risk.

What attaches to it

The remaining architectural chapters each fill a slot in the envelope:

- **Bounded autonomy** (Chapter 5) is the set of limits — iteration, cost, deadline, risk — the harness checks before and after each turn.
- **Governance** (Chapter 6) is the pipeline the harness routes every proposed action through before it touches anything.
- **Memory** (Chapter 7) is the state the harness assembles into context each turn; the **ingestion pipeline** (Chapter 8) is how semantic memory is written.
- **Coordination** (Chapter 9) is what the harness does when the work needs more than one envelope.
- **Skills** (Chapter 10) are procedural know-how the harness loads into context at runtime.
- **The trace** (Chapter 12) is the record the harness writes of every model call and every action result.

This map is the payoff that justifies the section. The forward-references in those chapters now point at a defined structure: they describe the slots they fill in the envelope the reader has now seen.

A preview, not a design

This section has introduced the *concept* of the harness; it has not *designed* it. The design, the question of where a new capability belongs when one is needed, in harness code, in a tool, in a skill, or in a new agent, is the subject of Chapter 19. That chapter designs the loop in full and inverts the book's own thesis: the harness is the system, and the model is a bounded component within it. The harness is treated as a given in *design* terms through Parts II and III — each chapter develops a layer the harness enforces at each turn — and Part IV composes those layers into whole systems and operational practice before Chapter 19 designs the loop. This is the introduce-then-synthesize structure that gives the capstone its power.

A reader who wants the practical answer sooner, before the capstone, can jump ahead to the decision sequence in Chapter 19: it resolves a new capability by asking the cheapest distinguishing questions first, whether it is knowledge or capability, whether it needs a new effector or sensor the agent lacks, whether it is know-how over existing tools, or whether it is internal logic that must be guaranteed. The constraint that closes it, *know-how may move into skills freely; power may not*, is the rule that keeps the skills movement from becoming a governance bypass, and it is worth carrying through the intervening chapters, where the action surface (Chapter 5) and the skills layer (Chapter 10) are the two places the question arises most naturally.

Where the book goes from here

The cognitive patterns themselves are cataloged elsewhere, read Gulli, Anthropic, and Andrew Ng for full treatments, and used here by name without further re-derivation. For academic survey

treatments of the surrounding literature — agent methodology, planning, and tool learning — see Chapter 21. The book's budget goes to the structure that did not migrate: the envelope and its slots, beginning with bounded autonomy in Chapter 5.

About this sample

This is a free sample of *Architecting Agentic Systems: Engineering Dependable AI Agents*, by Damian Beresford. It contains the **Preface** and all of **Part I, Foundations** (Chapters 1-4):

- Chapter 1 — The nature of agentic systems
- Chapter 2 — The agent: formal definition and properties
- Chapter 3 — Architectural forces in agentic design
- Chapter 4 — Cognitive patterns: a reference map

Chapter 4 closes Part I by introducing the **harness** — the deterministic envelope that turns a model into an agent. The full book designs the harness in Chapter 19, after developing every layer that attaches to it: bounded autonomy, governance, memory, ingestion, control and coordination, and the skills layer (Part II); failure modes, trace discipline, the glass layer, enterprise integration, and model routing (Part III); and system architectures, the Concord worked example, and operationalization (Part IV).

The full book is available at **architecting-agentic-systems.net**.