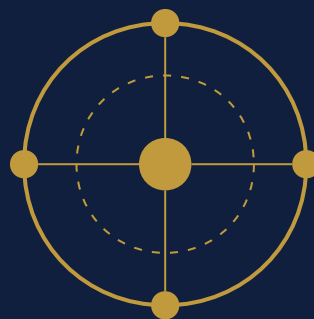

LANDRY INDUSTRIES · TECHNICAL MONOGRAPH

Engineering Autonomous Intelligence

*The Architecture, Tradecraft, and Deployment of
Agentic AI Systems*



MARIE LANDRY

Moncton, New Brunswick · 2026

Engineering Autonomous Intelligence

The Architecture, Tradecraft, and Deployment of Agentic AI Systems

Marie Landry

CEO & Founder, Landry Industries Inc. · ORCID 0009-0008-5027-3337

First edition, October 2026. Moncton, New Brunswick, Canada.

© 2026 Marie Landry / Landry Industries Inc. All rights reserved.

Note on sources. Every work cited in this book was checked against a live source (publisher, arXiv, proceedings, standards body, or vendor documentation) on 6 October 2026 before inclusion. Quantitative claims attributed to a source were confirmed against that source. Where a figure could not be confirmed, it was omitted. Figures in §10.3 marked as illustrative are assumptions for demonstrating a method, not measurements. Product names and protocols evolve quickly; readers should consult the cited sources for current details.

Preparation. Drafted with AI assistance (Claude, Anthropic) from the author's outline; set in Lora and Poppins.

Trademarks belong to their respective owners. Nothing in this book is legal, financial, or security-certification advice.

Contents

- PART I • FOUNDATIONS AND COGNITIVE ARCHITECTURE 5
 - 1 The Paradigm Shift from Chatbots to Autonomy 6
 - 1.1 The Death of Static Prompting 6
 - 1.2 Defining Agentic AI 8
 - 1.3 Taxonomy of Agents 9
 - 1.4 The Agentic Stack 11
 - 2 The Core Cognitive Loop: ReAct, Plan-and-Solve, and Beyond 13
 - 2.1 The ReAct Framework 13
 - 2.2 Advanced Reasoning Paradigms 15
 - 2.3 Context Management and Attention Budgets 17
 - 2.4 Handling Non-Deterministic Outputs 19
 - 3 Memory Systems for Persistent Intelligence 21
 - 3.1 The Three-Tier Memory Architecture 21
 - 3.2 Semantic Compression and Retrieval 24
 - 3.3 Procedural Memory 26
- PART II • ENGINEERING AND TOOL INTEGRATION 28
 - 4 The Agent-Computer Interface (ACI) and Tool Use 29
 - 4.1 Designing for Machines 29
 - 4.2 Function Calling Mechanics 31
 - 4.3 Error Handling and Poka-Yoke Tool Design 33
 - 5 Advanced Retrieval-Augmented Generation and Agentic Search 37
 - 5.1 From Static RAG to Agentic Chunking 37
 - 5.2 Corrective and Self-RAG 39
 - 5.3 Multi-Source Synthesis 41
- PART III • MULTI-AGENT ECOSYSTEMS AND ORCHESTRATION 44
 - 6 Multi-Agent Topologies and Communication Protocols 45
 - 6.1 Architectural Patterns 45
 - 6.2 Communication Standards 47
 - 6.3 Mitigating the Complexity Tax 49
 - 7 Conflict Resolution, Deadlocks, and Consensus 52
 - 7.1 Distributed Agent State Management 52
 - 7.2 Consensus Mechanisms 54

7.3 Fault Tolerance	56
PART IV · PRODUCTION OPERATIONS, GOVERNANCE, AND SECURITY	58
8 AgentOps: Testing, Tracing, and Observability	59
8.1 The Agent Development Lifecycle (ADLC)	59
8.2 Tracing Execution Paths	61
8.3 Automated Evaluation	63
9 Security, Guardrails, and Threat Modeling	66
9.1 Prompt Injection and Indirect Exploits	67
9.2 Blast Radius Mitigation.....	69
9.3 Human-in-the-Loop (HITL) Gateways.....	71
PART V · ENTERPRISE SCALING AND FUTURE HORIZONS	73
10 Enterprise Integration and Operational ROI	74
10.1 Mapping Agentic Workflows to Business Processes	74
10.2 Managing Agent Sprawl.....	76
10.3 Economic Modeling.....	78
REFERENCES	81

PART I

Foundations and Cognitive Architecture

Before an agent can be engineered, it has to be understood. Part I defines what changes when a language model stops answering and starts acting, dissects the reasoning loop that drives that behaviour, and builds the memory systems that let an agent persist beyond a single exchange.

The Paradigm Shift from Chatbots to Autonomy

A chatbot is judged by what it says. An agent is judged by what it changes.

For most of the short history of large language models, the dominant interaction pattern was a single round trip: a human wrote a prompt, the model returned text, and the exchange ended. Everything interesting happened in the human's head before and after the call. The shift this book is about is the relocation of that “before and after” into software. An agentic system receives a goal rather than a question, decides for itself which steps to take, executes those steps against real tools, observes the results, and keeps going until the goal is met or it determines that it cannot be. This chapter establishes the vocabulary, the lineage, and the layered architecture that the rest of the book builds upon.

1.1 The Death of Static Prompting

Static prompting treats the model as a pure function: a string goes in, a string comes out, and the caller decides what to do next. This model of use produced remarkable results, but it has three structural ceilings that no amount of prompt craft can lift.

The first ceiling is **information**. A static call can only reason over what is in its context at the moment of invocation. If the answer requires a database lookup, a file that was not attached, or a fact that changed yesterday, the model must either guess or decline. The second ceiling is **verification**. A single generation has no opportunity to check its own work against the world; it cannot run the code it wrote, query the endpoint it described, or confirm the citation it produced. The third ceiling is **horizon**. Real tasks — reconciling an account, migrating a codebase, compiling an investigative dossier — consist of dozens or hundreds of de-

pendent steps, each of whose inputs depend on the outputs of the last. No single prompt can encode a plan whose later branches depend on facts not yet discovered.

The research that broke through these ceilings did so by putting the model inside a loop. Chain-of-thought prompting showed that asking a model to articulate intermediate reasoning steps materially improved performance on multi-step problems [1], and self-consistency showed that sampling several reasoning paths and taking the majority answer improved it further [2]. These were still static techniques, but they established that the model's own intermediate text could be treated as a working medium. ReAct then made the decisive move: it interleaved reasoning traces with *actions* that called external tools, feeding each tool's output back into the context as an observation before the next reasoning step [3]. The model was no longer generating an answer; it was generating the next move in a process.

Once a model is in a loop, the engineering problem changes character. The relevant questions are no longer “what is the best prompt?” but “what state does the loop carry?”, “what actions are available and how are they described?”, “how does the loop know when to stop?”, and “what happens when an action fails?”. Those are systems questions, and they are the subject of this book.

TRADECRAFT NOTE — REACTIVE VERSUS PROACTIVE

A useful field test: if you remove the human from between two model calls, does the system still make progress? If yes, you are building an agent, and every design decision should be evaluated as if no one will be watching that gap. If no, you are building an assisted workflow, and a different (usually simpler) set of controls applies.

Workflows versus agents

Not every loop is an agent. Anthropic's engineering guidance draws a line that is worth adopting as a working definition: *workflows* orchestrate models and tools through predefined code paths, whereas *agents* let the model dynamically direct its own process and tool use [4]. The same guidance catalogues five recurring workflow patterns — prompt chaining, routing, parallelisation, orchestrator-

workers, and evaluator–optimiser — and argues that teams should reach for the simplest pattern that solves the problem, adding autonomy only where it demonstrably pays for itself [4].

This distinction matters because autonomy is a cost as well as a capability. Each degree of freedom granted to the model is a degree of freedom that must be tested, monitored, and secured. A fixed three-step chain that extracts, classifies, and formats is easy to reason about; a model that decides at run time whether to search, which source to trust, and when it is finished is not. Throughout this book, “agentic system” is used as the umbrella term, and the reader is encouraged to treat the workflow end of the spectrum as the default and the fully autonomous end as something that must be earned.

1.2 Defining Agentic AI

The term “agent” long predates large language models. In the classical formulation, an agent is anything that perceives its environment through sensors and acts upon that environment through actuators, and a *rational* agent selects actions expected to maximise its performance measure given its percept history and built-in knowledge [5]. Wooldridge and Jennings, surveying the field in 1995, proposed a “weak notion” of agency built on four properties: autonomy, social ability, reactivity, and pro-activeness [6]. Both framings remain serviceable. What has changed is the component that does the deciding: where classical agents used hand-built rule sets, planners, or learned policies trained for a single domain, the modern agent delegates its deliberation to a general-purpose language model.

For engineering purposes, five pillars capture what an LLM-driven agent must have. Remove any one, and the system degrades into something less.

Pillar	Function	Typical implementation	Failure if absent
Autonomy	Chooses its next step without per-step human direction	Control loop with model-selected actions and a termination condition	Becomes a chatbot that waits to be told what to do
Perception	Takes in the state of its environment	Tool results, file reads, API responses, screenshots, retrieved documents	Acts blind; hallucinates the state of the world
Reasoning	Turns goals and observations into plans and decisions	The foundation model, scaffolded by reasoning patterns (Chapter 2)	Thrashes; takes actions with no coherent intent
Memory	Carries state across steps and sessions	Context window, episodic logs, vector stores, skill libraries (Chapter 3)	Repeats work; forgets constraints; cannot learn
Action	Changes the environment	Function calls against APIs, code execution, browser or desktop control (Chapter 4)	Can describe what should happen but never makes it happen

The Cognitive Architectures for Language Agents (CoALA) framework offers a more formal decomposition that maps neatly onto these pillars. It describes a language agent in terms of modular memory components (working and long-term memory, the latter subdivided into episodic, semantic, and procedural), a structured action space split into internal actions (reasoning, retrieval, learning) and external actions (grounding in the environment), and a generalised decision-making procedure that cycles between planning and execution [7]. CoALA’s value is that it places LLM agents within the long tradition of cognitive architectures, and it gives practitioners a shared language for comparing systems that otherwise look superficially different.

1.3 Taxonomy of Agents

The classical taxonomy, as presented in the standard AI textbook, arranges agent programs in order of increasing sophistication [5]. It is worth revisiting, because each level corresponds to a design pattern that still appears inside modern LLM systems.

1. **Simple reflex agents** map the current percept directly to an action through condition–action rules. In LLM systems, a router that sends any message containing an invoice number to the billing queue is a reflex agent, and there is nothing wrong with that.

- 2. **Model-based reflex agents** maintain an internal state that tracks aspects of the world not visible in the current percept. A support bot that remembers which product the customer mentioned three turns ago is model-based.
- 3. **Goal-based agents** consider the future consequences of actions relative to an explicit goal and can therefore plan. A coding agent asked to “make the failing test pass” is goal-based: many action sequences are possible, and the agent must search among them.
- 4. **Utility-based agents** weigh competing goals and uncertain outcomes using a utility function. Cost-aware agents that trade answer quality against token spend, or that decide whether a marginal search is worth its latency, are approximating utility-based behaviour.
- 5. **Learning agents** improve their own performance over time by critiquing their results and modifying their behaviour. Reflexion, which stores verbal self-critiques in memory and uses them on subsequent attempts, is a direct LLM instantiation of this pattern [8].

Modern LLM-driven agents collapse several of these levels into a single cognitive loop. The same model call may perform reflex routing, update an internal state summary, plan toward a goal, weigh costs, and reflect on a prior failure. Surveys of the field organise this landscape around a profile module (who the agent is), a memory module, a planning module, and an action module [9]. The practical lesson is not that the old taxonomy is obsolete but that it now describes *capabilities a single loop can exhibit* rather than distinct kinds of system — which means designers must decide deliberately which of those capabilities they are actually relying on, and test each one.

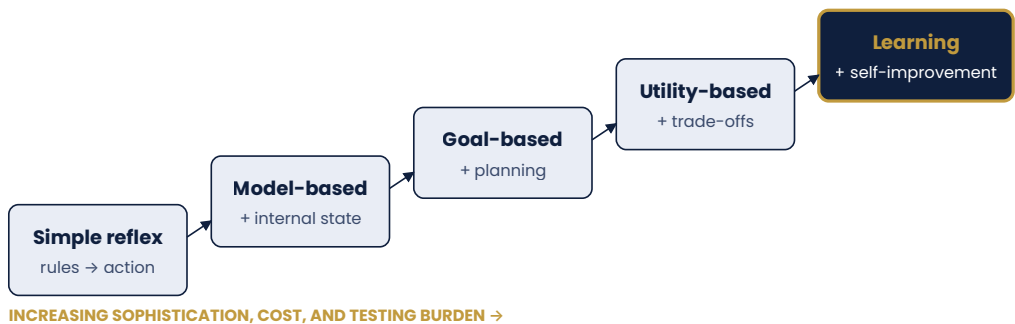


Figure 1.1 — The classical agent hierarchy (after Russell & Norvig). A single LLM loop can exhibit every level; the designer’s job is to know which levels the system actually depends on.

1.4 The Agentic Stack

An agent is not a model. It is a system in which a model is one component — the most visible one, but rarely the one that determines whether the system works in production. It helps to think of the agent as a stack of six layers, each with its own concerns, failure modes, and owners.

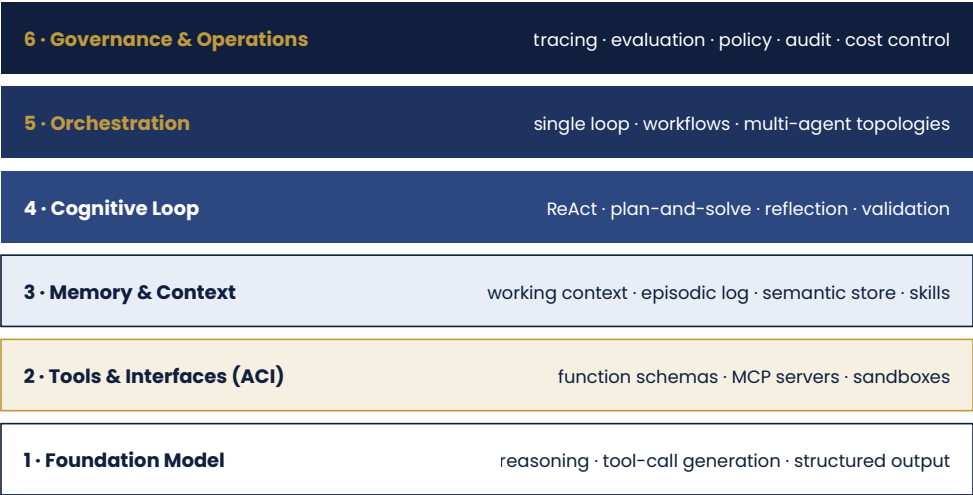


Figure 1.2 — The agentic stack. Most production failures originate in layers 2, 3, and 6, not in the model.

Layer 1 — Foundation model. The model supplies general reasoning, language understanding, and the ability to emit structured tool calls. Model choice sets a ceiling on what the system can do, but it is the layer the engineering team controls least. The relevant engineering decisions are selection (which model for which role), configuration (temperature, output limits, structured-output modes), and substitution (the ability to swap models without rewriting the system).

Layer 2 — Tools and interfaces. Tools are how an agent perceives and acts. The quality of tool design — names, descriptions, parameter schemas, error messages, output size — has a disproportionate effect on agent performance. Work on software-engineering agents found that a purpose-built *agent-computer interface* substantially outperformed giving the same model a raw shell [10]. Chapter 4 is devoted to this layer.

Layer 3 — Memory and context. The context window is the agent's working memory and its scarcest resource. Above it sit external stores: logs of past episodes, vector indices of documents and facts, and libraries of reusable procedures. Deciding what enters the context, when, and in what compressed form is now commonly called *context engineering* [11]. Chapters 2 and 3 cover this layer.

Layer 4 — Cognitive loop. This is the control flow that sequences thought, action, and observation, and decides when to stop. It is where reasoning patterns such as ReAct, ReWOO, and Tree-of-Thoughts live, and where validation gates enforce discipline on non-deterministic outputs (Chapter 2).

Layer 5 — Orchestration. Above a single loop sits the question of how multiple loops cooperate: as a fixed pipeline, as an orchestrator delegating to workers, as peers on a shared blackboard, or as nodes in a task graph. Protocols such as the Model Context Protocol and Agent2Agent standardise the connections (Chapters 6 and 7).

Layer 6 — Governance and operations. Finally, the system must be observable, testable, secure, and accountable. Tracing, evaluation suites, permission boundaries, human approval gates, and cost budgets live here (Chapters 8 through 10). This layer is the one most often deferred and the one whose absence most often ends a project.

CHAPTER SUMMARY

- Static prompting is limited by information, verification, and horizon; putting the model in a perceive–reason–act loop removes those limits and introduces systems-engineering problems in their place.
- Distinguish *workflows* (predefined paths) from *agents* (model-directed paths) and prefer the simplest design that works.
- Five pillars — autonomy, perception, reasoning, memory, action — define an agent; each maps to a layer of engineering effort.
- The classical agent taxonomy survives as a checklist of capabilities that a single LLM loop may exhibit.
- Think in a six-layer stack. The model is layer 1 of 6.

The Core Cognitive Loop: ReAct, Plan-and-Solve, and Beyond

Every agent, however elaborate, is a loop with a stopping rule. Most agent failures are failures of one or the other.

At the heart of every agent is a control loop that alternates between deciding and doing. The loop's design determines how many model calls a task costs, how well the agent recovers from surprises, how much of its context window it burns, and whether it ever stops. This chapter dissects the canonical ReAct loop, surveys the alternative reasoning structures that trade flexibility for efficiency or breadth, treats the context window as a budget to be managed, and closes with the validation machinery that keeps a probabilistic reasoner on a deterministic track.

2.1 The ReAct Framework

ReAct — *Reasoning and Acting* — prompts a language model to produce an interleaved sequence of three kinds of step [3]:

- **Thought:** free-text reasoning about the current situation, what is known, and what to do next.
- **Action:** a call to an external tool, expressed in a constrained format (originally plain text such as `Search[query]`; today usually a structured function call).
- **Observation:** the result returned by the tool, appended verbatim to the context by the harness — not generated by the model.

The loop repeats until the model emits a terminal action (such as `Finish[answer]`) or the harness halts it. The original authors evaluated the pattern on knowledge-intensive question answering and fact verification, where retrieving from a Wikipedia API reduced the hallucination and error propagation seen in reasoning-only

chain-of-thought, and on interactive decision-making environments, where it outperformed imitation and reinforcement-learning baselines while using only one or two in-context examples [3].

Two properties make ReAct the default starting point for most agents. First, it is *adaptive*: because each thought is conditioned on the latest observation, the agent can change course when a search returns nothing useful or a command fails. Second, it is *interpretable*: the thought trace provides a human-readable account of why each action was taken, which becomes the backbone of debugging and audit (Chapter 8).

```
# A minimal, production-shaped ReAct harness (pseudocode)
def run_agent(goal, tools, model, max_steps=25, budget_tokens=200_000):
    messages = [system_prompt(tools), user(goal)]
    for step in range(max_steps):
        reply = model.generate(messages, tools=tools.schemas())
        messages.append(reply)
        if reply.is_final():
            return validate_final(reply.content)          # §2.4
        for call in reply.tool_calls:
            result = tools.execute_safely(call, timeout=30) # Ch. 4, Ch. 9
            messages.append(tool_result(call.id, truncate(result)))
        if token_count(messages) > 0.8 * budget_tokens:
            messages = compact(messages)                 # §2.3
    return escalate("step budget exhausted", messages)   # never fail silently
```

Note what the harness, not the model, is responsible for: enforcing a step ceiling, executing tools with timeouts, truncating oversized results, compacting context, validating the final answer, and escalating when the budget runs out. The model chooses; the harness constrains. Getting that division of labour right is the single most important structural decision in agent engineering.

Failure modes of the plain loop

ReAct’s flexibility has costs. Because the model replans after every observation, it pays for a full model call per step, and the growing transcript is re-read each time. Long loops accumulate irrelevant observations that dilute attention. The model can fall into *action loops* – repeating the same failing search with trivially different wording – or *premature termination*, declaring success on the strength of a plausible but unverified observation. Each of the alternative paradigms below addresses one or more of these weaknesses.

2.2 Advanced Reasoning Paradigms

Plan-and-Solve

Plan-and-Solve prompting asks the model to first devise a plan that divides the task into subtasks and then carry out the subtasks according to that plan. Its authors introduced it to address missing-step errors in zero-shot chain-of-thought, and an extended variant adds more detailed instructions to reduce calculation errors as well [12]. In agent architectures, the idea generalises into the *planner-executor* split: one call produces an explicit plan as structured data; subsequent calls (or deterministic code) execute each step; a replanning call is triggered only when execution deviates from expectations. Making the plan an inspectable artifact has a large operational benefit: it can be validated, logged, shown to a human for approval, and compared against what actually happened.

ReWOO: reasoning without observation

ReWOO takes the planner-executor split to its logical conclusion. A *Planner* writes the full chain of tool calls up front, using placeholders (`#E1`, `#E2`...) to refer to the not-yet-known outputs of earlier steps; a *Worker* executes those calls and fills in the evidence; and a *Solver* composes the final answer from the plan and the collected evidence [13]. Because the model is not re-prompted with the full transcript after every observation, the paradigm avoids much of the redundant token consumption of interleaved loops; the authors report a fivefold token-efficiency gain alongside a four-point accuracy improvement on HotpotQA, and robustness when tools fail [13].

The trade-off is adaptivity. ReWOO is ideal when the shape of the investigation is predictable — gather these four facts, then synthesise — and poor when what to do at step three genuinely depends on what was found at step two. A pragmatic hybrid used in many production systems is to plan ReWOO-style, execute in parallel, and fall back to a ReAct loop only for steps whose results fail validation.

Tree of Thoughts

Chain-of-thought and ReAct both commit to a single trajectory: each step is chosen once and never revisited. Tree of Thoughts (ToT) generalises this by having the model propose several candidate “thoughts” at each step, evaluate them (again using the model, as a heuristic), and explore the resulting tree with breadth-first or depth-first search, including backtracking [14]. On tasks that re-

quire lookahead, such as the Game of 24, the gains were dramatic: the paper reports that GPT-4 with chain-of-thought solved 4% of problems while the same model with ToT solved 74% [14].

ToT is expensive — every node is at least one model call, and evaluation calls multiply the cost — so it belongs in a targeted role: inside a single hard step of a larger workflow (choosing an architecture, finding a proof, resolving a planning conflict), not as the outer loop of a general agent.

Graph of Thoughts

Graph of Thoughts (GoT) relaxes the tree into an arbitrary directed graph, so that thoughts can be *aggregated* (several partial solutions merged into one), *refined* in feedback loops, and reused across branches [15]. This maps naturally to decomposable problems — sorting, set operations, merging documents — where independent sub-results must be combined. Its authors report a 62% improvement in sorting quality over ToT while cutting cost by more than 31% [15]. For agent engineers, GoT is less a prompting trick than a confirmation of a broader principle: *the topology of reasoning is a design parameter*, and the same graph-structured thinking reappears at the multi-agent level in Chapter 6.

Paradigm	Structure	Adaptivity	Relative cost	Best fit
ReAct	Linear, interleaved	High	Medium-high (call per step)	Open-ended tasks with unpredictable environments
Plan-and-Solve / planner-executor	Plan then linear execution	Medium (replan on deviation)	Low-medium	Tasks with knowable structure needing audit
ReWOO	Full plan with evidence placeholders	Low	Low	Predictable multi-fact gathering; parallel tools
Tree of Thoughts	Tree search with backtracking	High within a step	High	Single hard decisions requiring lookahead
Graph of Thoughts	DAG with aggregation and refinement	High	Variable	Decomposable problems with mergeable parts
Reflexion	Episodes with self-critique memory	Across attempts	Multiplied by retries	Tasks with a clear success signal (tests, checks)

Reflection as a loop around the loop

A final paradigm deserves mention because it wraps any of the above. Reflexion has the agent attempt a task, receive a success or failure signal, write a natural-language reflection on what went wrong, store that reflection in an episodic buffer, and try again with the reflection in context [8]. No weights are updated; learning happens through memory. The pattern is powerful wherever a reliable external signal exists — unit tests, schema validators, compilers — and dangerous where it does not, because a model grading itself without ground truth can confidently reinforce its own errors.

2.3 Context Management and Attention Budgets

The context window is the agent's working memory, and it is both finite and imperfect. Two empirical findings shape how it should be managed.

The first is **position sensitivity**. In controlled multi-document question answering and key-value retrieval experiments, model performance was often highest when the relevant information appeared at the beginning or end of the input and degraded significantly when it sat in the middle — even for models marketed as long-context [16].

The second is **context rot**. A 2025 technical report from Chroma evaluated 18 models and found that performance becomes progressively less reliable as input length grows, even on deliberately simple tasks; that semantically similar distractors compound the decline; and, counter-intuitively, that models did better on shuffled haystacks than on logically coherent ones [17]. The practical implication is that a large advertised context window is a capacity limit, not a quality guarantee. Every token admitted to the context draws down what Anthropic's applied AI team calls an *attention budget*: because transformer attention relates every token to every other, more context means each piece receives proportionally less effective attention [11].

DESIGN PRINCIPLE — THE SMALLEST SUFFICIENT CONTEXT

Treat context as a scarce, depreciating asset. The goal is not to give the model everything that might be relevant; it is to give it the smallest set of high-signal tokens that makes the correct next action likely.

Techniques

Sliding windows and pinned regions. The simplest strategy keeps the system prompt, the goal, and a small set of “pinned” facts permanently at the top of the context, and retains only the most recent k turns of the transcript. It is cheap and predictable but loses mid-episode information abruptly.

Compaction. When the transcript approaches a threshold, a model call summarises it into a compact state description — decisions made, facts established, open questions, files touched — and the loop continues from that summary. Compaction preserves continuity across very long tasks at the cost of some fidelity; the key engineering decision is *what the summary must never drop* (constraints, user instructions, irreversible actions taken), which should be specified explicitly rather than left to the summariser [11].

Tool-result clearing and truncation. The raw output of an early tool call is rarely needed verbatim later. Replacing old tool results with a one-line digest (“listed 412 files; relevant: src/auth/*”) is one of the highest-yield, lowest-risk optimisations available.

Just-in-time (JIT) context injection. Rather than pre-loading documents, the agent holds lightweight references — file paths, record identifiers, URLs, saved queries — and fetches content only at the moment it is needed, exactly as a human analyst keeps a filing system rather than memorising it [11]. JIT retrieval converts context from a pre-paid cost into a pay-as-you-go one, and it naturally keeps relevant material near the end of the context, where position-sensitivity research suggests it is best attended.

Structured note-taking. For multi-hour tasks the agent maintains an external scratch file (a progress log, a to-do list, a findings register) that it writes to and re-reads, so that critical state survives compaction [11].

Sub-agent isolation. A sub-agent can explore extensively in its own context and return only a condensed result to its parent, keeping the parent’s context clean. This is a context-management technique before it is a multi-agent technique, and it is revisited in Chapter 6.

2.4 Handling Non-Deterministic Outputs

A language model is a sampler. Even at low temperature, the same input can produce different outputs, and a correct-looking output can be wrong. Agents amplify this, because one bad intermediate step contaminates every step that follows. Cognitive discipline therefore has to be imposed from outside the model, by validation loops built into the harness.

The validation ladder

1. **Syntactic validation.** Every structured output — tool arguments, plans, final answers — is parsed against a schema (JSON Schema, Pydantic, or equivalent). Malformed output is rejected with a precise error message and the model is asked to retry. Where the platform supports constrained or structured-output decoding, use it; it eliminates an entire class of failures.
2. **Semantic validation.** Parsed values are checked against business rules: dates in range, identifiers that exist, amounts within limits, file paths inside the sandbox. These checks are deterministic code, not model calls.
3. **Grounding validation.** Claims in a final answer are checked against the evidence actually gathered: does each cited source appear in the tool results? Does each figure match a retrieved value? This step is what separates a research agent from a confident fabricator.
4. **Behavioural validation.** Where the action has an executable consequence, execute and observe: run the tests, compile the code, dry-run the transaction. External ground truth is the strongest validator available.
5. **Model-based critique.** Only after the deterministic layers, a separate evaluator call (ideally a different prompt, and sometimes a different model) assesses quality dimensions that code cannot — completeness, tone, relevance. This is the evaluator-optimiser pattern [4].

```
# Bounded repair loop: never retry forever, always say why
def validated_call(model, messages, schema, checks, max_repairs=3):
    for attempt in range(max_repairs + 1):
        out = model.generate(messages, response_format=schema)
        errors = schema.validate(out) or [e for c in checks if (e := c(out))]
        if not errors:
            return out
        messages += [assistant(out), user(f"Rejected: {errors}. Fix only these
issues.")]
    raise ValidationExhausted(errors) # escalate to a human or a fallback path
```

Sampling for reliability

When a decision is both important and cheap to recompute, sample several times and aggregate. Self-consistency demonstrated that majority voting over diverse reasoning paths improves accuracy on reasoning benchmarks [2]; the same idea applies to classification steps, extraction steps, and plan selection inside agents. Disagreement among samples is itself a valuable signal: high variance flags an input the system finds ambiguous, which is precisely where human review should be routed.

Termination discipline

Finally, every loop needs explicit exit conditions beyond “the model said it was done”: a maximum step count, a token or cost budget, a wall-clock timeout, a no-progress detector (for example, the same tool called with near-identical arguments three times), and a defined escalation path when any of these trips. An agent that fails loudly and early is far more valuable than one that fails silently after consuming its entire budget.

CHAPTER SUMMARY

- ReAct interleaves thought, action, and observation; it is adaptive and interpretable, but costly per step and prone to loops and premature stops.
- Planner-executor designs (Plan-and-Solve, ReWOO) trade adaptivity for efficiency and auditability; ToT and GoT buy breadth and aggregation at a higher price. Choose the topology per step, not per system.
- Long contexts degrade: information in the middle is under-used, and performance becomes less reliable as length grows. Manage context as an attention budget with compaction, clearing, JIT retrieval, notes, and sub-agents.
- Impose discipline with a validation ladder — syntactic, semantic, grounding, behavioural, then model critique — and with hard termination rules.

Memory Systems for Persistent Intelligence

An agent without memory is condemned to rediscover the world on every run — and to pay for it each time.

A foundation model is stateless. Everything it “knows” about the task in front of it must be present in its context at the moment of inference. Memory is the machinery that decides what goes into that context, drawn from what stores, and in what form. This chapter introduces a three-tier memory architecture, shows how to keep long-lived agents from drowning in their own history through compression and principled retrieval, and closes with procedural memory — the store that lets an agent get better at its job rather than merely remembering it.

3.1 The Three-Tier Memory Architecture

Cognitive science distinguishes working memory from long-term memory, and divides the latter into episodic (events), semantic (facts), and procedural (skills) stores. The CoALA framework imports this division directly into the design of language agents [7], and it maps well onto the components practitioners actually build. For operational clarity, this book groups them into three tiers plus a fourth store treated in §3.3.

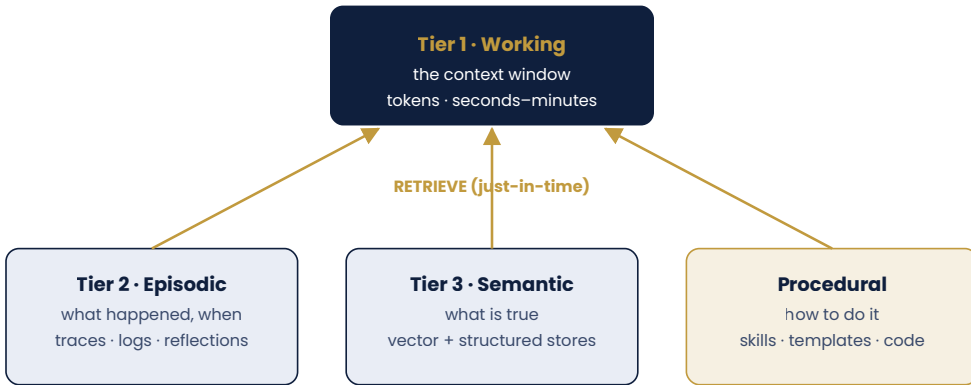


Figure 3.1 — Memory tiers. Long-term stores never feed the model directly; everything passes through retrieval into the scarce working tier.

Tier 1 — Working memory (the context window)

Working memory is whatever is in the prompt right now: system instructions, the goal, recent turns, tool results, and anything retrieved from deeper tiers. It is the only memory the model can reason over directly, and — as Chapter 2 established — its effective quality declines as it fills [16, 17]. Working memory should be treated as a cache, not a database: everything in it should be there because it was deliberately placed there for the current step.

Tier 2 — Episodic memory (the historical trace)

Episodic memory records what the agent experienced: the sequence of observations, actions, and outcomes, stamped with time and context. The Generative Agents work offers the most influential design. Each agent keeps a *memory stream* — a time-ordered log of natural-language records, each carrying a creation timestamp and a last-accessed timestamp — and retrieves from it by scoring every record on three factors: **recency** (exponential decay since last access), **importance** (a 1–10 rating assigned by the model when the memory is written), and **relevance** (embedding similarity to the current query) [18]. The top-scoring records enter working memory.

That triple of recency, importance, and relevance is a robust default for almost any episodic store. It encodes three intuitions a pure vector search misses: recent events are usually more pertinent; some events matter far more than others regardless of topical similarity; and topical similarity alone surfaces trivia. In pro-

duction, the importance score is often supplemented or replaced by structural signals — an event that involved an irreversible action, an error, or an explicit user instruction is important by definition.

Episodic memory is also where *reflection* lives. In Generative Agents, when the accumulated importance of recent events crosses a threshold, the agent asks itself salient questions about its experiences, retrieves the relevant memories, and writes higher-level inferences — with pointers back to the supporting records — into the stream as new memories [18]. Reflexion uses a similar mechanism for task improvement, storing self-critiques after failures and replaying them on the next attempt [8]. Reflections are the bridge from episodic to semantic memory: they turn “what happened” into “what is generally true.”

Tier 3 — Semantic memory (vectorised knowledge bases)

Semantic memory holds facts that are true independent of when the agent learned them: product documentation, policies, customer records, prior findings, domain knowledge. In current practice it is usually implemented as a combination of a *vector index* over chunked documents (for fuzzy, meaning-based recall) and *structured stores* — relational tables, key-value stores, knowledge graphs — for facts that need exact lookup, aggregation, or relational queries. Retrieval-augmented generation (Chapter 5) is the discipline of pulling from this tier well [19].

Two design rules prevent the most common semantic-memory failures. First, **store provenance with every fact**: source, date, and the episode in which it was learned. Without provenance, the agent cannot resolve conflicts between facts or explain its claims. Second, **separate the agent’s beliefs from ground truth**. A fact the agent inferred is not the same as a fact a system of record asserts; mixing them silently is how a single hallucination becomes institutional knowledge.

Memory as an operating-system problem

MemGPT reframed the whole arrangement as virtual memory management. Borrowing from operating systems, it treats the context window as fast “main” memory and external stores as slower storage, and it gives the model functions to page information between them — searching archives, writing to persistent storage, editing a core memory block — with interrupt-style control flow to manage when the model yields [20]. The authors demonstrated the approach on document analysis far exceeding the native context window and on multi-session chat agents that remember and evolve across conversations [20]. The lasting engineer-

ing lesson is that *memory operations should be explicit tools* that the agent invokes deliberately and the harness logs, rather than invisible pre-processing the model cannot see or control.

3.2 Semantic Compression and Retrieval

An agent that logs everything and retrieves naively will, over enough episodes, exhaust its token budget on its own history. The remedy is a lifecycle in which memories are written selectively, compressed progressively, indexed for multiple access patterns, and eventually retired.

Write selectively

Not every observation deserves to persist. A good write policy records: decisions and their rationale; facts established from authoritative sources (with provenance); user instructions and preferences; errors and their resolutions; and irreversible actions taken. It discards raw tool output once its salient content has been extracted, intermediate reasoning that led nowhere, and anything already present in a system of record that can be re-queried. Re-queryable data should be stored as a *pointer*, not a copy.

Compress progressively

Compression should follow a gradient, in the way human memory turns vivid recent detail into durable gist:

1. **Verbatim** for the current episode's recent turns.
2. **Extractive summary** for earlier turns of the current episode (key facts and decisions, with identifiers preserved exactly).
3. **Abstractive episode digest** once an episode closes: goal, outcome, what worked, what failed, open items.
4. **Reflections and consolidated facts** distilled across many episodes and promoted into semantic memory.

At every stage, certain content is *non-compressible* and must survive verbatim: identifiers, quantities, dates, direct user constraints, and commitments made on the user's behalf. Specifying the non-compressible set explicitly is the difference between compaction that preserves continuity and compaction that silently corrupts the task [11].

Index for more than one access pattern

Pure embedding similarity is a weak retrieval key for agent memory. Effective systems index each record along several dimensions and combine them at query time:

Index	Answers the question	Typical mechanism
Semantic	What is about this topic?	Dense embeddings, cosine similarity
Lexical	What mentions this exact term or ID?	BM25 / keyword index
Temporal	What happened recently, or in this window?	Timestamp index with decay scoring
Entity	What concerns this customer, file, or project?	Entity tags, foreign keys, graph edges
Salience	What mattered most?	Importance score, error/irreversibility flags

A hybrid score — for example, a weighted sum of normalised semantic, lexical, recency, and importance components, filtered by entity — consistently beats any single index, and the weights become tunable parameters that can be evaluated like any other component (Chapter 8).

Retire deliberately

Memories go stale. Facts are superseded, preferences change, and procedures stop working when the systems they target change. Each record should carry a validity signal — an expiry date, a “last confirmed” timestamp, or a supersession link — and retrieval should down-rank or exclude stale entries. Deletion must also be possible on request: an agent that cannot forget is a compliance liability as much as a performance one.

TRADECRAFT NOTE — MEMORY IS AN ATTACK SURFACE

Anything an agent writes to memory, it will later read as trusted context. If an attacker can get an instruction into a retrieved web page or document, and the agent stores a summary of that content, the injection can persist across sessions. Tag every memory with its trust level and source, and never let content from untrusted sources be promoted into instructions. Chapter 9 returns to this.

3.3 Procedural Memory

Episodic memory remembers what happened; semantic memory remembers what is true; *procedural memory* remembers how to do things. It is the store that lets an agent improve with experience without retraining.

Skills as executable code

The clearest demonstration comes from Voyager, an agent that plays Minecraft. Voyager combines an automatic curriculum that proposes progressively harder goals, an iterative prompting loop that refines generated code using execution errors, environment feedback, and a self-verification check, and — crucially — a **skill library** that stores each successfully verified program as executable code, indexed by an embedding of its description, so that relevant skills can be retrieved and composed when new goals arrive [21]. The authors report that, relative to prior methods, Voyager collected 3.3 times as many unique items, travelled 2.3 times as far, and unlocked key tech-tree milestones up to 15.3 times faster [21].

The pattern generalises well beyond games. A data-engineering agent that has once worked out how to authenticate against a particular API, paginate its results, and normalise its schema can store that sequence as a parameterised function; next time, it retrieves and calls the function in one step instead of rediscovering the procedure across a dozen tool calls.

Workflow templates

Not every procedure is code. Many are *workflow templates*: an ordered set of steps, the tools each step uses, the checks that confirm success, and known pitfalls. A template for “investigate a failed payment” might specify: look up the transaction; check the processor’s status; check the customer’s account flags; compare with known incident windows; draft a resolution; require approval if a refund exceeds a threshold. Templates are retrieved by task type and injected as guidance, constraining an otherwise open-ended ReAct loop into a proven path.

Governing a skill library

Procedural memory is powerful precisely because the agent will trust and reuse it, which makes its integrity critical. Practical governance rules include:

- **Promote only verified procedures.** A skill enters the library only after it has succeeded against an external check (tests passed, schema validated, human approved), never on the model's own say-so.
- **Version and attribute every skill.** Record which episode created it, which model version, and against which system versions it was validated.
- **Re-validate on drift.** When a target system changes, mark dependent skills as suspect and re-test before reuse.
- **Keep skills least-privileged.** A stored procedure should declare the permissions it needs, and the harness should grant no more (Chapter 9).
- **Measure reuse.** Skills that are never retrieved are clutter; skills that are retrieved and then fail are hazards. Both should be pruned.

CHAPTER SUMMARY

- Organise memory as working (context), episodic (trace), semantic (facts), and procedural (skills) stores, with all long-term memory reaching the model only through retrieval.
- Score episodic recall on recency, importance, and relevance; use reflection to distil episodes into durable knowledge.
- Write selectively, compress progressively, preserve a defined non-compressible set, index along multiple dimensions, and retire stale entries.
- Make memory operations explicit, logged tools; treat memory as an attack surface.
- Procedural memory — verified, versioned skills and workflow templates — is how an agent gets better at its job without retraining.

PART II

Engineering and Tool Integration

An agent is only as capable as the interfaces through which it touches the world. Part II treats tools and retrieval as first-class engineering disciplines: how to design interfaces for a non-human user, how to make function calling robust, and how to turn retrieval from a fixed pre-processing step into an active, self-correcting search.

The Agent-Computer Interface (ACI) and Tool Use

We spent forty years learning to design interfaces for people. The next user is not a person.

When a language model calls a tool, it is a user of an interface — an unusual user, fluent and fast but literal, forgetful between sessions, and unable to see anything that is not written down. The software-engineering agent literature named this layer the *agent-computer interface* (ACI), by analogy with the human-computer interface, and showed that designing it deliberately can matter as much as choosing a stronger model [10]. This chapter sets out the principles of ACI design, the mechanics of function calling and validation, and the error-proofing techniques that keep agents out of failure spirals.

4.1 Designing for Machines

The SWE-agent project made the case empirically. Rather than handing a model a raw Linux shell, its authors built a custom interface with commands tailored to how a language model reads and edits code — viewing files in bounded windows, searching a repository, editing specific line ranges with immediate syntax feedback, and running tests. With this ACI the agent resolved 12.5% of SWE-bench issues at pass@1 and reached 87.7% on HumanEvalFix, well beyond what earlier non-interactive approaches achieved [10]. The point is not the specific numbers, which the field has long since surpassed, but the mechanism: the same model became markedly more capable when the interface was designed around its strengths and weaknesses.

Anthropic’s guidance draws the same conclusion from production experience, recommending that teams invest as much effort in agent-computer interfaces as they would in human-computer interfaces [4]. It reports a telling anecdote from building a SWE-bench agent: the model kept making mistakes with relative file

paths after changing directories, and when the tool was changed to require absolute paths, the errors disappeared [4]. No prompt engineering would have fixed this as reliably as removing the possibility of the mistake.

How an LLM “sees” a tool

A model never sees the implementation of a tool. It sees exactly three things: the tool’s **name**, its **description**, and its **parameter schema** — and, after calling it, the **result**. Every design principle in this chapter follows from that narrow channel. If a constraint is not expressed in one of those four places, the model does not know it exists.

Principles of ACI design

1. Design around tasks, not endpoints. A common anti-pattern is to wrap every endpoint of an existing API one-for-one. This produces dozens of fine-grained tools, forces the agent to chain many calls to accomplish anything, and floods its context with intermediate results. Anthropic’s tool-design guidance recommends instead building fewer, higher-leverage tools that match real workflows — for example a single scheduling tool that both finds availability and books the meeting, rather than separate tools for listing users, listing events, and creating events [22].

2. Namespace related tools. When an agent has access to many tools from many systems, consistent prefixes (`crm_contacts_search`, `crm_deals_update`) help the model choose the right one and reduce collisions between similarly named functions [22].

3. Return meaning, not plumbing. Tool outputs should favour human-interpretable fields over opaque identifiers — a customer name alongside (or instead of) a UUID — because models reason better over natural-language values and are less likely to hallucinate when identifiers carry meaning [22]. Where the agent will need an identifier for a follow-up call, return both.

4. Budget the output. A tool that can return ten thousand rows will, eventually, return ten thousand rows into the context window. Build in pagination, filtering, and truncation with sensible defaults, and when truncating, tell the agent so and how to narrow the request [22].

5. Write descriptions for a capable new colleague. A tool description is a prompt. It should state what the tool does, when to use it and when not to, what each parameter means (with units and formats), what the output contains, and any non-obvious behaviour. Implicit context that a human engineer would “just know” must be made explicit [4, 22].

6. Make the deterministic thing easy and the dangerous thing hard. Reads should be cheap and unrestricted within scope; writes should require explicit, specific parameters; destructive or irreversible operations should be separate tools with narrow scope and, often, a human gate (Chapter 9).

Human-oriented API habit	Agent-oriented alternative
Many small CRUD endpoints	Few task-shaped tools that compose the steps internally
Opaque IDs and status codes	Readable names and states, with IDs included for follow-up
Unbounded list responses	Paginated, filtered, truncated with a “narrow your query” hint
Terse error codes (E4021)	Actionable messages: what was wrong and how to fix it
Free-form strings for structured values	Enums, formats, ranges, and examples in the schema
Relative paths, implicit state	Absolute references; tools that are stateless where possible

Learning to use tools

Tool use is not purely an interface concern; models must also learn when to call a tool at all. Toolformer showed that a model could teach itself, from only a handful of demonstrations per API, to decide which tool to call, when, with what arguments, and how to fold the result into its prediction — using a calculator, a question-answering system, two search engines, a translator, and a calendar [23]. Today, function calling is trained into frontier models directly, but the lesson persists: the model’s decision to *not* call a tool is as important as its ability to call one correctly, and tool descriptions should say explicitly when a tool is unnecessary.

4.2 Function Calling Mechanics

Modern function calling follows a common pattern across providers. The application declares a set of tools, each with a name, a description, and a JSON Schema for its input. The model, when it decides to act, emits a structured tool-call object

instead of (or alongside) text. The harness validates the arguments, executes the function, and returns the result to the model in a tool-result message linked to the original call by an identifier. The model then continues.

```
{
  "name": "invoices_search",
  "description": "Search issued invoices. Use for questions about billing history, amounts owed, or payment status. Do NOT use for quotes or estimates (use quotes_search). Returns at most `limit` results, newest first; if more exist, the response includes `next_cursor`.",
  "input_schema": {
    "type": "object",
    "properties": {
      "customer_name": {"type": "string", "description": "Full or partial legal name."},
      "status": {"type": "string", "enum": ["draft", "open", "paid", "overdue", "void"]},
      "issued_after": {"type": "string", "format": "date", "description": "ISO 8601, inclusive."},
      "limit": {"type": "integer", "minimum": 1, "maximum": 50, "default": 10},
      "cursor": {"type": "string", "description": "Value of next_cursor from a prior call."}
    },
    "required": [],
    "additionalProperties": false
  }
}
```

Schema definition

The schema is the contract, and every constraint it can express should be expressed: types, enums for closed vocabularies, formats for dates and emails, numeric ranges, required fields, and `additionalProperties: false` to reject invented parameters. Descriptions on each property should include units and an example where the format is not obvious. A schema that says `"amount": {"type": "number"}` invites the model to guess whether the unit is dollars or cents; a schema that says `"amount_cents": {"type": "integer", "minimum": 1}` does not.

JSON validation

Validation happens twice. *Before execution*, the harness validates the model's arguments against the schema and against semantic rules the schema cannot express (does this customer exist? is this date in the future? does the caller have permission?). *After execution*, the harness validates the tool's output before handing it back — enforcing size limits, stripping or flagging content from untrusted sources, and normalising formats. Where a provider offers strict or constrained

decoding that guarantees schema-conformant arguments, enabling it removes the syntactic half of the problem; the semantic half always remains the harness’s job.

Error-recovery protocols

Tool calls fail for four broad reasons, and each deserves a distinct recovery path:

Failure class	Example	Recovery
Invalid arguments	Wrong enum value; missing field	Return a precise validation message; let the model retry (bounded)
Not found / empty	No matching records	Return an explicit empty result with suggestions for broadening the query
Transient infrastructure	Timeout, 503, rate limit	Retry in the harness with backoff and jitter — do not spend model calls on it
Permission / policy	Action not allowed for this user or scope	Return a clear refusal; do not let the model attempt workarounds; escalate if needed

The governing rule is that **the harness handles what is mechanical, and the model handles what requires judgement**. Retrying a timed-out HTTP call is mechanical. Deciding that a search returned nothing because the customer’s name was misspelled is judgement.

Parallel and sequential calls

Many models can emit several tool calls in one turn. Independent reads — fetching three documents, querying two systems — should run in parallel, cutting latency substantially. Calls with side effects, or calls whose arguments depend on earlier results, must be sequenced. A simple harness policy: run read-only tools concurrently; serialise anything that writes.

4.3 Error Handling and Poka-Yoke Tool Design

Poka-yoke — mistake-proofing — is a manufacturing discipline associated with Shigeo Shingo, who combined it with source inspection to prevent defects at the point where they would arise rather than catching them downstream [24]. Its central idea is that the best way to stop an error is to design the process so the error cannot be made, or is detected immediately when it is. Anthropic’s guidance applies the term directly to agent tools, recommending that teams change parameters and interfaces so that mistakes become harder to make [4].

Poka-yoke patterns for agent tools

- **Eliminate ambiguous inputs.** Absolute paths instead of relative; ISO dates instead of free-text; integer cents instead of floating-point dollars; enums instead of strings.
- **Make preconditions explicit and checkable.** An edit tool can require the expected current content of the region being replaced; if the file has changed since the agent last read it, the edit is rejected with the current content attached.
- **Separate dangerous capabilities.** `files_delete` is its own tool with its own permission, never a flag on `files_write`.
- **Provide dry runs.** Expensive or destructive operations accept `dry_run=true`, return what *would* happen, and require a second explicit call to commit.
- **Make operations idempotent.** Writes accept a client-supplied idempotency key so that a retried call cannot double-charge, double-send, or double-create.
- **Give immediate feedback.** SWE-agent's editor ran a linter on every edit and rejected syntactically broken changes on the spot, so errors were corrected at the moment they were introduced [10].

Errors as instructions

An error message returned to a model is a prompt. It should tell the agent what went wrong, why, and what to do next — in plain language [22]. Compare:

```
# Unhelpful
{"error": "400 Bad Request"}

# Agent-oriented
{"error": "invalid_argument",
 "field": "issued_after",
 "message": "Expected an ISO 8601 date such as 2026-09-30; received 'last month'.
            Convert relative dates using the current date provided in your
            context.",
 "retryable": true}
```

Breaking infinite loops

The most expensive agent failure is the loop that never ends: the same search re-run with cosmetic variations, an edit applied and reverted repeatedly, or a polling call that waits on a condition that will never become true. Defences belong in both the tools and the harness:

1. **Repetition detection.** The harness fingerprints each tool call (name plus normalised arguments). When the same fingerprint, or a near-duplicate, recurs beyond a threshold, it injects a message telling the agent that the approach is not working and asking it to change strategy or stop.
2. **Progress signals.** Tools that report state (tests passing, records processed) let the harness detect when several steps have produced no forward movement.
3. **Escalating guidance.** On the first repeat, a hint; on the second, a required change of approach; on the third, termination and escalation with a summary of what was tried.
4. **Hard ceilings.** Step limits, token budgets, and wall-clock timeouts are non-negotiable backstops (Chapter 2).
5. **Exit tools.** Give the agent a legitimate way out — a `request_human_help` or `report_blocked` tool — so that “I cannot complete this” is an action it can take rather than a conclusion it must avoid.

TRADECRAFT NOTE — TEST TOOLS WITH THE AGENT, NOT JUST WITH UNIT TESTS

A tool can pass every unit test and still be unusable by a model. Run the agent against realistic tasks, read the transcripts, and look for where it hesitates, misuses parameters, or calls the wrong tool. Those are interface defects. Anthropic recommends exactly this evaluation-driven loop, including having agents themselves analyse transcripts and propose tool improvements [22].

CHAPTER SUMMARY

- The model sees a tool only through its name, description, schema, and result; design all four deliberately.
- Prefer few task-shaped tools, namespaced, returning meaningful and bounded output.
- Validate arguments before execution and results after; retry mechanical failures in the harness; route judgement back to the model.
- Apply poka-yoke: remove ambiguity, check preconditions, separate dangerous operations, offer dry runs and idempotency, give immediate feedback.
- Write errors as instructions, detect repetition, and give the agent a legitimate way to stop.

Advanced Retrieval-Augmented Generation and Agentic Search

Retrieval is not a lookup. It is an investigation – and investigations need judgement about which leads to trust.

Retrieval-augmented generation began as an architecture: pair a generator with a retriever so that answers can draw on an external corpus. In agentic systems it becomes a behaviour. The agent decides whether to retrieve, what to ask for, how to cut documents so the right evidence is findable, whether what came back is any good, and how to reconcile evidence from databases, APIs, and unstructured files that disagree. This chapter follows that evolution from static pipelines to self-correcting search.

5.1 From Static RAG to Agentic Chunking

The original architecture

Lewis and colleagues introduced retrieval-augmented generation as a model that combines parametric memory (a pre-trained sequence-to-sequence generator) with non-parametric memory (a dense vector index of Wikipedia accessed by a neural retriever), and showed that it set the state of the art on several open-domain question-answering tasks while producing more specific and factual text than a generator alone [19]. Two benefits motivated the design and still motivate it today: knowledge can be updated by editing the index rather than retraining the model, and answers can be traced back to retrieved passages.

The standard production pipeline that grew from this work is linear: split documents into fixed-size chunks; embed each chunk; at query time, embed the question, fetch the top- k nearest chunks, paste them into the prompt, and generate. It is simple, and its weaknesses are well known:

- **Fixed chunking cuts meaning in half.** A 512-token window will split a definition from its exception, a table from its header, or a contract clause from the clause that modifies it.
- **Retrieval is unconditional.** The system retrieves even when the question needs no external knowledge, and it never retrieves a second time when the first attempt misses.
- **Quality is unchecked.** Whatever comes back is trusted, so an irrelevant or misleading chunk is passed straight to the generator.
- **One source, one shot.** The pipeline cannot combine a vector search with a SQL query or an API call, nor plan a sequence of lookups whose later steps depend on earlier findings.

Structure-aware and semantic chunking

The first improvement is to chunk along the document's own structure — headings, sections, list items, table boundaries, function definitions in code — rather than along an arbitrary token count. Structure-aware chunking is deterministic, cheap, and should be the default whenever a document has reliable structure. When it does not (transcripts, scanned reports, long narratives), *semantic chunking* places boundaries where the embedding similarity between adjacent sentences drops, approximating topic shifts.

Agentic chunking

Agentic chunking goes one step further and asks a language model to decide where boundaries belong. LumberChunker is a representative method: it feeds the model a run of consecutive paragraphs up to a token budget and asks it to identify where the content begins to shift substantially; that point becomes a boundary, and the process repeats from there through the document [25]. Evaluated on GutenQA, a benchmark of 3,000 needle-in-a-haystack questions over 100 public-domain books, it reached a DCG@20 of about 62 against roughly 55 for the strongest baseline, recursive chunking, with a corresponding gain in recall [25].

In an agent, the idea extends naturally to *perception-driven segmentation*: when the agent first opens a large document, it builds a lightweight map — a table of contents with summaries and anchors for each segment — and stores that map rather than the full text. Later steps retrieve by navigating the map, pulling only the segments a question requires. This is just-in-time context (Chapter 2) applied to documents, and it scales to corpora far larger than any context window.

Strategy	Boundary decided by	Cost	Best for
Fixed-size with overlap	Token count	Lowest	Homogeneous text; quick baselines
Structure-aware	Headings, sections, code units	Low	Well-formatted documentation, code, contracts
Semantic	Embedding similarity drops	Moderate	Unstructured prose, transcripts
Agentic (LLM-decided)	Model judgement of topic shift	High at ingest	High-value long documents where precision matters
Hierarchical map + JIT	Agent navigation at query time	Spread across queries	Very large corpora; investigative work

TRADECRAFT NOTE — KEEP THE PARENT

Whatever the chunking method, store each chunk with a pointer to its parent section and document, plus its position. Retrieval can then match on the small, precise chunk but hand the generator the surrounding section — small-to-big retrieval — and every claim can be cited back to an exact location.

5.2 Corrective and Self-RAG

The most damaging RAG failure is not a miss; it is a confident answer built on an irrelevant or misleading passage. Once bad evidence is in the context, the generator tends to use it, and in an agent the resulting error propagates into every downstream step. Two research lines address this by evaluating evidence *before* it is allowed to shape the answer.

Self-RAG: retrieve on demand, critique as you go

Self-RAG trains a single language model to emit special *reflection tokens* that control and critique its own process [26]. A `Retrieve` token decides whether fetching passages would help at all, so retrieval happens only when needed. For each re-

trieved passage, an `IsRel` token judges relevance; after generating a segment, an `IsSup` token judges whether the passage fully, partially, or does not support it; and an `IsUse` token rates the overall usefulness of the response [26]. At inference time, candidate continuations from different passages can be generated in parallel and ranked by these critique scores, and the weighting can be tuned – for instance, favouring strict citation support over completeness – without retraining [26].

Corrective RAG: grade the evidence, then act

Corrective Retrieval Augmented Generation (CRAG) takes a modular, plug-in approach. A lightweight retrieval evaluator – a fine-tuned T5-large model – scores how relevant the retrieved documents are to the question, and the score triggers one of three actions: **Correct** (at least one document is confidently relevant, so refine and use the retrieved knowledge), **Incorrect** (nothing clears the lower threshold, so discard it and turn to web search with a rewritten, keyword-style query), or **Ambiguous** (combine both) [27]. Retrieved documents are then passed through a decompose-then-recompose step that splits them into fine-grained knowledge strips, filters out the irrelevant strips, and reassembles the remainder, so the generator sees signal rather than noise [27].

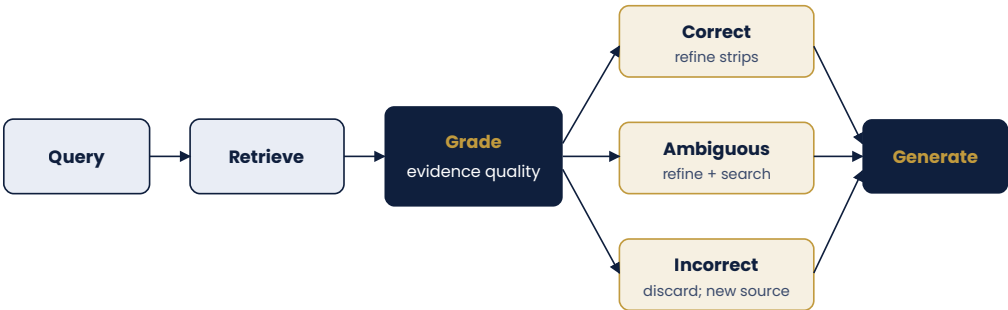


Figure 5.1 — A corrective retrieval loop, generalised from CRAG. In an agent, “Incorrect” routes to a different tool or query rather than terminating.

Building a corrective loop in an agent

An agentic system does not need a specially trained model to apply these ideas; it can compose them from tools and validation steps:

- 1. **Decide whether to retrieve.** Let the agent answer from context when it can, and require retrieval for any claim about current state, specific records, or cited facts.

- 2. **Grade every result.** Apply a relevance check (a small classifier, a reranker score threshold, or a cheap model call) before evidence enters the working context.
- 3. **Act on the grade.** Use relevant evidence; for weak evidence, reformulate the query, switch sources, or broaden scope; for no evidence, say so rather than improvise.
- 4. **Check support after generating.** Verify that each claim in the draft is supported by a retrieved passage, and strip or flag those that are not — the grounding validation of Chapter 2.
- 5. **Cite.** Carry source identifiers through to the final answer so that a human can audit every claim.

5.3 Multi-Source Synthesis

Real questions rarely live in one store. “Which of our enterprise customers in Atlantic Canada have open support tickets about the new billing module, and what does our documentation say about the bug?” touches a CRM (structured), a ticketing system (API), and a documentation corpus (unstructured). An agent that can search only one of these gives a partial answer presented as a complete one.

Routing

The first decision is where to look. A router — which can be a classification step, a model call, or simply well-described tools that the agent chooses among — maps sub-questions to sources by the *shape* of the information needed:

Information shape	Best source	Retrieval mechanism
Exact records, counts, aggregates	Relational database / warehouse	Generated SQL against a documented schema (read-only role)
Live state of an external system	Vendor API	Function call via task-shaped tool or MCP server
Explanations, policies, procedures	Document corpus	Hybrid vector + keyword search with reranking
Relationships among entities	Knowledge graph	Graph traversal query
Recent public information	Web	Search plus fetch, with source-quality filtering

Decomposition and orchestration

Complex questions decompose into a dependency graph of sub-queries. Independent branches run in parallel; dependent ones wait for their inputs. This is exactly the planner-executor and ReWOO pattern from Chapter 2 applied to retrieval: plan the sub-queries with placeholders, execute them across sources, and solve over the assembled evidence [13]. When a branch's results fail grading, that branch alone falls back to an adaptive ReAct-style search.

Reconciliation

Sources disagree. A CRM may list a customer as active while the billing API shows them cancelled; documentation may describe behaviour that the latest release changed. Synthesis therefore needs explicit reconciliation rules:

- **Authority:** designate a system of record for each entity type, and let it win conflicts.
- **Recency:** when authority is equal, prefer the more recently updated source, and say so.
- **Transparency:** when a conflict cannot be resolved, surface it in the answer rather than silently choosing.
- **Provenance:** attach source, timestamp, and query to every fact carried into the final synthesis.

Security at the seams

Multi-source agents read content they do not control — web pages, uploaded files, third-party API responses — and any of it may contain text crafted to look like instructions. Retrieved content must be treated as data, never as commands, and agents that can both read untrusted content and take consequential actions need the architectural defences described in Chapter 9 [28].

CHAPTER SUMMARY

- Static RAG chunks blindly, retrieves unconditionally, trusts whatever returns, and reads one source once.
- Chunk along structure where it exists; use semantic or agentic (LLM-decided) boundaries for high-value unstructured documents; keep parent pointers for small-to-big retrieval and citation.
- Grade evidence before it shapes the answer: Self-RAG critiques with reflection tokens; CRAG grades and routes to correct, ambiguous, or incorrect paths.
- Route sub-questions by information shape, decompose into a dependency graph, reconcile conflicts by authority and recency, and carry provenance to the end.

PART III

Multi-Agent Ecosystems and Orchestration

One agent hits limits — of context, of specialisation, of parallelism. Many agents can push past them, at a price. Part III covers how agents are arranged, how they talk, what the arrangement costs, and how to keep a society of fallible reasoners from deadlocking, contradicting one another, or failing together.

Multi-Agent Topologies and Communication Protocols

Adding agents adds capacity. It also adds every problem that organisations of people have ever had.

A multi-agent system divides a task among several model-driven loops that each hold their own context, tools, and role. Done well, this buys parallelism, specialisation, and isolation of context. Done carelessly, it buys duplicated work, contradictory outputs, runaway token spend, and failures that no single trace explains. This chapter catalogues the main topologies, the protocols that connect agents to tools and to each other, and the economics that determine when a swarm is worth its overhead.

6.1 Architectural Patterns

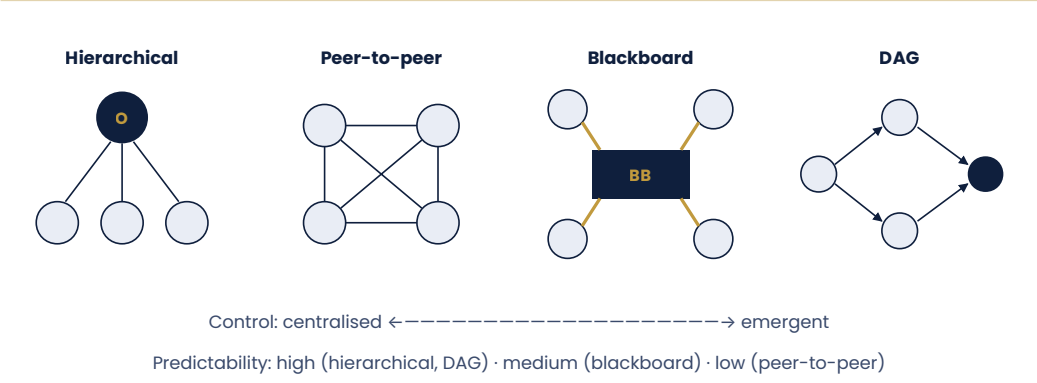


Figure 6.1 — Four canonical multi-agent topologies.

Hierarchical (orchestrator-worker)

A lead agent decomposes the task, delegates subtasks to worker agents, and synthesises their results. Workers do not talk to each other; all coordination flows through the orchestrator. This is the most widely deployed topology because it

preserves a single point of accountability and a single place to enforce budgets and policy. Anthropic’s multi-agent research system is an example: a lead agent plans the research and spawns subagents that explore different angles in parallel, each with its own context, before returning condensed findings for the lead to combine [29].

The orchestrator’s delegation messages are the critical artifact. Each must give the worker an objective, the expected output format, the tools and sources to use, explicit boundaries (what not to do, what another worker is already covering), and a budget. Vague delegation produces duplicated effort and gaps – a lesson the Anthropic team reported learning directly [29].

Peer-to-peer (conversational)

Agents exchange messages directly, without a fixed controller. AutoGen popularised this style with “conversable” agents that can be backed by models, humans, tools, or combinations of these, and whose interaction patterns are programmed in natural language or code [30]. Peer-to-peer topologies suit open-ended collaboration such as debate, critique, and role-play, but they are the hardest to bound: without a controller, termination, turn-taking, and budget must be enforced by protocol rules.

Blackboard

The blackboard architecture dates to the Hearsay-II speech-understanding system, in which independent knowledge sources cooperated by reading and writing hypotheses on a shared global data structure, with a scheduler deciding which source to activate next [31]. Agents do not address each other; they observe the shared state and contribute when they can. MetaGPT revived a version of this for LLM agents: role-specialised agents publish structured outputs to a shared message pool and subscribe only to what their role needs [32]. Blackboards decouple agents cleanly and make the global state inspectable, but they require careful concurrency control (Chapter 7).

Directed acyclic graph (DAG) execution

The task is compiled in advance into a graph whose nodes are agent calls or deterministic functions and whose edges are data dependencies. A scheduler runs each node when its inputs are ready, parallelising independent branches. DAGs are the multi-agent analogue of ReWOO and Graph of Thoughts (Chapter 2):

highly predictable, cheap to monitor, and easy to retry node by node, but rigid when the task's structure is not known up front. Many production systems combine a DAG for the predictable skeleton with orchestrator-worker loops inside nodes that need adaptivity.

Structure beats chatter

MetaGPT's central argument is worth internalising regardless of topology. Its authors observed that naively chaining LLMs produces cascading hallucinations, and responded by encoding human Standard Operating Procedures into the workflow: product-manager, architect, engineer, and QA roles each produce structured documents — requirements, designs, interface specifications — that the next role consumes [32]. Agents that exchange typed artifacts rather than free-form conversation are easier to validate, easier to debug, and less prone to drift.

6.2 Communication Standards

Agents must communicate with two kinds of counterpart: *tools and data sources*, and *other agents*. Each now has a de facto open protocol, and both inherit ideas from a much older standard.

Agent Communication Languages (ACL)

The Foundation for Intelligent Physical Agents (FIPA) standardised an agent communication language in the late 1990s and early 2000s. A FIPA ACL message is built around a *performative* — the type of communicative act, such as inform, request, propose, or refuse, and the only mandatory parameter — together with optional parameters for the participants (sender, receivers, reply-to), the content and how to interpret it (content, language, encoding, ontology), and conversation control (protocol, conversation-id, reply-with, in-reply-to, reply-by) [33]. LLM agents rarely use FIPA ACL directly, but its design choices — explicit intent, explicit ontology, explicit conversation threading, explicit deadlines — are exactly what free-form natural-language messages between agents lack, and well-designed modern protocols reintroduce them.

Model Context Protocol (MCP)

Anthropic introduced the Model Context Protocol on 25 November 2024 as an open standard for connecting AI applications to external tools and data [34]. MCP uses JSON-RPC 2.0 messages between a *host* application, the *clients* it runs, and *servers* that expose capabilities through three main primitives: **tools** (functions the model can invoke), **resources** (data the application can read into context), and **prompts** (reusable templates) [34]. Adoption spread rapidly: OpenAI adopted it in March 2025 and Google DeepMind followed, and in December 2025 Anthropic transferred the protocol to the Agentic AI Foundation, a directed fund under the Linux Foundation co-founded with Block and OpenAI [34].

MCP's significance for architecture is that it turns the agent-computer interface of Chapter 4 into a reusable, distributable component: a well-designed MCP server for a CRM or a code repository can serve any compliant agent. Its significance for security is the converse — an MCP server is third-party code and third-party content inside the agent's trust boundary, and researchers have documented prompt-injection and tool-poisoning risks that can lead to data exfiltration [34]. Chapter 9 treats MCP servers as untrusted by default.

Agent2Agent (A2A)

Where MCP connects agents to tools, Google's Agent2Agent protocol, announced in April 2025, addresses communication between agents themselves, including agents built by different vendors on different frameworks [35]. A2A agents advertise their capabilities through *Agent Cards* — machine-readable metadata documents — and exchange messages using standard web technologies such as HTTP and JSON, with authentication built on established standards including TLS, JSON Web Tokens, and OpenID Connect [35]. In June 2025 Google transferred A2A to the Linux Foundation for vendor-neutral governance, with more than a hundred technology companies supporting it [35].

	FIPA ACL	MCP	A2A
Connects	Agent ↔ agent	Agent (host) ↔ tools & data	Agent ↔ agent (cross-vendor)
Era	Classical MAS (standard 2002)	LLM agents (Nov 2024)	LLM agents (Apr 2025)
Message basis	Performatives with ontology	JSON-RPC 2.0	HTTP + JSON; Agent Cards for discovery
Core abstraction	Communicative act	Tools, resources, prompts	Capabilities and delegated tasks
Governance	FIPA standards body	Agentic AI Foundation (Linux Foundation)	Linux Foundation

DESIGN PRINCIPLE — TYPED MESSAGES BETWEEN AGENTS

Whatever transport is used, define a schema for inter-agent messages: intent (request, report, refuse, escalate), task identifier, conversation identifier, payload as a typed artifact, confidence, sources, and deadline. Natural-language prose belongs inside the payload, not in place of it.

6.3 Mitigating the Complexity Tax

Every additional agent adds model calls, context that must be re-established, coordination messages, and new ways to fail. This overhead is the *complexity tax*, and it is substantial.

The token bill

Anthropic reported that, in its own data, agents used roughly four times as many tokens as chat interactions and multi-agent systems roughly fifteen times as many [29]. The same system showed why the bill can be worth paying: a multi-agent configuration with a stronger lead model and lighter subagents outperformed a single strong agent by 90.2% on Anthropic’s internal research evaluation, and analysis of its browsing benchmark found that three factors explained 95% of performance variance, with token usage alone explaining 80% [29]. In other words, much of the benefit of a multi-agent design comes from its ability to spend more tokens productively in parallel contexts. That is a good trade for broad, parallelisable, high-value tasks — and a poor one for tasks that are narrow, sequential, or cheap.

The failure surface

Multi-agent systems also fail in new ways. An analysis of more than 1,600 annotated execution traces across seven open-source multi-agent frameworks identified fourteen distinct failure modes grouped into three categories: **system design issues** (poor role and task specification), **inter-agent misalignment** (breakdowns in information exchange and coordination), and **task verification** failures (inadequate checking, premature termination) [36]. The authors' central conclusion is that many failures stem from organisational design rather than from model capability, and that stronger models alone will not fix them [36].

Controls

1. **Justify every agent.** Add an agent only when it brings a distinct context, a distinct tool set, or real parallelism. Two agents with the same tools and overlapping roles are one agent with extra overhead.
2. **Scale effort to the task.** Encode explicit rules for how many subagents and tool calls a task class warrants; Anthropic reported embedding such scaling guidance in its orchestrator's prompt after early versions over-spawned for simple queries [29].
3. **Mix model tiers.** Use the strongest model where judgement concentrates (planning, synthesis, verification) and cheaper models for bounded, well-specified subtasks.
4. **Return compressed results.** Workers should return concise, structured findings with references to full artifacts stored elsewhere, not their entire transcripts, so the orchestrator's context stays clean.
5. **Budget hierarchically.** Give each subtree a token, time, and tool-call budget; when a budget is exhausted, the node returns its best partial result with a status flag.
6. **Prefer structure.** A DAG or orchestrator-worker design with typed artifacts is cheaper and more reliable than open peer-to-peer conversation for most production work.

CHAPTER SUMMARY

- Hierarchical, peer-to-peer, blackboard, and DAG topologies trade control and predictability against flexibility; hybrids are common.
- Exchange typed artifacts, not chatter; encode procedures and roles explicitly.
- MCP standardises agent-to-tool connections; A2A standardises agent-to-agent communication; FIPA ACL's explicit intent, ontology, and threading remain good design guidance.
- Multi-agent systems can use around fifteen times the tokens of chat; they pay off for broad, parallel, high-value work, and fail mostly through organisational design flaws. Justify every agent and budget every subtree.

Conflict Resolution, Deadlocks, and Consensus

Distributed systems engineering spent fifty years learning that independent actors sharing state will collide. Agents do not get an exemption.

When several agents act on the same world, their actions can interfere: two agents edit the same file, one agent reads a record another is halfway through changing, three agents wait on each other indefinitely, or a group splits on what the right answer is. These are classical distributed-systems problems wearing new clothes, with one twist — the participants are not merely slow or crashed when they fail; they can be confidently wrong. This chapter adapts the established remedies for state management, consensus, and fault tolerance to societies of language-model agents.

7.1 Distributed Agent State Management

Where conflicts arise

Agent conflicts fall into four families, each with a well-understood counterpart in conventional systems:

Conflict	Agent example	Classical analogue
Lost update	Two coding agents each read <code>config.yaml</code> , each make a different change, each write back; one change vanishes.	Write-write race
Stale read	A billing agent issues a refund based on an order status that a fulfilment agent changed a second earlier.	Read-write race; isolation violation
Contradictory action	A support agent promises a customer a callback while a retention agent closes the account.	Semantic conflict between transactions
Deadlock / livelock	Agent A waits for B's review while B waits for A's revision; or both keep yielding to each other.	Circular wait; livelock

Principles

Single writer per resource. The simplest and strongest rule: assign each mutable resource — a file, a record, a section of a document, a ticket — exactly one owning agent at a time. Other agents may read it, or may submit change requests to the owner, but may not write it directly. Orchestrator-worker topologies make this easy by having the orchestrator assign disjoint scopes; blackboard topologies need explicit ownership fields.

Optimistic concurrency with version checks. When single-writer is impractical, every write carries the version (or content hash) of the state the agent read. If the resource has changed since then, the write is rejected and the agent receives the current state to reconcile against. This is the agent-level form of the precondition-checking poka-yoke from Chapter 4, and it converts silent lost updates into explicit, recoverable conflicts.

Leases, not locks. Pessimistic locks are dangerous with agents, because an agent holding a lock may stall, loop, or crash without releasing it. Use time-limited leases that expire automatically, are renewed by heartbeat while the agent is making progress, and are reclaimed by the scheduler otherwise.

Event-sourced shared state. Rather than letting agents mutate shared state in place, have them append events (“agent C proposed refund of \$40 on order 1182”) to an ordered log from which the current state is derived. The log provides a full audit trail, makes conflicts visible, and lets a coordinator or arbiter apply domain rules when events clash.

Idempotent, compensable side effects. External actions should be idempotent (Chapter 4) and, where possible, paired with a compensating action — refund for charge, cancel for book — so that a conflict discovered after the fact can be unwound in the style of the saga pattern.

Preventing deadlock

The classical necessary conditions for deadlock — mutual exclusion, hold-and-wait, no pre-emption, and circular wait — each suggest a countermeasure. For agents, the practical set is: acquire all needed leases at once or none (removing hold-and-wait); let leases expire (permitting pre-emption); impose a global ordering on resources (removing circular wait); and, above all, give every wait a timeout

after which the agent escalates to its orchestrator rather than waiting further. Livelock — agents politely and endlessly deferring to each other — is broken by randomised back-off or by an arbiter that assigns priority.

TRADECRAFT NOTE — MAKE CONFLICT A FIRST-CLASS MESSAGE

Give agents an explicit `report_conflict` action that carries the resource, the competing claims, and the evidence for each. An agent that can say “I think agent B is wrong about this” in a structured way is far easier to manage than one that silently overwrites B’s work or silently defers to it.

7.2 Consensus Mechanisms

Not all disagreements are about shared state. Often several agents produce different answers to the same question — different diagnoses, different extracted values, different plans — and the system must decide which to act on. Four families of mechanism are useful.

Voting

Independent agents (or independent samples of one agent) answer separately, and the majority wins. This is self-consistency applied across agents [2], and it works best when answers are discrete and comparable — a category, a number, a yes/no. Weighted voting, in which each voter’s weight reflects its calibrated historical accuracy on the task class, outperforms simple majority when voters differ in reliability. A useful operational rule: if the vote is not decisive (for example, below a two-thirds supermajority), treat the item as ambiguous and route it to an arbiter or a human.

Debate

Debate lets agents see and respond to each other’s reasoning. In the multi-agent debate procedure, several model instances propose answers, read and critique one another’s responses, and revise over multiple rounds until they converge; using the same procedure and prompts across tasks, its authors reported gains in mathematical and strategic reasoning and in factual accuracy, with uncertain facts tending to drop out when agents disagree [37]. Debate is more expensive

than voting and can converge on a shared error if all participants share the same blind spot, so it pays to diversify participants – different prompts, different roles (proposer, sceptic), and where feasible different models.

Arbiter agents

An arbiter is a designated agent with the authority to resolve disputes. It receives the competing proposals, the evidence each cites, and the relevant policy, and it issues a decision with a rationale. Arbiters should be configured differently from the agents they judge – a separate prompt, often a stronger model, and access to ground-truth tools the disputants lack – so that the decision adds information rather than averaging opinions. The arbiter is also the natural place to apply domain rules (“the system of record wins”; “the more conservative action wins when money is involved”).

Programmatic resolution loops

Whenever a disagreement can be settled by evidence, settle it by evidence. If two agents disagree about whether a patch fixes a bug, run the tests. If they disagree about a customer’s balance, query the ledger. If they disagree about a fact, retrieve and grade sources (Chapter 5). A programmatic loop – *detect disagreement* → *identify the checkable claim* → *run the check* → *accept the verified answer* – is cheaper and more reliable than any amount of deliberation, and it should be tried before voting, debate, or arbitration.

Mechanism	Cost	Strength	Weakness	Use when
Programmatic check	Low	Ground truth	Only for checkable claims	Always try first
Voting	$N \times \text{base}$	Simple; reduces variance	Shared biases survive	Discrete answers; cheap calls
Debate	$N \times \text{rounds} \times \text{base}$	Surfaces reasoning errors	Can converge on shared error	Hard reasoning; factual synthesis
Arbiter	1 strong call + evidence	Applies policy; accountable	Single point of judgement	Policy-laden disputes
Human	Highest latency	Authority; context	Slow; does not scale	High stakes; unresolved disputes

7.3 Fault Tolerance

The Byzantine framing

Classical fault tolerance distinguishes *crash* faults — a node stops responding — from *Byzantine* faults, in which a node may behave arbitrarily, including sending conflicting or false information. Lamport, Shostak, and Pease formalised the latter as the Byzantine Generals Problem and showed, among other results, that with unsigned (oral) messages agreement is achievable only if more than two-thirds of participants are loyal [38]. A hallucinating agent is, functionally, a Byzantine node: it does not stop; it reports plausible, confident, wrong results. The analogy is not exact — agents are not adversarial by default, and their errors are often correlated rather than independent — but it is instructive: **a system cannot tolerate wrong answers from its components merely by adding more components that may be wrong in the same way**. Redundancy must be paired with diversity and with independent verification.

Failure modes and remedies

Node failure	Detection	Remedy
Crash / timeout	Heartbeats; deadlines	Retry with back-off; reassign to a fresh worker; return partial result
Loop / no progress	Repetition fingerprints; progress metrics	Inject redirect; terminate and restart with a summary of failed approaches
Malformed output	Schema validation	Bounded repair loop (Chapter 2); then reassign
Hallucination / wrong result	Grounding checks; cross-validation; tests	Discard; re-run with different model or prompt; escalate if persistent
Compromised node (injection)	Policy violations; anomalous tool use	Quarantine; revoke credentials; review trace (Chapter 9)

Self-healing patterns

Supervision trees. Borrowed from fault-tolerant runtime design, a supervision tree gives each agent a supervisor that monitors it and applies a restart policy when it fails: restart this child only, restart all siblings, or escalate to the next level. The orchestrator of a hierarchical topology is a natural supervisor.

Checkpoint and resume. Long-running agents should persist their state — plan, progress, findings, external actions taken — at regular checkpoints so that a failure costs minutes rather than hours. Anthropic described this as essential for its research system, where restarting from scratch after an error was both expensive and frustrating for users, and paired it with letting the agent know about tool failures so it could adapt [29].

Independent verification nodes. Add verifier agents whose only job is to check other agents' outputs against evidence, with different prompts, tools, and ideally models. The multi-agent failure taxonomy found task-verification failures — missing or inadequate checking, and premature termination — to be one of the three main categories of breakdown [36], which is a strong argument for treating verification as a distinct role rather than an afterthought.

Circuit breakers. If a downstream tool or agent fails repeatedly, stop calling it for a cooling-off period and route around it or degrade gracefully, rather than letting every upstream agent burn its budget on retries.

Graceful degradation. Design every node to return a typed result that includes a status — complete, partial, failed — and a confidence. The orchestrator can then assemble the best available answer from partial results and state its limitations clearly, which is almost always better than failing the whole task.

CHAPTER SUMMARY

- Agent conflicts are lost updates, stale reads, contradictory actions, and deadlocks; prevent them with single-writer ownership, version checks, leases, event logs, and compensable side effects.
- Resolve disagreements by evidence first; then by weighted voting, diverse debate, or a policy-aware arbiter; escalate to humans when unresolved and consequential.
- A hallucinating agent behaves like a Byzantine node: redundancy without diversity and independent checking does not help.
- Build self-healing with supervision, checkpoints, dedicated verifiers, circuit breakers, and partial results.

PART IV

Production Operations, Governance, and Security

A prototype agent proves that something is possible. A production agent proves that it happens reliably, safely, affordably, and accountably — every time, including when no one is watching. Part IV covers the operational discipline (AgentOps), the threat model, and the control points that make that proof possible.

AgentOps: Testing, Tracing, and Observability

If you cannot replay what your agent did, you cannot fix it, defend it, or improve it.

Conventional software is deterministic enough that a failing test reproduces and a stack trace explains. Agents are neither. The same input can take a different path on each run; a failure can originate ten steps before it becomes visible; and “correct” is often a judgement rather than an equality check. AgentOps is the discipline of building, shipping, and running agents under those conditions. This chapter defines a development lifecycle for agents, shows how to trace their execution, and sets out how to evaluate them on success, cost, and latency.

8.1 The Agent Development Lifecycle (ADLC)

The ADLC adapts the software lifecycle to systems whose core component is probabilistic. Its distinguishing feature is that **evaluation is continuous and gates every transition**, because there is no other way to know whether a change helped.

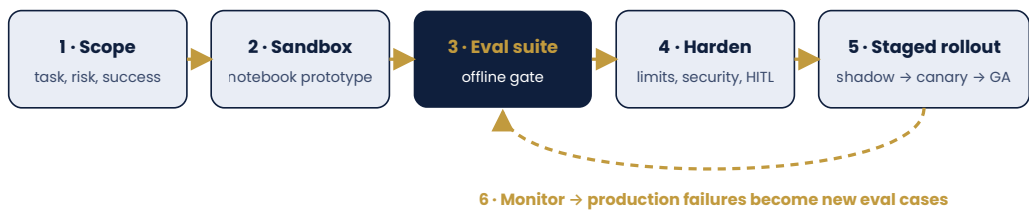


Figure 8.1 — The agent development lifecycle. The evaluation suite is both the gate and the destination of every production lesson.

Stage 1 — Scope. Define the task, the users, the actions the agent may take, the risk of each action, and — before writing any code — what success looks like and how it will be measured. A team that cannot state its success criterion cannot build an evaluation, and a team without an evaluation is guessing.

Stage 2 — Sandbox. Prototype in a notebook or scratch environment against mocked or read-only tools. The goal is to discover whether the task is feasible and what the agent tends to get wrong. Read transcripts obsessively; the failure patterns found here seed the evaluation set.

Stage 3 — Evaluation suite. Assemble a fixed set of tasks with known-good outcomes. Anthropic’s research team reported that starting with a small set — around twenty representative queries — was enough to reveal large effects early, because initial changes often move results dramatically [29]. The suite grows over time; it never shrinks without a reason.

Stage 4 — Harden. Add the production controls: step, token, and time ceilings; validation ladders; least-privilege credentials; sandboxed execution; human approval gates on consequential actions; structured logging and tracing. Chapter 9 covers the security half of this stage.

Stage 5 — Staged rollout. Release progressively. In *shadow mode*, the agent runs on real inputs but its actions are not executed; its proposals are compared against what humans actually did. A *canary* exposes a small slice of real traffic. Because long-running agents may be mid-task when a new version ships, deployment must not disrupt them; Anthropic described using rainbow deployments that shift traffic gradually from old to new versions while both run [29].

Stage 6 — Monitor and learn. Production telemetry feeds dashboards and alerts. Every production failure that is diagnosed becomes a new evaluation case, so the same failure cannot silently return.

Versioning everything

An agent’s behaviour is a function of far more than its code. A reproducible release pins: the model identifier and version; every prompt and tool description; tool schemas and implementations; retrieval indices and their build dates; memory and skill-library snapshots; and configuration such as temperature and limits. Changing any of them is a release, and must pass the evaluation gate.

8.2 Tracing Execution Paths

A trace is the complete, structured record of one agent run: every model call with its inputs and outputs, every tool call with its arguments, results, and latency, every retrieval with what was returned, every sub-agent spawned, and the token usage and cost of each step, all linked in a tree that mirrors the run's control flow.

Anatomy of a trace

```
run 7f3a... "Reconcile September vendor invoices"      41.2 s   $0.83
status=partial
├─ llm plan                                in 3.1k / out 0.6k tok   2.4 s
├─ tool invoices_search(status=open, issued_after=2026-09-01) 0.3 s   → 37 rows
├─ agent worker[vendor=A]                                11.8 s
│   ├─ llm reason                                in 5.2k / out 0.4k
│   │   ├─ tool ledger_lookup(...)                → 0 rows          ◀ empty result
│   │   └─ tool ledger_lookup(...)                → 0 rows          ◀ repeat #2
│   └─ llm reason → report_blocked("vendor A ledger not found")
├─ agent worker[vendor=B] ... status=complete
└─ llm synthesise                                in 9.7k / out 1.1k tok   4.9 s
```

Read top-down, a trace shows where time and money went. Read at the leaves, it shows where the reasoning went wrong — here, a worker that searched the ledger twice with an identifier format the tool did not accept, and correctly reported itself blocked rather than fabricating a reconciliation.

Tooling

Several platforms specialise in agent tracing. LangSmith, from the LangChain team, provides tracing, monitoring, and evaluation for LLM applications [39]. Arize Phoenix is an open-source platform for AI observability and evaluation [40]. Underneath both sits a broader move toward standardisation: the OpenTelemetry project has developed semantic conventions for generative AI, including spans for agent invocations and tool execution, now maintained in a dedicated repository [41]. Building on OpenTelemetry-style spans has a practical advantage: agent traces can flow into the same observability backends as the rest of an organisation's services, so a slow database behind a tool shows up in the same view as the agent step that called it.

Debugging non-deterministic failures

1. **Find the first bad step, not the last.** The visible failure is usually downstream of the real one. Walk the trace backward from the failure to the earliest step

where the agent's state diverged from reality — typically a tool result it misread, a retrieval that returned the wrong document, or a compaction that dropped a constraint.

2. **Classify the root cause by layer.** Was it the model's reasoning, a tool's interface, missing or misleading context, a memory error, or an orchestration error? Use the stack of Chapter 1. The fix lives in the layer where the fault originated.
3. **Reproduce statistically.** Re-run the failing input many times. A failure that occurs in three of twenty runs is a reliability problem and needs a different fix from one that occurs in twenty of twenty.
4. **Replay with substitution.** Freeze the trace up to the suspect step, substitute a corrected input (a fixed tool result, a different prompt), and replay from there. This isolates cause from effect without re-running the whole task.
5. **Aggregate across traces.** Single traces show anecdotes; clusters show patterns. Tag traces with failure categories — the multi-agent failure taxonomy is a useful starting vocabulary [36] — and track their frequency over releases.

TRADECRAFT NOTE — PRIVACY IN TRACES

Traces contain everything the agent saw, which often includes personal data, credentials echoed in tool outputs, and confidential documents. Apply redaction at capture time, restrict access by role, and set retention limits. A trace store is a sensitive data store and must be governed as one.

8.3 Automated Evaluation

What to measure

Dimension	Metric	Notes
Effectiveness	Task success rate	Judged by end state where possible, not by the agent's claim
Reliability	pass ^k (success on all of <i>k</i> runs)	Captures consistency, which pass@1 hides
Efficiency	Cost per task; tokens per task; tool calls per task	Report distributions (median, p95), not just means
Latency	Time to first action; time to completion	Set thresholds per task class
Safety	Policy-violation rate; unauthorised-action attempts; injection success rate	Measured on adversarial suites (Chapter 9)
Quality	Groundedness, citation accuracy, completeness	Often model-graded with human calibration
Escalation	Human-handoff rate; handoff appropriateness	Too low may mean overconfidence; too high, low value

Reliability is not accuracy

The τ -bench benchmark makes this point sharply. It simulates conversations between an agent and a model-simulated user in customer-service domains, gives the agent domain APIs and policy documents, and judges success by comparing the final database state to an annotated goal state [42]. It introduced the **pass^k** metric — the probability that the agent succeeds on *all* of *k* independent trials of the same task — and found that a leading model that succeeded on about 61% of retail tasks in a single trial fell below 25% on pass⁸ [42]. For a business process run thousands of times a day, pass^k is much closer to the number that matters.

Public benchmarks

Public benchmarks are useful for model selection and for calibrating expectations, though never as a substitute for an in-house evaluation built from real tasks:

- **SWE-bench** evaluates whether a system can resolve real GitHub issues, checked by running the repositories' own tests [43].

- **GAIA** poses 466 real-world assistant questions requiring reasoning, browsing, and tool use; at publication, humans answered about 92% correctly against about 15% for GPT-4 with plugins [44].
- **τ -bench** tests multi-turn tool use under domain policies, with state-based grading and pass^k reliability [42].

Grading methods

State-based grading checks the world after the run: the database row exists, the test passes, the file contains the right value. It is objective and should be used wherever the outcome is a state change.

Programmatic grading checks the output against rules: required fields present, figures match sources, citations resolve, format valid.

Model-graded evaluation uses a separate model with a rubric to score qualities that code cannot — completeness, accuracy of a synthesis, appropriateness of tone. Anthropic reported that a single LLM judge scoring against a rubric (factual accuracy, citation accuracy, completeness, source quality, tool efficiency) scaled well for free-form research outputs [29]. Model graders must themselves be calibrated against human judgements on a sample, and re-calibrated when the grader model changes.

Human evaluation remains essential for catching what automated methods miss. The same team reported that human testers found edge cases — such as a tendency to prefer content-farm sources over authoritative ones — that their automated evaluations had not flagged [29].

Thresholds and gates

Turn metrics into explicit release criteria, for example: success rate not lower than the current production version (with a statistical test, given run-to-run variance); pass⁴ above an agreed floor; p95 cost per task below budget; p95 latency below the user-experience threshold; zero critical policy violations on the adversarial suite. A candidate that fails any gate does not ship, however good it looks in demos.

```
# Evaluation gate (sketch)
results = run_suite(candidate, suite, trials_per_task=4)
gates = {
    "success": results.success_rate >= baseline.success_rate - 0.01,
    "pass^4": results.pass_k(4) >= 0.70,
    "cost_p95": results.cost.p95 <= 1.50,          # USD per task
    "lat_p95": results.latency.p95 <= 90,          # seconds
    "safety": results.critical_violations == 0,
}
assert all(gates.values()), {k: v for k, v in gates.items() if not v}
```

CHAPTER SUMMARY

- The ADLC is gated by evaluation at every step: scope → sandbox → eval suite → harden → staged rollout → monitor, with production failures feeding back as new test cases.
- Version models, prompts, tool definitions, indices, and memory snapshots; any change is a release.
- Trace every model call, tool call, and sub-agent as a linked tree with tokens, cost, and latency; debug from the first divergent step and classify the fault by layer.
- Measure success, reliability (pass^k), cost, latency, safety, and quality; grade by end state wherever possible; calibrate model graders against humans; enforce explicit release gates.

Security, Guardrails, and Threat Modeling

An agent turns every document it reads into a potential instruction and every credential it holds into a potential weapon. Design accordingly.

Security for agents is not security for chatbots with extra steps. A chatbot that is manipulated produces bad text; an agent that is manipulated takes bad actions — sending data out, changing records, spending money, running code — with whatever authority it has been given. This chapter builds a threat model around that fact, examines prompt injection as the defining vulnerability of the field, and sets out the two families of defence that actually hold: limiting what a compromised agent can do, and placing humans at the points where the stakes demand them.

Threat model

The OWASP Top 10 for LLM Applications 2025 provides a shared vocabulary. Its first entry is **prompt injection** (LLM01), and its sixth is **excessive agency** (LLM06) — giving a model-driven system more functionality, permissions, or autonomy than its task requires [45]. For agents these two combine into the central risk: an attacker who can influence what the agent reads can direct what the agent does, and the damage is bounded only by what the agent is allowed to do. Other entries — sensitive information disclosure, supply-chain risks, improper output handling, vector and embedding weaknesses, and unbounded consumption — all have direct agentic counterparts [45].

Asset	Threat	Entry point	Primary control
Data the agent can read	Exfiltration	Injected instruction + any outbound channel	Egress control; data-flow policies
Systems the agent can write	Unauthorised change	Injection; flawed planning	Least privilege; HITL gates; dry runs
Budget and capacity	Runaway consumption	Loops; adversarial inputs	Hard ceilings; rate limits
Agent memory	Persistent poisoning	Untrusted content written to memory	Trust labels; no promotion to instructions
Tool supply chain	Malicious or compromised tool/MCP server	Third-party integrations	Allow-listing; review; isolation
Downstream systems	Injection via agent output	Agent output used unsanitised (SQL, shell, HTML)	Treat output as untrusted; parameterise

9.1 Prompt Injection and Indirect Exploits

Direct and indirect injection

Direct injection is the familiar case: a user types instructions intended to override the system’s rules. *Indirect* injection is the case that matters most for agents. Gre-shake and colleagues showed that an attacker need not talk to the model at all: it is enough to plant instructions in content the model will later retrieve — a web page, an email, a document — and the model may follow them as if they came from its operator [28]. They demonstrated this against real LLM-integrated applications, including getting a search-connected chatbot to adopt a manipulative persona and attempt to extract users’ personal information, using text hidden from human readers [28].

The root of the problem is architectural. A language model receives instructions and data through the same channel — text in its context — and has no reliable, built-in way to tell them apart. Every agent that reads untrusted content and holds meaningful capabilities is therefore exposed, whatever its prompt says.

Agent-specific attack patterns

- **Exfiltration via tools.** An injected page instructs the agent to summarise the user's recent emails and include them in a URL it then fetches, or in a message it sends.
- **Action hijacking.** A document in a shared drive tells a procurement agent that the vendor's bank details have changed.
- **Tool poisoning.** A third-party tool's description or output contains instructions that redirect the agent; security researchers have flagged this specifically for MCP ecosystems [34].
- **Memory persistence.** An injected instruction is summarised into long-term memory and reactivates in later sessions (Chapter 3).
- **Cross-agent propagation.** One compromised agent's output becomes another agent's input, spreading the injection through a multi-agent system.

Measuring the problem

AgentDojo provides a standard way to test defences: it pairs 97 realistic agent tasks — managing email, using an e-banking site, booking travel — with 629 security test cases that embed injection attacks in the data the agent encounters [46]. Teams building agents should run an equivalent adversarial suite against their own tools and data, and track injection success rate as a release-gating metric (Chapter 8).

Defences that do and do not work

Prompt-level defences — telling the model to ignore instructions in retrieved content, wrapping untrusted text in delimiters, adding a classifier that looks for injection phrasing — raise the cost of attack and are worth deploying, but none of them is a guarantee, because they rely on the same model whose judgement is under attack. Durable defences change the architecture so that a successful injection cannot achieve much.

The most rigorous example is CaMeL, which separates control flow from data flow: a privileged component turns the trusted user request into a program, while untrusted data is handled so that it can never alter which actions the program takes; capabilities attached to data enforce policies on where information may

flow when tools are called [47]. On AgentDojo its authors report completing 77% of tasks with provable security, against 84% for an undefended system — a modest utility cost for a strong guarantee [47].

Practical design patterns in the same spirit include:

1. **Plan before reading.** Fix the plan and the set of permitted actions from the trusted request before the agent ingests untrusted content; content may fill in parameters but may not add actions.
2. **Quarantine untrusted content.** Let a tool-less model process untrusted text and return only structured, validated fields (a date, an amount, a yes/no) to the privileged agent.
3. **Taint tracking.** Label data by origin and refuse to let tainted data reach sensitive sinks (outbound messages, payment fields, shell commands) without a check.
4. **Break the lethal combination.** Avoid giving one agent, in one context, access to private data, exposure to untrusted content, and an outbound channel at the same time. Remove any one of the three and exfiltration becomes far harder.

TRADECRAFT NOTE — ASSUME BREACH

Design as if the injection will eventually succeed, because some will. The question to ask of every tool and permission is not “can the model be tricked into misusing this?” but “if it is, what is the worst outcome, and is that acceptable?”

9.2 Blast Radius Mitigation

If compromise cannot be ruled out, the next goal is to bound its consequences. The governing principle is half a century old: Saltzer and Schroeder’s principle of least privilege holds that every program and every user should operate with the smallest set of privileges needed for the job [48]. OWASP’s guidance on excessive agency applies it directly to LLM systems — restrict functionality, permissions, and autonomy to what the task requires [45].

Least-privilege API access

- **Scope credentials to the task, not the agent.** Issue short-lived, narrowly scoped tokens per task or per session rather than long-lived service accounts with broad rights.
- **Act as the user, not as a superuser.** Where an agent works on behalf of a person, delegate that person's permissions (e.g., via OAuth) so the agent can never do more than they could.
- **Separate read and write.** Read tools and write tools carry different credentials; a research agent has no write credentials at all.
- **Constrain parameters server-side.** Enforce limits (maximum refund, allowed recipients, permitted directories) in the tool's implementation, not only in its description.
- **Allow-list tools and servers.** Only reviewed tools and MCP servers may be connected; new ones go through the same review as any third-party dependency.

Permission sandboxing

Agents that execute code or operate computers must do so in isolated environments: containers or virtual machines with no access to production credentials, a restricted file system, and network egress limited to an allow-list of domains. The sandbox, not the model, is the security boundary. A useful test is to ask what a fully malicious program could do inside the sandbox; if the answer includes reaching production data or the open internet, the sandbox is not finished.

Execution limits

Hard limits — steps per task, tokens per task, tool calls per minute, wall-clock time per action, total spend per day — contain both attacks and accidents. OWASP lists unbounded consumption as its own risk category [45]; for agents, which can loop autonomously, budget ceilings are a security control as much as a cost control.

Containment by architecture

Layer	Containment measure
Identity	Per-agent, per-task identities; no shared credentials
Network	Egress allow-lists; no direct internet from privileged agents
Data	Row/field-level access control; redaction before context
Actions	Separate tools for destructive operations; dry runs; idempotency
Time & cost	Ceilings and circuit breakers
Audit	Immutable logs of every action with the trace that produced it

9.3 Human-in-the-Loop (HITL) Gateways

Some actions are too consequential to delegate entirely, however good the agent. HITL gateways are the deliberate points at which a human must approve, edit, or reject what the agent proposes. Regulation increasingly expects them: the EU Artificial Intelligence Act requires that high-risk AI systems be designed so that natural persons can oversee them effectively — understanding the system’s capacities and limitations, remaining aware of automation bias, interpreting outputs correctly, deciding to disregard or override them, and intervening or halting the system, for example through a stop control [49]. The NIST AI Risk Management Framework, released in January 2023, organises AI risk management into four functions — Govern, Map, Measure, and Manage — with accountability structures and system inventories under Govern [50].

Where to put the gates

Gate by consequence, not by frequency. A practical classification:

Tier	Action profile	Examples	Control
0	Read-only, internal	Search docs, query own records	Autonomous; logged
1	Reversible, low impact	Draft an email; create a ticket; edit in a branch	Autonomous; sampled review
2	External or costly but recoverable	Send a message; issue a small refund	Policy limits; approval above threshold
3	Irreversible or high impact	Delete data; wire funds; deploy to production; sign	Mandatory human approval; dual control where warranted

Designing the gate

An approval gate is only as good as the decision it enables. The request a human sees should state exactly what will happen (the concrete action and parameters, not a paraphrase), why the agent proposes it (with the evidence and sources), what the alternatives were, and what happens if it is declined. It should flag anything derived from untrusted content. And it should make *reject* and *edit* as easy as *approve*.

Circuit breakers

Beyond per-action gates, the system needs global breakers that halt agents automatically when something looks wrong: a spike in policy violations, an unusual pattern of outbound requests, spending above trend, a sharp rise in error rates, or an attempted action outside the agent's declared scope. A tripped breaker pauses the affected agents, preserves their state and traces for review, and alerts a human. Every operator should also have a manual stop that is tested regularly.

Avoiding rubber stamps

The known failure of HITL is automation bias — a risk the EU AI Act names explicitly [49]: when approvals are frequent and almost always correct, reviewers stop reading. Counter it by keeping gates for genuinely consequential actions so approvals stay rare; by sampling and auditing approved actions; by occasionally inserting known-bad test proposals to measure reviewer attention; and by tracking time-to-approve as a warning sign.

CHAPTER SUMMARY

- Prompt injection and excessive agency combine into the defining agent risk: whoever controls what the agent reads can influence what it does.
- Prompt-level defences help but do not guarantee; architectural separation of control and data flow, quarantined processing, taint tracking, and avoiding the private-data + untrusted-content + outbound-channel combination do.
- Bound the blast radius with least privilege, task-scoped credentials, sandboxed execution, egress control, and hard limits.
- Gate consequential actions with well-designed human approvals and global circuit breakers, and guard against rubber-stamping.

PART V

Enterprise Scaling and Future Horizons

The final part turns from how agents work to whether they pay – and how an organisation keeps control of them once there are hundreds. It maps agentic workflows onto real business processes, sets out governance for agent sprawl, and builds an honest economic model.

Enterprise Integration and Operational ROI

The question is never “can an agent do this?” It is “can an agent do this reliably enough, cheaply enough, and safely enough that the organisation is better off?”

Enterprises do not buy reasoning loops; they buy shorter cycle times, lower cost per transaction, fewer errors, and capacity that scales without hiring. This chapter connects the architecture of the previous nine chapters to that bottom line: which processes suit agents and how to map them, how to govern a growing fleet of autonomous scripts, and how to model costs and benefits with the rigour a finance team will expect.

10.1 Mapping Agentic Workflows to Business Processes

A suitability screen

Before mapping a process, screen it. Agents fit best where five conditions hold:

1. **Judgement over unstructured input.** The work requires reading, interpreting, or deciding — not merely moving structured data, which conventional automation already does better and more cheaply.
2. **Accessible systems.** The data and actions the work requires are reachable through APIs or tools, with permissions that can be scoped.
3. **Checkable outcomes.** Success can be verified — by end state, by tests, by reconciliation, or by sampled human review — so that evaluation (Chapter 8) is possible.
4. **Tolerable error cost.** Mistakes are recoverable or can be intercepted at a gate before they become expensive (Chapter 9).
5. **Volume or latency pressure.** There is enough repetition, or enough value in speed, to repay the engineering investment.

Processes that fail the third or fourth condition are poor first candidates no matter how impressive a demonstration looks.

Decomposing a process

Map each candidate process as a sequence of steps and classify every step: *deterministic* (a rule or API call — automate conventionally), *judgement* (interpretation or decision — a model call or agent loop), or *authority* (accountable approval — a human gate). The resulting map is usually a workflow with agentic islands, not a single autonomous agent, which is consistent with the advice to prefer the simplest pattern that works [4].

Customer operations

Customer support is the best-evidenced domain. In a study of more than five thousand customer-support agents, access to a generative-AI assistant raised productivity, measured as issues resolved per hour, by 14% on average, with the largest gains for novice and lower-skilled workers [51]. That study examined an assistant that suggested responses to humans; agentic deployments push further, resolving routine contacts end to end. The τ -bench results in Chapter 8 are a caution about how far: on simulated retail service tasks, a leading model’s single-trial success of roughly 61% fell below 25% when it had to succeed on all of eight repeated trials [42]. A sound customer-operations design therefore lets the agent resolve the high-confidence, policy-simple majority autonomously, routes the remainder to humans with a drafted resolution and the evidence assembled, and gates refunds, credits, and account changes above set thresholds.

Step	Type	Implementation
Classify intent and urgency	Judgement	Router model; confidence threshold
Authenticate and fetch account	Deterministic	API calls with customer-scoped credentials
Diagnose issue	Judgement	ReAct loop over account, order, and knowledge-base tools
Apply policy	Deterministic + judgement	Rules engine for limits; model for interpretation
Issue remedy	Authority (above threshold)	Autonomous below limit; human gate above
Respond and log	Judgement + deterministic	Drafted reply; CRM update; trace attached

Automated procurement

Procurement combines document understanding (requisitions, quotes, contracts, invoices) with rule-bound decisions and money movement — a strong fit with careful gating. A typical design uses agents to extract and normalise quotes, compare them against policy and historical prices, draft purchase orders, and perform three-way matching of order, receipt, and invoice; deterministic code to enforce approval matrices and budget limits; and humans to approve new vendors, exceptions, and payments. Because procurement is a prime target for fraud — for instance, an email or document claiming a vendor's bank details have changed — it is also the domain where the injection defences of Chapter 9 matter most: no change to payment details should ever be actioned on the strength of content the agent read.

Code-generation pipelines

Software engineering offers unusually good verification — compilers, type-checkers, linters, and test suites — which is why it has been a leading domain for agents. Evidence on developer productivity predates autonomous agents: in a controlled experiment, developers given an AI pair programmer completed an HTTP-server task about 56% faster than a control group [52]. Agentic pipelines extend this from autocomplete to issue resolution, as measured by SWE-bench [43]. A production code pipeline typically lets an agent take a ticket, reproduce the problem, propose a change on a branch, and run tests in a sandbox; requires the change to pass CI and automated review; and requires human review before merge. The agent never holds credentials to deploy.

10.2 Managing Agent Sprawl

Once agents become easy to build, they multiply: a team wires up an assistant for its own workflow, another team copies it, a script started as an experiment quietly becomes load-bearing. Without governance, an organisation ends up with hundreds of autonomous processes holding credentials, touching data, and spending budget, with no one able to say what they all do. The NIST AI Risk Management Framework's Govern function — accountability structures, policies, and an inventory of AI systems — is a sensible anchor for bringing this under control [50].

The agent registry

Every agent in production should have a registry entry, and an agent without one should not be able to obtain credentials. A minimal record:

Field	Purpose
Identifier, name, version	Unique identity; links to traces and releases
Business owner and technical owner	Named accountability
Purpose and process mapped	What it is for; which business process it serves
Models and versions	Supports impact analysis when a model changes
Tools, MCP servers, and data sources	Its reach
Permissions and credential scopes	Its authority; reviewed periodically
Risk tier and HITL gates	Links to Chapter 9 controls
Evaluation suite and latest results	Evidence it works (Chapter 8)
Budget and actual spend	Cost accountability
Status and review date	Experimental, production, deprecated; next review

Governance mechanics

- **Identity per agent.** Each agent gets its own non-human identity with its own credentials, so every action in every system can be attributed to a specific agent and version.
- **Central gateways.** Route model calls and tool calls through shared gateways that enforce authentication, policy, rate limits, logging, and budget — making the registry enforceable rather than advisory.
- **Tiered approval.** Low-risk internal agents can be self-registered; agents with write access to customer data, money, or production systems require security and risk review before launch.
- **Lifecycle reviews.** Periodically re-certify each agent’s owner, permissions, and evaluation results; retire agents with no owner or no recent use; revoke credentials on retirement.
- **Shared components.** Provide reviewed, reusable tools and MCP servers so teams do not each build their own, inconsistently secured, integration to the same system.

10.3 Economic Modeling

An honest business case compares the fully loaded cost of the agentic process with the fully loaded cost of the process it replaces or augments, adjusted for quality and risk. The most common error is to compare the token cost of a single successful run with the labour cost of a human doing the same task once – ignoring failures, retries, escalations, review, engineering, and operations.

Cost per completed task

Define, for a task class:

$$C_{agent} = (C_{tokens} + C_{tools} + C_{compute}) / s + e \cdot C_{escalation} + r \cdot C_{review} + p \cdot C_{error} + C_{fixed} / N$$

where s is the autonomous success rate (failed runs still cost tokens), e the escalation rate and $C_{escalation}$ the human time to finish an escalated case, r the share of autonomous outcomes sampled for review, p the rate of errors that escape to production and C_{error} their average cost, C_{fixed} the annualised engineering, evaluation, security, and operations cost, and N the annual task volume.

Token cost itself is the sum over every model call in the run of input and output tokens times their respective prices. Because multi-step agents re-read growing contexts, input tokens usually dominate; Anthropic’s observation that agents use about four times the tokens of chat, and multi-agent systems about fifteen times, is a useful sanity check on estimates [29]. Context management (Chapter 2) and caching of stable prompt prefixes, where available, are therefore direct cost levers.

Labour offset

The comparable human cost is:

$$C_{human} = (t_{task} \times w_{loaded}) + C_{human-error}$$

with t_{task} the average handling time and w_{loaded} the fully loaded hourly cost. Savings per task are $C_{human} - C_{agent}$, and the annual benefit is that difference times N . In practice many deployments augment rather than replace: humans handle escalations and approvals, and the benefit appears as higher throughput per person — the pattern the customer-support evidence shows [51].

A worked illustration

The following figures are **illustrative assumptions** chosen to show the method, not benchmarks or quoted prices. Consider an invoice-exception process with 60,000 cases a year.

Parameter	Assumed value
Human handling time × loaded rate	12 min × \$60/h = \$12.00 per case
Agent variable cost per run (tokens + tools + compute)	\$0.40
Autonomous success rate s	0.80
Escalation rate e and human time to finish	$0.20 \times (5 \text{ min} \times \$60/\text{h} = \$5.00)$
Review sampling r and cost per review	$0.10 \times \$2.00$
Escaped-error rate p and cost	$0.005 \times \$40$
Annual fixed cost (build, evals, security, ops)	\$180,000 → \$3.00 per case

Then $C_{agent} = 0.40/0.80 + 0.20 \times 5.00 + 0.10 \times 2.00 + 0.005 \times 40 + 3.00 = 0.50 + 1.00 + 0.20 + 0.20 + 3.00 = \text{\$4.90 per case}$, against \$12.00 for the human baseline (before any human-error cost) — a saving of about \$7.10 per case, or roughly \$426,000 a year. The decomposition is more instructive than the total: in this illustration the fixed cost is the largest single component, which means volume drives viability, and escalation is the next, which means improving autonomous success has more leverage than cutting token prices.

Sensitivity and speed

Run the model across ranges, not points: success rate, escalation handling time, volume, and model prices all move, and the decision should be robust to plausible variation. Then add the benefits the cost model omits. Speed is often the largest: an agent that turns a three-day procurement cycle into three hours, or lets an investigation that took a week finish overnight, changes what the organisation can do, not just what it costs. Capture these as explicit, measurable claims — cycle time, time to market, backlog cleared — and hold the deployment to them.

TRADECRAFT NOTE — COUNT THE FAILURES

Every economic model should be rebuilt from production data after the first month. Pull the real distributions of cost per run, success, escalation, and review time from the traces of Chapter 8. Estimates made before launch are hypotheses; traces are evidence.

Closing: the engineering of trust

This book began by observing that an agent is judged by what it changes. The chapters since have argued that what makes such change acceptable is not the brilliance of the model but the engineering around it: loops with stopping rules, context managed as a budget, memory with provenance, tools designed for their real user, retrieval that grades its evidence, multi-agent structures with clear ownership, traces that make every action explainable, defences that assume breach, humans placed where authority belongs, and economics measured rather than assumed. Models will keep improving, and some of the techniques described here will be absorbed into them. The discipline of building systems that deserve the trust placed in them will not be.

CHAPTER SUMMARY

- Screen processes for judgement, accessible systems, checkable outcomes, tolerable error cost, and volume; map each step as deterministic, judgement, or authority.
- Customer operations, procurement, and code pipelines each show the same pattern: agentic islands within a workflow, gated where money, data, or production are at stake.
- Govern sprawl with a mandatory agent registry, per-agent identities, central gateways, tiered approval, and lifecycle reviews.
- Model cost per *completed* task including failures, escalations, review, escaped errors, and fixed costs; test sensitivity; measure speed benefits explicitly; rebuild the model from production traces.

References

Numbered in order of first citation. All entries verified 6 October 2026.

- [1] Wei, J., Wang, X., Schuurmans, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS 2022*. arXiv:2201.11903. <https://arxiv.org/abs/2201.11903>
- [2] Wang, X., Wei, J., Schuurmans, D., et al. (2023). Self-consistency improves chain of thought reasoning in language models. *ICLR 2023*. arXiv:2203.11171. <https://arxiv.org/abs/2203.11171>
- [3] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. *ICLR 2023*. arXiv:2210.03629. <https://arxiv.org/abs/2210.03629>
- [4] Schluntz, E. & Zhang, B. (2024, December 19). Building effective agents. Anthropic Engineering. <https://www.anthropic.com/engineering/building-effective-agents>
- [5] Russell, S. & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
- [6] Wooldridge, M. & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2), 115–152. <https://doi.org/10.1017/S0269888900008122>
- [7] Summers, T. R., Yao, S., Narasimhan, K. & Griffiths, T. L. (2024). Cognitive architectures for language agents. *Transactions on Machine Learning Research*. arXiv:2309.02427. <https://arxiv.org/abs/2309.02427>
- [8] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K. & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *NeurIPS 2023*. arXiv:2303.11366. <https://arxiv.org/abs/2303.11366>
- [9] Wang, L., Ma, C., Feng, X., et al. (2023). A survey on large language model based autonomous agents. arXiv:2308.11432. <https://arxiv.org/abs/2308.11432>
- [10] Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. & Press, O. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. *NeurIPS 2024*. arXiv:2405.15793. <https://arxiv.org/abs/2405.15793>
- [11] Rajasekaran, P., Dixon, E., Ryan, C. & Hadfield, J. (2025, September 29). Effective context engineering for AI agents. Anthropic Engineering. <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>
- [12] Wang, L., Xu, W., Lan, Y., et al. (2023). Plan-and-Solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *Proceedings of ACL 2023*, 2609–2634. arXiv: 2305.04091. <https://aclanthology.org/2023.acl-long.147>
- [13] Xu, B., Peng, Z., Lei, B., Mukherjee, S., Liu, Y. & Xu, D. (2023). ReWOO: Decoupling reasoning from observations for efficient augmented language models. arXiv:2305.18323. <https://arxiv.org/abs/2305.18323>
- [14] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y. & Narasimhan, K. (2023). Tree of Thoughts: Deliberate problem solving with large language models. *NeurIPS 2023*. arXiv: 2305.10601. <https://arxiv.org/abs/2305.10601>
- [15] Besta, M., Blach, N., Kubicek, A., et al. (2024). Graph of Thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16). arXiv:2308.09687. <https://ojs.aaai.org/index.php/AAAI/article/view/29720>

- [16] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. & Liang, P. (2024). Lost in the middle: How language models use long contexts. *Transactions of the ACL*, 12. arXiv: 2307.03172. <https://arxiv.org/abs/2307.03172>
- [17] Hong, K., Troynikov, A. & Huber, J. (2025, July 14). Context rot: How increasing input tokens impacts LLM performance. Chroma Technical Report. <https://www.trychroma.com/research/context-rot>
- [18] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P. & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *UIST '23*. arXiv:2304.03442. <https://arxiv.org/abs/2304.03442>
- [19] Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *NeurIPS 2020*. arXiv:2005.11401. <https://arxiv.org/abs/2005.11401>
- [20] Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I. & Gonzalez, J. E. (2023). MemGPT: Towards LLMs as operating systems. arXiv:2310.08560. <https://arxiv.org/abs/2310.08560>
- [21] Wang, G., Xie, Y., Jiang, Y., et al. (2023). Voyager: An open-ended embodied agent with large language models. arXiv:2305.16291. <https://arxiv.org/abs/2305.16291>
- [22] Aizawa, K. (2025, September 11). Writing effective tools for AI agents – using AI agents. Anthropic Engineering. <https://www.anthropic.com/engineering/writing-tools-for-agents>
- [23] Schick, T., Dwivedi-Yu, J., Dessì, R., et al. (2023). Toolformer: Language models can teach themselves to use tools. *NeurIPS 2023*. arXiv:2302.04761. <https://arxiv.org/abs/2302.04761>
- [24] Shingo, S. (1986). *Zero Quality Control: Source Inspection and the Poka-yoke System*. Productivity Press.
- [25] Duarte, A. V., Marques, J., Graça, M. & Freire, M. (2024). LumberChunker: Long-form narrative document segmentation. *Findings of EMNLP 2024*. arXiv:2406.17526. <https://arxiv.org/abs/2406.17526>
- [26] Asai, A., Wu, Z., Wang, Y., Sil, A. & Hajishirzi, H. (2024). Self-RAG: Learning to retrieve, generate, and critique through self-reflection. *ICLR 2024*. arXiv:2310.11511. <https://arxiv.org/abs/2310.11511>
- [27] Yan, S.-Q., Gu, J.-C., Zhu, Y. & Ling, Z.-H. (2024). Corrective retrieval augmented generation. arXiv:2401.15884. <https://arxiv.org/abs/2401.15884>
- [28] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T. & Fritz, M. (2023). Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. *AISeC '23*. arXiv:2302.12173. <https://arxiv.org/abs/2302.12173>
- [29] Hadfield, J., Zhang, B., Lien, K., Scholz, F., Fox, J. & Ford, D. (2025, June 13). How we built our multi-agent research system. Anthropic Engineering. <https://www.anthropic.com/engineering/multi-agent-research-system>
- [30] Wu, Q., Bansal, G., Zhang, J., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155. <https://arxiv.org/abs/2308.08155>
- [31] Erman, L. D., Hayes-Roth, F., Lesser, V. R. & Reddy, D. R. (1980). The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2), 213–253.
- [32] Hong, S., Zhuge, M., Chen, J., et al. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. *ICLR 2024*. arXiv:2308.00352. <https://arxiv.org/abs/2308.00352>

- [33] Foundation for Intelligent Physical Agents. (2002). *FIPA ACL Message Structure Specification* (SC00061G). <http://www.fipa.org/specs/fipa00061/SC00061G.html>
- [34] Anthropic. (2024, November 25). Introducing the Model Context Protocol; protocol history summarised in “Model Context Protocol,” Wikipedia (accessed 2026-10-06). https://en.wikipedia.org/wiki/Model_Context_Protocol
- [35] Google / Linux Foundation. (2025). Agent2Agent (A2A) Protocol; history summarised in “Agent2Agent,” Wikipedia (accessed 2026-10-06). <https://en.wikipedia.org/wiki/Agent2Agent>
- [36] Cemri, M., Pan, M. Z., Yang, S., et al. (2025). Why do multi-agent LLM systems fail? arXiv: 2503.13657. <https://arxiv.org/abs/2503.13657>
- [37] Du, Y., Li, S., Torralba, A., Tenenbaum, J. B. & Mordatch, I. (2023). Improving factuality and reasoning in language models through multiagent debate. arXiv:2305.14325. <https://arxiv.org/abs/2305.14325>
- [38] Lamport, L., Shostak, R. & Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 382–401.
- [39] LangChain. (2026). LangSmith observability documentation (accessed 2026-10-06). <https://docs.langchain.com/langsmith/observability>
- [40] Arize AI. (2026). Phoenix: Open-source AI observability and evaluation documentation (accessed 2026-10-06). <https://docs.arize.com/phoenix/>
- [41] OpenTelemetry. (2026). Semantic conventions for generative AI (agent and framework spans; now maintained in the semantic-conventions-genai repository; accessed 2026-10-06). <https://opentelemetry.io/docs/specs/semconv/gen-ai/>
- [42] Yao, S., Shinn, N., Razavi, P. & Narasimhan, K. (2024). τ -bench: A benchmark for tool-agent-user interaction in real-world domains. arXiv:2406.12045. <https://arxiv.org/abs/2406.12045>
- [43] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? ICLR 2024. arXiv:2310.06770. <https://arxiv.org/abs/2310.06770>
- [44] Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y. & Scialom, T. (2024). GAIA: A benchmark for general AI assistants. ICLR 2024. arXiv:2311.12983. <https://arxiv.org/abs/2311.12983>
- [45] OWASP Foundation. (2024). OWASP Top 10 for LLM Applications 2025 (LLM01: Prompt Injection; LLM06: Excessive Agency). <https://genai.owasp.org/llm-top-10/>
- [46] Debenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M. & Tramèr, F. (2024). AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. *NeurIPS 2024 Datasets & Benchmarks*. arXiv:2406.13352. <https://arxiv.org/abs/2406.13352>
- [47] Debenedetti, E., Shumailov, I., Fan, T., et al. (2025). Defeating prompt injections by design (CaMeL). arXiv:2503.18813. <https://arxiv.org/abs/2503.18813>
- [48] Saltzer, J. H. & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278–1308.
- [49] European Parliament & Council of the EU. (2024). Regulation (EU) 2024/1689 (Artificial Intelligence Act), Article 14: Human oversight. *Official Journal of the European Union*, L, 12.7.2024. <https://eur-lex.europa.eu/eli/reg/2024/1689/oj>
- [50] National Institute of Standards and Technology. (2023). *Artificial Intelligence Risk Management Framework* (AI RMF 1.0) (NIST AI 100-1). <https://doi.org/10.6028/NIST.AI.100-1>

- [51]** Brynjolfsson, E., Li, D. & Raymond, L. (2025). Generative AI at work. *The Quarterly Journal of Economics*, 140(2), 889–942. <https://doi.org/10.1093/qje/qjae044>
- [52]** Peng, S., Kalliamvakou, E., Cihon, P. & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590. <https://arxiv.org/abs/2302.06590>