7. Applications: Agents

Table of contents

1 Overview **2**

2 What Is an Agent **3**
2.1 An Agent Is a Controller, Not a "Talking Model" 3
2.2 The Agent Control Loop: From a Single Answer to an Iterative System 5
2.3 Where Prompts, RAG, Tools, and Skills Sit in the Loop 7

3 System Architecture **8**
3.1 The Core Components of an Agent System 8
3.2 RAG: Connecting External Knowledge to the System 10
3.3 Tool Use: Letting the System Interact with the Real World . . . 13
3.4 State and Memory: Letting Tasks Span More Than One Inference Step 16

4 Engineering **18**
4.1 Verification and Guardrails: Constraining a Probabilistic System into a Product 18
4.2 Observability, Evaluation, and Monitoring 21

5 Common Failure Modes **23**

6 Case Studies **24**
6.1 Toy News Research Agent 24
6.2 Support and Operations Agents 27
6.3 Coding Agents 27
6.4 Enterprise Search 28

7 Chapter Summary **28**
7.1 Question Recap 29
7.2 A Practical Checklist for AI Engineers 30

8 References **32**

aiengineeringhandbook.com

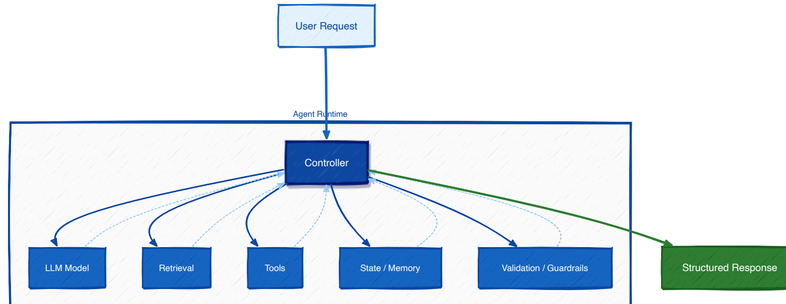

Figure 1: System Over Model — A Modern Agent System Overview

1 Overview

A finance engineer asks an internal assistant: "Summarize the changes from last week that affected the billing system, and tell me which enterprise customers may have been impacted." The answer arrives in seconds. It looks polished. It is concise, fluent, and wrong. The system missed a migration note, ignored a support escalation, and flagged the wrong customers because it never checked production state. In another incident, support asked the system to issue a refund. The runtime retried after a timeout, and the refund executed twice. None of this is "AI magic." All of it is systems failure.

That is the difference between a model demo and an AI application. A real application does not just generate text. It retrieves evidence, calls APIs, manages state, verifies results, enforces permission boundaries, and decides when it should stop. Once AI starts touching the real world, the hard part is no longer just prompting. It is systems engineering.

The user is not talking directly to the model. The user is talking to the runtime. The runtime decides how the model is used, which evidence is retrieved, which tools may be called, which state is retained, and which verifiers must approve the result first. The model sits at the center, but the model itself is not the system.

A useful engineering approximation is:

$$P(\text{task success}) \neq P(\text{model answers correctly})$$

More concretely, you can write:

aiengineeringhandbook.com

$$P(\text{task success}) \approx P(\text{sufficient evidence}) \cdot P(\text{correct action}) \cdot P(\text{grounded answer}) \cdot P(\text{passes verification}).$$

This is not claiming these events are statistically independent. It expresses a systems decomposition: reliability is not obtained inside the model alone. It is earned jointly across multiple components.

So the real question this chapter answers is:

> **How do engineers build AI systems that can interact with the real world and still remain reliable?**

2 What Is an Agent

2.1 An Agent Is a Controller, Not a "Talking Model"

> **!** Important
>
> This section answers a few key questions first:
> 1. What is an "agent application," and how is it different from a single LLM call?
> 2. How should you choose between a workflow and an agent?
> 3. Why is the key to modern AI applications the system rather than the model itself?

A useful definition is: **an agent is a controller that coordinates a model, retrieval, tools, state, and verification to complete a task that may require multiple steps.** The key word is **coordinates**. The model does not directly constitute the application. The application is the execution system built around the model.

You can formalize it as:

$$a_t \sim \pi_\theta(a \mid o_t, s_t, b_t),$$

where (o_t) is the current observation, (s_t) is the explicit task state, (b_t) is the remaining budget, and (a_t) is the next action. That action might be "answer directly," "run retrieval," "call a tool," "ask for clarification," or "stop." After the action executes, the system obtains a new result and updates state:

$$s_{t+1} = f(s_t, o_t, a_t, y_t), \qquad b_{t+1} = b_t - c(a_t).$$

This also explains why an agent is not "a model with extra prompt text." It is a controller that runs under constraints on state, cost, and risk.

aiengineeringhandbook.com

From an engineering standpoint, a single LLM call is well suited to relatively closed tasks such as summarization, rewriting, classification, and extraction. But once a task needs external facts, cross-system actions, multi-step correction, or staged decision-making, one call is not enough. The system has to enter a loop of "observe → decide → act → check." That is where the agent actually starts to matter.

Workflow, Agent, or Agentic RAG?

The safest engineering default is: **if a workflow can solve it, do not rush to use an agent.** In production, low variance is usually worth more than extra autonomy.

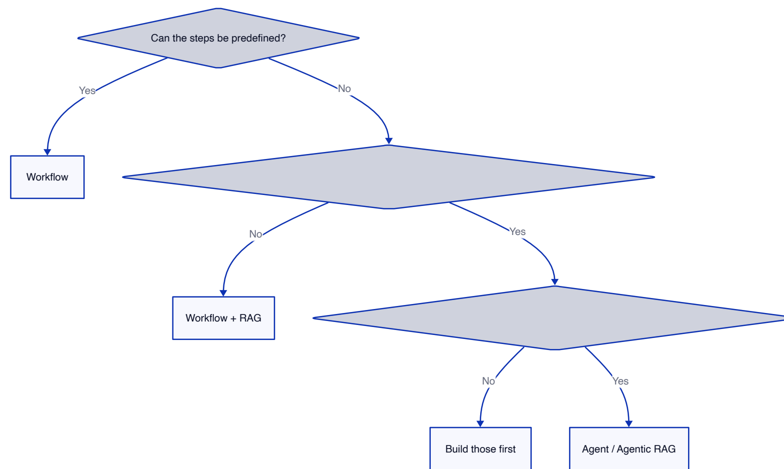

Figure 2: A decision framework for Workflow vs Agent.

Read Figure Figure 2 from top to bottom. If the steps can be enumerated and tested, prefer a workflow. If the main problem is just grounding and not dynamic planning, then `workflow + RAG` is usually enough. Only when the system truly needs iterative search, replanning, or interaction with live world state—and you already have verifiers and budget controls—does an agent become the right abstraction.

A more practical decision table is:

Scenario	Safer default
Steps are stable and can be predefined	Workflow

Scenario	Safer default
External knowledge is needed, but the task is still fundamentally single-turn Q&A	Workflow + RAG
Multi-step search, replanning, or cross-tool judgment is required	Agentic RAG or Agent
Operational risk is high, but validators, budgets, and stop conditions are still immature	Do not ship agent autonomy yet

The reason to think this way is simple: in practice, most failures are not "the model is too small." They are "the system did not design boundaries around uncertainty." Without an explicit state object, you cannot recover or replay. Without tool contracts, you cannot execute safely. Without a verification layer, you cannot constrain a probabilistic system into a product.

2.2 The Agent Control Loop: From a Single Answer to an Iterative System

! Important

This section answers a few key questions first:

1. What is the minimal form of the agent control loop?
2. Why is an agent not "generate an answer once," but an iterative system?
3. Why do feedback loops increase success rates on complex tasks?

The cleanest way to understand an agent is not to memorize a framework name, but to internalize a control loop:

Perceive → Think → Act → Check → Update State → Repeat

After every action, the system can re-enter the perception stage. That is the fundamental difference between an agent and a single generation.

A minimal execution model can be written as:

$$a_t = \pi(o_t, s_t, b_t), \quad y_t = \mathrm{Execute}(a_t), \quad v_t = \mathrm{Verify}(y_t), \quad s_{t+1} = \mathrm{Update}(s_t, y_t, v_t).$$

The loop terminates when the success condition is satisfied or the budget is exhausted:

$$\mathrm{done}(s_t) = 1 \quad \lor \quad b_t \leq 0.$$

aiengineeringhandbook.com

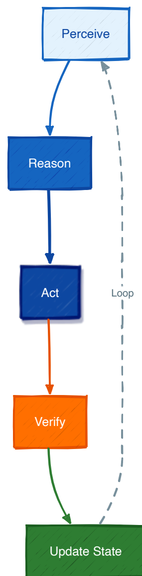

Figure 3: The agent control loop — perceive, think, act, verify, update

aiengl... .com

This loop matters because it rewrites the engineering question from "will the model get it right in one shot?" into "can the system move toward the right answer through feedback?" Research tasks need multi-hop retrieval. Support tasks need to verify live business state. Coding tasks need to run tests and then repair. A single generation is rarely reliable enough.

These five steps each carry a different responsibility:

- **Perceive**: read the user's goal, existing state, tool results, and retrieved evidence;
- **Think**: decide whether to answer, retrieve, call a tool, ask for clarification, or stop;
- **Act**: execute external actions such as API calls, database queries, and code execution;
- **Check**: validate format, permissions, citations, and task constraints;
- **Update State**: distill the facts, conclusions, and progress that affect later decisions.

Many so-called "agent demos" really only have "think" and "act." They have almost no "check" and almost no "update state." So once they enter a long task, they drift almost immediately.

2.3 Where Prompts, RAG, Tools, and Skills Sit in the Loop

> **!** Important
>
> This section answers a few key questions first:
> 1. What problems do prompts, retrieval, tools, and skills each solve?
> 2. Why is none of them "the agent itself"?
> 3. How are they orchestrated by the control loop?

"What changes the next decision?" Prompts and contracts shape how the system thinks and what shape its outputs take. RAG brings evidence into perception and thinking. Tools connect action to the external world. Skills provide reusable procedural knowledge, such as research workflows, support SOPs, or internal operating manuals.

A good engineering analogy is onboarding a new hire:

- **Tools** are the software and APIs they can use;
- **Skills** are the team's existing SOPs, templates, and scripts;
- **RAG** is the documentation and knowledge base they can look up;
- **Prompts and contracts** are the job description and delivery format;
- **The agent** is the controller that decides "what to look up now, what to do now, and when to stop."

aiengineeringhandbook.com

This distinction matters because many teams try to solve agent problems by "writing a longer prompt." In practice, the more reliable gains usually come from better contracts, better retrieval, clearer tool interfaces, and stricter verification.

3 System Architecture

3.1 The Core Components of an Agent System

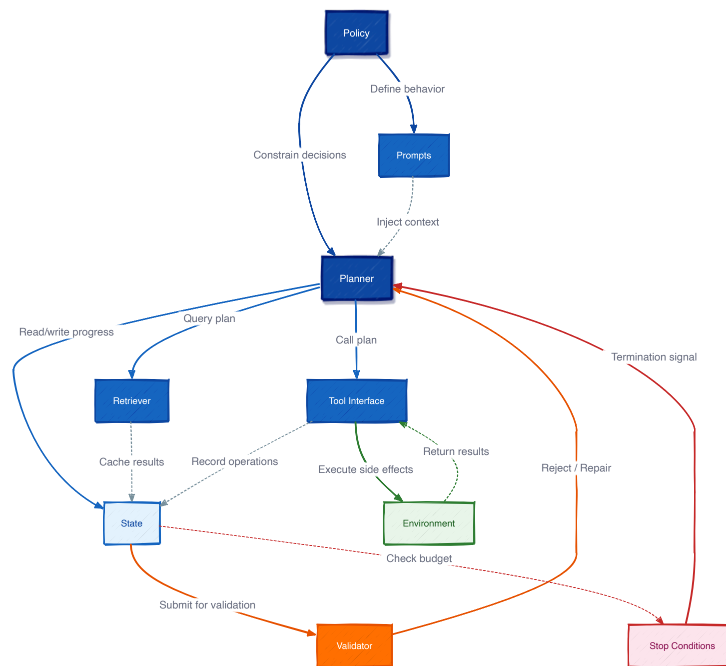

Figure 4: The core components of an agent system — policy, planner, retriever, tools, verifier, termination

> **❗ Important**
>
> This section answers a few key questions first:
> 1. What components usually make up a modern agent system?
> 2. What is the relationship among policy, planner, state, tools, and verification?

3. Why do architecture decisions usually matter more than another round of prompt edits?

If you take apart a production-grade agent system, you will usually see these components:

- **Policy**: model selection, routing logic, high-level action preferences;
- **Prompt / Contracts layer**: output schema, instructions, retry constraints, and result shape;
- **Planner**: optional task decomposition and next-step decision-making;
- **Retriever**: access to external knowledge;
- **Tool Interface**: structured access to external systems;
- **State / Memory**: explicit task progress;
- **Verifier / Guardrails**: acceptance checks, safety checks, and escalation;
- **Termination logic**: budget limits and stop reasons;
- **Environment**: the external world the system observes and acts on.

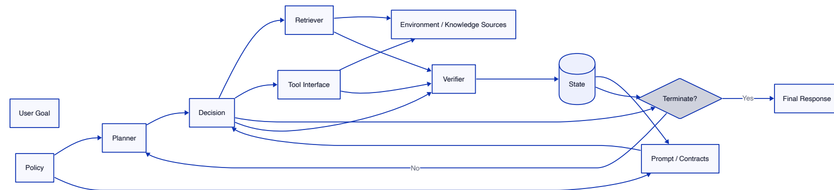

Figure 5: The core components of an agent system.

Read Figure Figure 5 as a control architecture diagram. Policy determines overall system behavior. The planner compresses the goal into the single most worthwhile next step. The retriever and tools expose the environment to the system. The verifier decides whether the result is acceptable. State retains what will actually affect subsequent decisions. The termination node prevents unbounded exploration.

A compact objective for the controller can be written as:

$$a_t^* = \arg\max_{a \in \mathcal{A}(s_t)} \mathbb{E}[U(a \mid o_t, s_t)] - \lambda C(a) - \mu R(a),$$

where (U) is expected utility, (C(a)) is latency or compute cost, and (R(a)) is operational risk. In engineering practice you may not solve this explicitly, but the formula captures the core intuition: every action is a tradeoff among quality, cost, and risk.

Fast models, strong models, and routing

aiengineeringbook.com

A very common system pattern is to use fast models for routing, extraction, or low-risk formatting, and stronger models for complex reasoning, synthesis, and ambiguity resolution. This keeps the overall system within acceptable latency and cost budgets without significantly sacrificing quality.

A planner can be optional, but termination cannot

Some tasks need an explicit planner. Many tasks do not. But every production agent must have a clear termination policy. If the system has not defined "coverage is sufficient," "budget is exhausted," or "must escalate to a human," then it is not a proper controller. It is an unbounded loop.

3.2 RAG: Connecting External Knowledge to the System

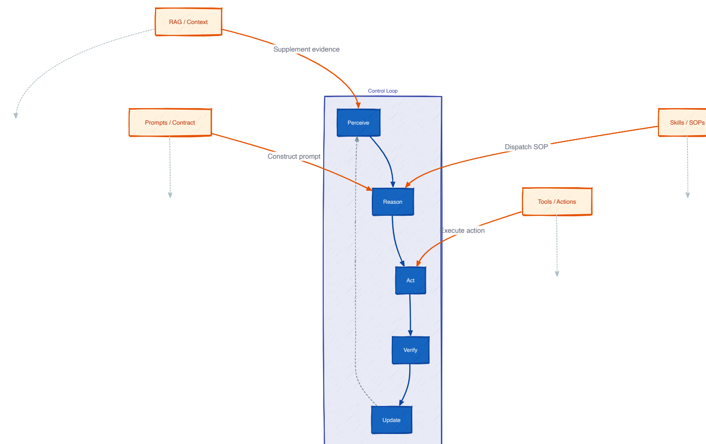

Figure 6: Where RAG / tools / skills / prompts sit in the loop

> **! Important**
>
> This section answers a few key questions first:
> 1. What is the standard RAG pipeline?
> 2. Why does retrieval improve accuracy and grounding?
> 3. When is single-shot RAG enough, and when do you need agentic RAG?
> 4. How should retrieval quality be evaluated?

RAG (Retrieval-Augmented Generation) is, at root, about attaching external knowledge in front of the LLM so the model does not answer from parameter memory alone, but retrieves evidence first and then generates from that evidence.

The engineering goal of RAG is direct: **before generating an answer, retrieve the external evidence most relevant to the current question.** It solves "does the system have enough evidence to answer?" rather than "can the model sound fluent?"

A standard RAG pipeline usually includes:

1. **Ingestion**: parse documents, metadata, and permissions;
2. **Chunking**: split them into retrievable units;
3. **Embedding / Sparse Representation**;
4. **Indexing**: build a queryable structure;
5. **Retrieval**: pull back candidates;
6. **Reranking**: rank them more precisely;
7. **Generation**: answer from evidence, with citations.

At the ingestion stage, you parse documents, extract main text, and preserve metadata and permissions. At the chunking stage, you should not just cut blindly by token count. It is better to split along natural boundaries such as headings, paragraphs, tables, and code blocks. At the representation layer, you often build both dense embeddings and sparse retrieval features. The indexing layer puts vectors and metadata into a queryable structure. At query time, you first do recall, then rerank with a reranker, then feed the evidence to the model for generation, ideally with citations required.

The hard part is how to design each step. If chunks are too small, information fragments. If chunks are too large, noise increases. In retrieval, a common default is hybrid retrieval, because dense retrieval is better at semantic similarity while sparse methods like BM25 are better at exact terms such as error codes, product names, IDs, and function names. If you need high precision after recall, you add a **cross-encoder reranker**. It is usually more accurate, but more expensive, so you only run it over top-(k) candidates.

> **Note**
>
> **A concrete retrieval example**
> Suppose the user asks:
>
> > "Did the March billing migration change retry behavior for enterprise invoices?"
>
> If the system only queries `billing migration retry behavior`, it may first retrieve a general FAQ, an old retry runbook, and an irrelevant incident postmortem. The critical migration note is still missing, because the document uses the term **dunning backoff** rather than **retry behavior**.
> A more mature approach does three more things:
>
> 1. **Query rewriting**: add synonyms such as `retry`, `dunning`, `backoff`, `enterprise invoice`, and `migration`;
> 2. **Hybrid retrieval**: combine semantic search with exact keyword matching;

aiengineeringhandbook.com

3. **Reranking**: place the migration note and support escalation ahead of the general FAQ.

A simplified candidate set might look like this:

Stage	Top results
Before query rewriting	Billing FAQ, general retry runbook, old incident postmortem
After query rewriting + hybrid retrieval	Migration note, support escalation, retry runbook
After reranking	Migration note, support escalation, retry runbook

The resulting answer is no longer "the model sounds plausible." It is "the system actually found the migration note, cross-checked it with the support escalation, and then answered with citations."

Why retrieval improves accuracy

The value of retrieval is that it can supply knowledge the model parameters do not contain, ground answers in visible evidence, and add freshness and traceability. But RAG is not "stuff documents into the model and it gets more accurate." It usually fails in one of these places:

- **Chunking failure**: chunks are too small, so semantics break apart; chunks are too large, so signal gets drowned in noise;
- **Recall failure**: the truly relevant evidence was never retrieved;
- **Grounding failure**: the evidence is already in context, but the model did not actually answer from it.

Dense retrieval, sparse retrieval, and hybrid retrieval

- **Dense retrieval** is good at semantic similarity and paraphrase matching;
- **Sparse retrieval** such as BM25 is good at exact terminology;
- **Hybrid retrieval** is usually the safest place to start in production.

If the system is dealing with error codes, statute numbers, field names, or SKUs, exact matching matters enormously. If it is handling natural-language intent, semantic similarity is indispensable. So mature systems rarely bet on a single retrieval mode.

Reranking, citations, and evaluation

Retrieval gives you candidates. Reranking decides what actually enters the context window. Cross-encoder rerankers are often effective because they jointly model the ((query, document)) pair, but the cost is latency and compute.

At least two retrieval metrics are worth writing down explicitly:

aiengineeringhandbook.com

$$\text{Recall@}k = \frac{1}{|Q|}\sum_{q\in Q}\mathbf{1}\{\exists d \in top_k(q) : rel(q,d) = 1\}$$

and

$$\text{MRR} = \frac{1}{|Q|}\sum_{q\in Q}\frac{1}{rank_q}.$$

`Recall@k` asks: did the correct evidence appear in the top (k) results? `MRR` asks: how high did the first relevant result appear? But these are still not enough. End-to-end evaluation also has to check whether the final answer is grounded, whether it is complete, and whether it abstains correctly when evidence is insufficient.

Beyond recall@k and MRR at the retrieval layer, the answer layer also needs to track faithfulness, completeness, citation coverage, and conflict rate.

Online, you also need to log hit documents, retrieval scores, rerank order, and final citation sources. Otherwise, when something breaks, you cannot localize it.

Single-shot RAG vs Agentic RAG

If the question is clear enough and the evidence base is stable enough, single-shot RAG is sufficient. Agentic RAG re-enters the control loop: it rewrites queries, fills coverage gaps, fetches more sources, and stops when evidence is sufficient or the budget is exhausted. Research assistants, complex enterprise search, and multi-document synthesis usually fall into the latter category.

3.3 Tool Use: Letting the System Interact with the Real World

> **! Important**
>
> This section answers a few key questions first:
> 1. Why do AI systems need external tools?
> 2. What should a good tool interface look like?
> 3. What is structured output, and why is it a prerequisite for tool use?
> 4. Why must permissions, idempotency, and error semantics be implemented outside the model?

If retrieval lets the system **know** external facts, tools let the system **touch** the external world. A model without tools can at most generate language. Once tools exist, the system can read live state, execute actions, verify results, and even change the world.

Common tool types include:

aiengineeringhandbook.com

- API calls
- Database queries
- Code execution
- Search and fetching
- Business actions such as tickets, email, refunds, and deployments

Tool contracts are the real interface

In production, a tool is not just "expose a function to the model." It is first an interface contract. At minimum it should define:

- input schema
- output schema
- parameter constraints
- timeout and retry semantics
- error classes
- side-effect boundaries
- risk level
- whether an idempotency key is required

A typed tool signature expresses design intent better than a sentence of natural language:

```
def refund_payment(
    payment_id: str,
    amount_cents: int,
    reason: Literal["duplicate_charge", "service_failure", "goodwill"],
    idempotency_key: str,
    dry_run: bool = False,
) -> RefundResult:
    ...
```

This is far more useful than "the assistant can issue a refund when needed." The contract tells the system what it can do, how it must do it, and what constraints the runtime can enforce.

Structured output: from text to executable object

Tool calling works only if the model's output is no longer treated as free text, but as a typed object. For example:

```
{
  "action": "refund_payment",
  "payment_id": "pay_01483",
  "amount_cents": 4999,
  "reason": "duplicate_charge",
  "idempotency_key": "refund:pay_01483:4999:v1",
  "dry_run": true
}
```

aiengineerhandbook.com

Compared with “I think we should refund this payment,” the object above can be validated, replayed, rate-limited, audited, and rejected directly when it violates policy.

For write tools, idempotency is not optional

For write tools that create side effects, a minimal correctness requirement is:

$$f(f(s,x,k),x,k) = f(s,x,k)$$

That is: under the same state (s), the same parameters (x), and the same idempotency key (k), repeated submission must not amplify the side effect.

> **Incident case: duplicate refund**
>
> A support assistant chose the right refund tool. After a network-layer timeout, the runtime automatically retried. Because the tool had not implemented idempotency, the same refund executed twice. The root cause was not the model. It was the interface contract.

Permissions, approvals, and rollback must be implemented outside the model

The model can make a recommendation, but it cannot decide whether it is authorized to execute it. Permissions, allowlists, tenant boundaries, approval flows, and human escalation must be enforced in the runtime and at the tool layer.

A practical way to classify risk is:

Risk level	Example	Typical controls
`read_low`	search, query, read state	timeout limits, result-count limits, least privilege
`write_medium`	create ticket, draft email	idempotency key, dry-run, result confirmation
`write_high`	refund, order cancellation, production change	dual confirmation, approval, human intervention

Tool errors must also be structured

Do not dump raw stack traces back into the model. A more stable approach is to return structured error labels such as:

- `retryable_timeout`

aiengineeringhandbook.com

- `invalid_args`
- `not_found`
- `permission_denied`
- `rate_limited`

This lets the runtime distinguish among "retry after repair," "ask the user for more information," and "must stop immediately."

3.4 State and Memory: Letting Tasks Span More Than One Inference Step

> **!** Important
>
> This section answers a few key questions first:
> 1. Why do long tasks require state management?
> 2. What is the difference between state and context?
> 3. How should short-term context and long-term memory divide responsibilities?
> 4. Why can uncontrolled memory drag system performance down?

Once a task extends beyond a single model call, the system has to represent "what has already happened" and "what should happen next." That is exactly what state and memory are for.

State is not a pile of chat history

State should be a structured object that stores the facts, constraints, decisions, and intermediate results the current task truly depends on. **Context**, by contrast, is the token payload actually sent into the model at the current step. The two are related, but they are not the same thing.

A useful budget equation is:

$$L_{\text{system}} + L_{\text{working}} + L_{\text{retrieved}} + L_{\text{response}} \leq L_{\text{max}},$$

where L_{max} is the model context window. That is why context engineering matters: every token you allocate to old logs is a token you lose for current evidence or final reasoning.

A layered memory architecture

Not everything the system knows should stay in the model window all the time. Pinned context is small and stable. The working set changes quickly as the task advances. Retrieved context enters on demand. Long-term memory lives outside the window and is pulled in only when truly relevant.

Short-term context vs long-term memory

aiengineeringhandbook.com

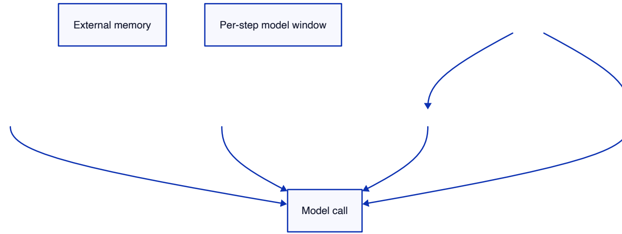

Figure 7: A layered memory and context architecture.

- **Short-term context** serves the current task: recent interactions, the active working set, temporary notes;
- **Long-term memory** stores what remains valuable across sessions: user preferences, persistent facts, approved procedures, stable configurations.

Long-term memory without provenance, timestamps, permissions, and TTL quickly turns into a hallucination cache. It looks as if the system "remembers more," but what it is really doing is retaining stale or unauthorized information for too long.

Compression and Context Rot

As the task trace grows, the model gets slower and less able to accurately locate the key information. This is often called **context rot**. The remedy is compression:

1. keep facts rather than raw payloads;
2. keep citations rather than entire documents resident all the time;
3. compress long traces into structured summaries;
4. when raw evidence is truly needed, fetch it back by ID.

A useful compression metric is:

$$\rho = \frac{L_{\text{raw}}}{L_{\text{summary}}},$$

but a high compression ratio is not the goal by itself. The real goal is to maintain high information density within the context budget.

aiengineeringhandbook.com

Note

A concrete example After 25 tool calls in a research task, the system should not keep all 25 original documents in the active window. It should keep the task goal, the queries already tried, accepted claims, unresolved conflicts, citations, and budget consumption, and then re-fetch raw material only when it is actually needed.

4 Engineering

4.1 Verification and Guardrails: Constraining a Probabilistic System into a Product

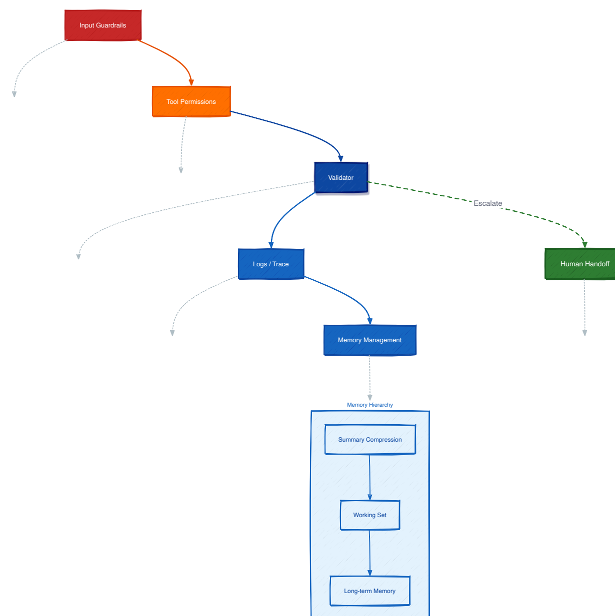

Figure 8: Production-ready agents — guardrails, observability, memory management

ook.com
aien

> **!** Important
>
> This section answers a few key questions first:
>
> 1. How do verification mechanisms improve system reliability?
> 2. What do guardrails actually do in a production system?
> 3. What are schema checks, rule checks, and LLM judges each good for?
> 4. Why must jailbreaks and prompt injection be defended separately?

If the model is responsible for "producing candidate results," then verification is responsible for "deciding whether that result is allowed to enter the system." This is also the part of AI engineering that most resembles software engineering: output has to become an object that can be checked, rejected, repaired, or escalated.

Guardrails do not exist only in the prompt. They exist at the input boundary, in the runtime, in tool permissions, in verifiers, in the memory architecture, in logging systems, and in the human escalation path. That is why production reliability is a system property, not a prompt property.

The three-layer verification stack

A practical verification stack usually includes:

1. **Schema validation**: is the structure valid?
2. **Rule checks**: are permissions, citations, invariants, and risk thresholds satisfied?
3. **LLM-as-judge**: is the result complete, coherent, and does it cover the task?

A compact acceptance function can be written as:

$$\text{accept}(y) = \mathbf{1}[S(y) = 1 \land R(y) = 1 \land J(y) \geq \tau],$$

where (S) is schema validation, (R) is deterministic rule checking, (J) is the judge score, and ( ) is the acceptance threshold. The order matters: anything that can be checked deterministically should never be handed to the judge.

Generate → Verify → Repair

In production, a more reliable pattern is:

```
for attempt in range(3):
    out = llm(prompt, context)
    ok, err = validate(out)  # schema + 规则
    if ok:
        return out
    prompt = prompt + f"\n请修复以下验证错误：{err}"
raise RuntimeError("validation_failed")
```


aiengineeringhandbook.com

This is much stronger than "try again," because the retry is not blind sampling. It is directed repair under the constraints of a verifier.

> ℹ Note
>
> **An example: validating a cited answer**
> Suppose the model outputs:
>
> ```
> {
>   "answer": " 这次迁移缩短了企业发票的重试窗口。",
>   "citations": []
> }
> ```
>
> This may look fluent at the language level. But if the task requires citations, then the schema or rule layer should reject it immediately. The judge layer should not be used to solve something that was deterministically checkable in the first place.

Jailbreak vs prompt injection

- **Jailbreak**: the user directly tries to override system policy;
- **Prompt injection**: an external document, once retrieved or read, tries to masquerade as a higher-priority instruction.

The defenses are different. Jailbreak requires stronger policy and refusal logic. Prompt injection requires that you always treat external content as **data**, not **instructions**. Retrieved text may be used as evidence. It may not rewrite system policy.

When human escalation is mandatory

When the system faces high-risk write actions, repeated tool failures, evidence conflicts it cannot resolve, or low confidence with real business impact, human intervention is not "conservative." It is correct system design.

> ℹ Incident case: enterprise search permission leak
>
> An enterprise search system did retrieve the correct confidential document and ranked it near the top, but access control was only applied at render time. So while the body text was blocked, the snippet in the results list still leaked sensitive content. The real fix was architectural: enforce access control before retrieval, during ranking, and before display.

aiengineeringhandbook.com

4.2 Observability, Evaluation, and Monitoring

> **!** Important
>
> This section answers a few key questions first:
> 1. How do engineers monitor AI system behavior?
> 2. What is the minimum a production agent should log?
> 3. What does a good evaluation scorecard and release gate look like?
> 4. If online task performance degrades after a model upgrade, how should you debug it?

A system that only inspects the final answer is almost impossible to truly debug. Agent systems fail along execution traces, not only inside a single output string. So observability has to cover the full trace.

What you should at least log

Layer	Minimum fields to log
Input layer	request, session ID, task goal, user or tenant scope, version information
Model layer	model version, prompt version, token usage, latency, stop reason
Tool layer	tool name, parameters, result summary, error labels, retry count
Retrieval layer	queries, hit document IDs, scores, rerank order, citation sources
State layer	state snapshots, budget consumption, unresolved conflicts
Safety layer	guardrail triggers, refusal reasons, human handoff reasons

Read failures in the right order

Do not blame the model first. Look at the tool layer, then the retrieval layer, then context construction, then generation, and only then guardrails. This order saves time because many problems that look like "reasoning failure" are actually schema drift, stale indexes, or broken context assembly.

Release scorecards and release gates

A serious agent project needs a gold set and a release gate. A compact scorecard should at least track:

- task success rate
- tool selection accuracy
- grounding / citation pass rate

aiengineeringhandbook.com

- safety violation rate
- cost per task
- (p95) latency
- escalation rate

You can write the release gate as:

$$\text{release} = \mathbf{1}[SR \ge \tau_s \wedge TA \ge \tau_t \wedge GP \ge \tau_g \wedge SV \le \tau_v \wedge P95 \le \tau_l \wedge C \le \tau_c],$$

where (SR) is success rate, (TA) is tool accuracy, (GP) is grounding pass rate, (SV) is safety violation rate, (P95) is latency, and (C) is cost per task.

A model with better benchmarks but worse tool accuracy and grounding pass rate should not ship.

Cost is part of quality

A useful decomposition of per-task cost is:

$$C_{\text{task}} = c_{\text{tok}}T + \sum_i c_{\text{tool},i} + c_{\text{rerank}}R + c_{\text{human}}H.$$

A system may be accurate. If its cost is an order of magnitude higher, it is still not production-ready.

> **Note**
>
> **Incident replay: a model upgrade causes schema drift**
> Imagine a model upgrade after which general benchmarks improve, but online task success drops. Why? The logs show that `invalid_args` errors on a certain tool rise sharply. Before the upgrade, the model often emitted `{"ticket_id": "INC-10428"}`; after the upgrade, it frequently emits `{"id": "INC-10428"}`. The benchmark improved, but the tool contract broke.
> This is why replay evaluation is necessary. If you compare only the final natural-language answers, you may miss the fact that the new model has already damaged tool-calling behavior. The real fix may be to harden the contract, update the verifier, or adjust tool prompting for that model specifically, rather than simply rolling back.

aiengineeringhandbook.com

5 Common Failure Modes

> **!** Important
>
> This section answers a few key questions first:
> 1. What are the most common failure modes of agents?
> 2. At which layer do these failures usually originate?
> 3. What engineering lessons should teams learn from them?

Agent failures are usually not mysterious. Most production incidents eventually collapse into a small number of recurring patterns.

Table 5: Common agent failure modes, root causes, and engineering lessons

Failure mode	Symptom	Root cause	Engineering lesson
Tool misuse	Chooses the wrong tool, passes wrong parameters, or routes a read request into a write action	Ambiguous tool names, overlapping interfaces, weak schema, and missing runtime control over permissions and side-effect boundaries	Tool design is interaction design in disguise—the model sees the contract, not your intent
Retrieval failure	Answers the wrong question, cites irrelevant content, or ignores evidence that is clearly present in the knowledge base	Bad chunking, stale indexes, retrieval-mode mismatch, missing reranking, or generation that is not actually grounded in sources	Check recall and context construction before editing the prompt
State explosion	As the task gets longer, the system gets slower, more expensive, and more likely to forget key facts	Raw tool results and full history are continually appended to the window, with no compression or layering	Context is a budget, not a warehouse

aiengineeringhandbook.com

Failure mode	Symptom	Root cause	Engineering lesson
Hallucinated output	Delivers high-confidence conclusions without supporting sources, or proposes unauthorized actions	Weak verification, no citation requirement, missing rule checks, no approval path	"Looks right" is not correct—the system must pass verification before it earns the right to act
Infinite exploration and budget runaway	Keeps searching, fetching, and rewriting queries but never converges	No explicit budget, no stop condition, no definition of "coverage is sufficient"	Autonomy without termination logic is just another form of uncontrolled cost

6 Case Studies

6.1 Toy News Research Agent

> **! Important**
>
> This section answers a few key questions first:
> 1. How would you design an AI research assistant?
> 2. What architecture supports search, fetch, synthesis, and stopping?
> 3. What does an end-to-end agentic RAG loop look like?

A research assistant is one of the best cases for understanding agents, because the task naturally requires iterative search. User questions are often underspecified. Evidence can conflict. The system has to decide whether to keep searching or whether it already has enough to stop.

The runtime owns the control loop. `news.search` and `news.fetch` are tools. The RAG layer handles deduplication, reranking, and claim extraction. The state object records coverage and conflicts. Budgets and `stop_reason` prevent endless search.

An end-to-end walkthrough

Suppose the user asks:

> "Within 30 seconds, write a 6-bullet briefing summarizing the important AI agent releases this week, and include citations."

aiengineeringhandbook.com

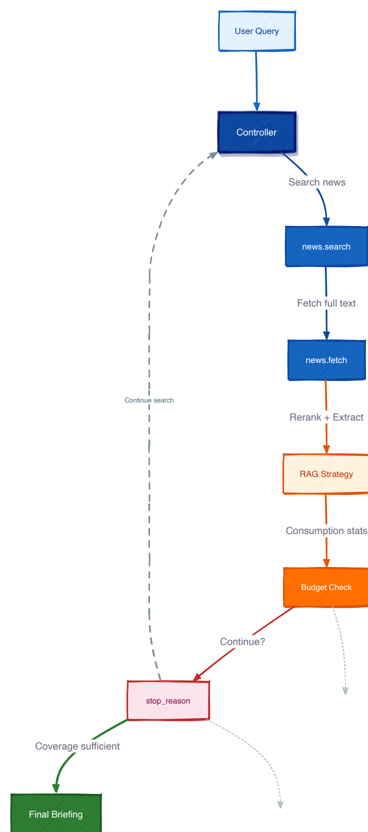

Figure 9: Toy news agent case — the full path from query to briefing

aien.com

A reasonable system path is:

1. initialize the goal and budget: max number of sources, max number of tool calls, max wall-clock time;
2. generate a short query plan;
3. call `news.search` for broad-coverage search;
4. fetch only the most promising results;
5. extract candidate claims and bind citations;
6. decide whether coverage and conflicts are already sufficient;
7. if sufficient, stop; otherwise continue searching;
8. generate a structured briefing, with `stop_reason` attached.

A minimal loop can be written as:

```
goal = {"topic": "AI agent announcements", "max_sources": 10, "max_tool_calls": 20}
state = {"queries_tried": [], "claims": [], "sources": [], "stop_reason": None}

while True:
    if len(state["sources"]) >= goal["max_sources"]:
        state["stop_reason"] = "source_budget_reached"
        break
    if tool_calls_used() >= goal["max_tool_calls"]:
        state["stop_reason"] = "tool_budget_reached"
        break

    plan = llm("Choose next action: search, fetch, or stop", context={"goal": goal, "state"

    if plan.action == "search":
        hits = news.search(plan.query)
        state = update_with_hits(state, hits)
    elif plan.action == "fetch":
        doc = news.fetch(plan.url)
        state = update_with_doc(state, doc)
    elif plan.action == "stop":
        state["stop_reason"] = plan.reason
        break

brief = llm("Write a cited briefing", context={"state": state})
```

What matters is not the code itself, but the architecture it expresses: the system decides, acts, updates state, and stops within budget. That is the difference between a "controller" and a "chat wrapper."

6.2 Support and Operations Agents

> **!** Important
>
> This section answers a few key questions first:
> 1. What tools and guardrails does a support agent need?
> 2. When should the system handle a request automatically, and when must it escalate?
> 3. Why do these systems depend more heavily on permissions and risk controls than research assistants?

Support and operations agents do not just answer questions. They may query orders, cancel subscriptions, resend invoices, or even prepare refunds. So the core challenge in these systems is not just answer quality, but **controlled authorization**.

A safer path usually looks like this:

1. classify intent and risk level;
2. route low-risk read requests through deterministic workflows;
3. allow the system to draft medium-risk actions, but require user confirmation;
4. require approval or human escalation for high-risk write actions.

A simple state machine can be written as:

$$\text{requested} \rightarrow \text{drafted} \rightarrow \text{confirmed} \rightarrow \text{executed} \rightarrow \text{verified} \rightarrow \text{closed}.$$

If a system is able to jump directly from `requested` to `executed` on a high-risk action, it is not mature enough for production.

6.3 Coding Agents

> **!** Important
>
> This section answers a few key questions first:
> 1. Why can a coding agent not work without "verify after acting"?
> 2. What kind of loop fits code search, editing, testing, and repair?
> 3. Why is test feedback so critical in the coding setting?

Coding agents are special because software engineering provides strong external judges: tests, compilers, linters, and static analyzers. That makes coding a natural fit for the **generate** → **execute** → **verify** → **repair** loop.

Common tools include:

- repository search

aiengineeringhandbook.com

- file read and edit
- test runners
- lint / type-check / compile tools
- diff review and rollback tools

A mature coding agent does not declare success immediately after editing a file. It continues to observe test results, compile output, and policy constraints, and only then decides whether it is worth continuing to repair. Coding makes one thing especially clear: reliable AI does not come from "how good the first draft is." It comes from whether the system can keep converging under external verification.

6.4 Enterprise Search

> **! Important**
>
> This section answers a few key questions first:
> 1. Why is enterprise search often one of the highest-value AI systems a company can build?
> 2. How is it different from general-purpose RAG?
> 3. Why do permissions, metadata, and freshness matter more here than writing style?

Enterprise search is often more valuable than a "general-purpose chatty agent," because many real business problems are, at root, "find the right information within the right permission boundary." The system has to give the right user, at the right time, the right answer from the right source.

Compared with open-domain RAG, enterprise search places more emphasis on:

- private data connectors
- permission filtering
- freshness and version management
- citations and traceability
- metadata-based ranking

An excellent enterprise search system often solves business pain more directly than a "smarter-looking" general agent. But it also punishes sloppy architecture more quickly: permission checks, metadata filtering, and index freshness all have to be elevated into first-class design concerns.

7 Chapter Summary

Once AI starts touching the real world, the unit of design is no longer the prompt, but the control loop: what the system can observe, what it is allowed to do, how it stores state, how it verifies results, and how it fails safely when

aiengineeringhandbook.com

uncertainty remains. This is why AI engineering increasingly looks like systems engineering. The model matters, of course. But the product truly lives in the contracts, budgets, traces, verifiers, and recovery paths around it. Reliable agents are not built by "hoping the model behaves like an engineer." They are built by "designing good engineering behavior into the system defaults."

7.1 Question Recap

1. What is an AI agent? An agent is a controller that coordinates models, retrieval, tools, state, and verification to complete tasks through multi-step feedback.

2. What is the difference between an AI system and a standalone large language model? A standalone model only handles reasoning and generation. An AI system places the model inside a closed loop that includes retrieval, tools, state, guardrails, and monitoring.

3. What components typically make up a modern AI system? A model, retrieval, tools, state or memory, verification or guardrails, and the control loop that coordinates these components.

4. What is the difference between a workflow and an agent? In a workflow, the steps are predefined by the engineer. An agent decides the next step at runtime. The former is more stable; the latter is more flexible.

5. Why do AI systems need external tools? Because model parameters are not live world state. To get current information, execute real actions, and verify results, the system has to connect to external tools.

6. How does the agent control loop work? The system reads the environment and state, decides the next step, executes an action, verifies the result, updates state, and keeps looping until success or budget exhaustion.

7. What role does retrieval play in AI systems? Retrieval brings external evidence into the current reasoning step, improving grounding, freshness, and traceability.

8. Why do long tasks require state management? Multi-step tasks have to retain intermediate conclusions, attempted paths, unresolved conflicts, and budget consumption. Without state management, the system can only keep stuffing raw history back into context.

9. How do verification mechanisms improve system reliability? They turn model outputs into checkable objects. Schema validation, rule checks, and judge models let the system run a "generate $\rightarrow$ verify $\rightarrow$ repair" loop.

10. What are the most common failure modes of agents? Tool misuse, retrieval failure, state explosion, ungrounded hallucinated output, and runaway exploration caused by missing budgets or stop conditions.

aiengineeringhandbook.com

11. How would you design an AI research assistant? Start from agentic RAG: query rewriting, retrieval, fetching, reranking, claim extraction, citation binding, budget control, and explicit stop reasons.

12. What architecture supports both tool calling and retrieval? A dedicated agent runtime that acts as a controller connecting the model layer, retrieval layer, tool layer, state layer, verification layer, and logging / budget infrastructure.

13. How do you keep an agent safe in production? Use tool contracts, structured outputs, permission control outside the model, risk tiering, verifiers, human escalation, observability, and release gates.

14. How do engineers monitor AI system behavior? Log inputs, model versions, tool calls, retrieval hits, state transitions, budget usage, stop reasons, safety events, cost, and latency.

15. What tradeoffs exist between agent autonomy and system reliability? The more autonomy you allow, the more flexibility you get—but also more variance, harder debugging, and more safety risk. The default path should begin with low-autonomy workflows.

16. How do you decide whether a task should be a workflow, an agent, or agentic RAG? If the steps can be enumerated, use a workflow. If the core problem is only grounding, use `workflow + RAG`. Only if the system must iteratively search, replan, or interact with live state should you consider agentic RAG or an agent.

17. If an agent can execute write actions, how would you design idempotency, rollback, and approval? Require strongly typed tool contracts, idempotency keys, explicit risk tiering, dry-run, a confirmation phase, compensating operations where needed, and human approval for high-risk writes.

18. A model upgrade improved benchmarks, but online task success dropped. How would you debug it? Replay the gold task set first, compare tool-calling traces, inspect schema validation failures, retrieval behavior, and context construction, and only then decide whether the degradation truly comes from the model itself.

7.2 A Practical Checklist for AI Engineers

Define the task first, then discuss architecture

Clarify first:

- what is the input?
- what is the output?
- what counts as success?
- what are the latency and cost budgets?
- how expensive is failure?

aiengineeringhandbook.com

If you cannot answer these yet, you are not ready to discuss agent architecture.

Build the workflow first, then hand the real long tail to the model

Hard-code the deterministic paths first. Give the model only the parts that truly require flexible reasoning.

Design tools as contracts

Every tool should define:

- input / output schema
- timeout and retry semantics
- error labels
- side-effect boundaries
- risk level
- idempotency strategy
- audit fields

Keep state minimal and explicit

Store only the facts, decisions, citations, conflicts, and budget values that truly affect task progression.

Move verification earlier

Use structured outputs, schema checks, citation rules, risk rules, and escalation logic as early as possible.

Do observability by default

If you cannot replay a failed task after launch, it does not yet deserve to be called a production system.

Put budgets on autonomy

At minimum, explicitly limit:

- maximum number of iterations
- maximum number of tool calls
- maximum tokens
- maximum wall-clock time
- high-risk write permissions

Use evaluation sets to drive system evolution

A representative task set should at least cover:

- high-frequency paths
- long-tail exceptions
- multi-tool flows
- prompt injection attempts
- high-risk requests
- schema drift and release regressions

aiengineeringhandbook.com

A compact summary table can be:

Build stage	Primary goal	Key artifacts
Workflow	Determinism	fixed DAG, tests
Workflow + RAG	Grounding	retrieval metrics, citations
Agentic RAG	Iterative search	budget + stop logic
Full Agent	Controlled action	tool contracts, verifiers, human handoff

8 References

1. OpenAI. *A Practical Guide to Building Agents.* 2025.
2. Anthropic. *Building Effective Agents.* 2024.
3. Anthropic. *Effective Context Engineering for AI Agents.* 2025.
4. Anthropic. *Equipping Agents for the Real World with Agent Skills.* 2025.

aiengineeringhandbook.com