AI Agents: Patterns and Evals

Version Nov 2026 · verified 2026-11-02

How LLM agents work and fail — the loop, tool design, context engineering, subagents, guardrails, evals (pass@k, pass^k) and budgets.

Video walkthrough — scan the code to watch.

Contents

Mental model 3

Workflows vs agents 3

The agent loop 5

Tool design 7

Context engineering and memory 9

Subagents and handoffs 10

Structured output and guardrails 12

Evals 13

Cost, latency and failure modes 15

Further reading 16

Mental model

An agent is a loop: call the model, run the tool it asked for, append the observation, call the model again — until it answers without a tool call or a stopping rule fires. The model is stateless, so everything it "knows" on each turn is what you put in the context window; most agent engineering is choosing that content and deciding when to stop. Evals are how you find out whether a change made it better, because a single run proves nothing about a nondeterministic system.

Workflows vs agents

Pattern	Shape
Prompt chaining	Fixed sequence; check between steps
Routing	Classify, then send to a specialist
Parallelization	Split into sections, or vote
Orchestrator-workers	LLM splits work at runtime

Pattern	Shape
Evaluator-optimizer	One generates, one critiques, repeat
Agent	LLM chooses its own steps and tools

In a workflow your code fixes the path and the LLM fills in steps; in an agent the LLM directs the path. Agents trade latency, cost and predictability for flexibility, so start with one well-prompted call, then a workflow, and reach for an agent only when the steps cannot be known in advance.

```
type LLM = (prompt: string) => Promise<string>;
declare const llm: LLM;

// Routing: a workflow, not an agent
async function handle(t: string) {
  const kind = await llm(
    `Reply refund|bug|other only:\n${t}`);
```

```
  if (kind.trim() === "refund")
    return llm(`Draft a refund reply:\n${t}`);
  return llm(`Draft a support reply:\n${t}`);
}
```

The agent loop

Part	Purpose
Context	Instructions, history, tool results
Model call	Returns text or tool calls
Tool execution	Your code acts; result appended
Stop rule	No tool call, step cap, budget, timeout

Each turn appends the model's tool calls and their results to the history, so context grows every step. The loop ends on a final answer; everything else — step caps, budgets, a human checkpoint — is a stopping rule you add because the model will not reliably stop itself.

```ts
type Call = { name: string; input: unknown };
type Turn = { text: string; calls: Call[] };
declare function model(h: unknown[]): Turn;
declare function run(c: Call): Promise<string>;

async function agent(task: string, max = 10) {
  const history: unknown[] = [task];
  for (let step = 0; step < max; step++) {
    const turn = await model(history);
    if (!turn.calls.length) return turn.text;
    for (const c of turn.calls)
      history.push(c, await run(c));
  }
  throw new Error("step cap reached");
}
```

> **Gotcha:** Without a step cap an agent that keeps retrying a failing tool loops until the context fills or the bill arrives. Always cap steps and spend, and surface “cap reached” as a failure.

Tool design

Do	Why
Few tools, no overlap	Ambiguous choice is a top failure
Name and describe like an API doc	The model reads only that
Constrain inputs (enums, ranges)	Fewer malformed calls
Make mistakes impossible	e.g. require absolute paths
Return actionable errors	The model can retry correctly

The tool definition *is* the prompt for that capability, so invest in it as you would in a human interface. Anthropic's term is "poka-yoke": change arguments so errors cannot happen — their SWE-bench agent stopped making path mistakes once tools required absolute paths. Keep results token-efficient: return the fields needed, not the whole record.

```
{
  "name": "search_orders",
  "description":
    "Find orders by status. One ID: get_order",
  "input_schema": {
    "type": "object",
    "properties": {
      "status": { "enum": ["open", "shipped"] },
      "limit": {
        "type": "integer", "maximum": 50 }
    },
    "required": ["status"]
```

```
  }
}
```

Context engineering and memory

Technique	What it does
Minimal system prompt	Right “altitude”; add rules from failures
Just-in-time retrieval	Keep paths/IDs; load data via tools
Compaction	Summarise history near the limit
Clear old tool results	Cheapest compaction
Structured notes	Persist state outside the window

Recall degrades as the window fills (“context rot”), so treat tokens as a budget, not free storage. Prefer references the agent can resolve on demand over pre-loading everything. For long tasks, have the agent keep

a notes file or todo list it re-reads, so a compaction or a fresh session loses nothing that matters.

```
type Msg = { role: string; text: string };
declare function summarise(m: Msg[]): string;

async function compact(h: Msg[], keep = 6) {
  if (h.length <= keep) return h;
  const note = { role: "user",
    text: summarise(h.slice(0, -keep)) };
  return [note, ...h.slice(-keep)];
}
```

Subagents and handoffs

Pattern	Control flow
Manager (agents as tools)	Lead calls specialists, keeps control
Decentralized handoff	Control passes to a peer agent

Pattern	Control flow
Subagent	Fresh context; returns a summary

A subagent explores in its own clean context and hands back a condensed result — often 1,000–2,000 tokens — so the lead's window stays small. Use them for parallel, independent work or a narrow toolset. Handoffs suit triage, where a different agent should own the rest of the conversation.

```
type Agent = (task: string, tools: string[]) =>
  Promise<string>;
declare const agent: Agent;

const areas = ["auth/", "billing/", "api/"];
const ask = (a: string) =>
  `List SQL injection risks in ${a}. ` +
  "Reply in under 200 words.";
```

```
const findings = await Promise.all(areas.map(
  (a) => agent(ask(a), ["read", "grep"])));
```

> **Gotcha:** A subagent starts with none of the parent's history. Whatever it needs — paths, constraints, the decision so far — must be in the task text you hand it.

Structured output and guardrails

Guardrail	Catches
Schema validation	Malformed or missing fields
Input checks	Off-topic, jailbreak, PII
Tool permissioning	Actions beyond the task
Human approval	Irreversible or high-value actions
Escalation rule	Repeated failure to understand intent

Validate every model output that code consumes against a schema and retry or fail on mismatch — never

`JSON.parse` and hope. Guardrails are layered checks around the model, not a better prompt. Plan for a human in the loop when the agent is about to do something expensive or irreversible, or keeps failing.

```
import { z } from "zod";

const Refund = z.object({
  orderId: z.string().regex(/^ord_\w+$/),
  cents: z.number().int().positive().max(5e4),
});
const raw = '{"orderId":"ord_9","cents":1200}';
const r = Refund.safeParse(JSON.parse(raw));
if (!r.success) throw new Error("invalid");
console.log(r.data.cents); // 1200
```

Evals

Term	Meaning
Task	One input with success criteria
Trial	One attempt; run several per task

Term	Meaning
Grader	Code, model or human check
pass@k	≥ 1 of k trials succeeds
pass^k	All k trials succeed

Start with 20–50 tasks drawn from real failures; do not wait for a big suite. Grade the outcome — is the refund in the database — not the exact path taken. Prefer code graders; calibrate LLM-as-judge against humans, give it an “Unknown” option, and judge one dimension per grader. Capability evals start low; regression evals should sit near 100% and run on every prompt or model change.

```
// p = per-trial success rate
const passAtK = (p: number, k: number) =>
  1 - (1 - p) ** k;
const passHatK = (p: number, k: number) =>
  p ** k;
```

```
passAtK(0.75, 3);  // ≈ 0.98: retry is fine
passHatK(0.75, 3); // ≈ 0.42: must always work
```

> **Tip:** Read transcripts before trusting a score. They show whether a failure was the agent's mistake or a grader rejecting a valid answer.

Cost, latency and failure modes

Failure	Countermeasure
Loops and retries	Step cap, budget, repeat detection
Wrong or misused tool	Fewer, sharper tools; eval it
Claimed success	Verify outcome with code
Context overflow	Compaction, subagents, notes
Injected instructions	Treat tool output as data

Every step re-sends the growing context, so cost and latency rise faster than step count. Set a budget per task,

use a cheaper model for routing and simple subagents, and parallelise independent calls. Track tokens, steps and wall time per run in your evals so a prompt change that doubles cost shows up before production.

```
class Budget {
  private spent = 0;
  constructor(readonly maxUsd: number) {}
  // illustrative rates, not a price list
  charge(inTok: number, outTok: number) {
    this.spent += inTok * 3e-6 + outTok * 15e-6;
    if (this.spent > this.maxUsd)
      throw new Error("budget exceeded");
  }
}
new Budget(0.5).charge(20_000, 1_000); // $0.075
```

Further reading

- Building effective agents
- Effective context engineering for AI agents

- Demystifying evals for AI agents
- A practical guide to building agents