Free chapters



The first two chapters

What an agent actually is, and Workflows before agents: the first two chapters of Agents in Production, the practitioner's playbook for agentic frameworks, MCP, skills and AI automation. Free.

Written by AI agents (Claude), directed by Yann

2026 to 2027 edition

What this book is

Agents in Production is a playbook for taking AI agents from a promising pilot to a system that runs every day and earns its keep. It is vendor-neutral and built around one mental model: an agent is a model, tools, a loop and context, wrapped in a harness that enforces budgets, guardrails, state and telemetry.

It is written for four readers, plus a Claude track for teams that build on Claude: the Builder, the Leader, the Solo operator and the no-code Ops builder. Each chapter says which sections your track can skip.

You will be able to:

- Decide when a fixed workflow beats an agent, and design tools, MCP servers, skills and context a model can use.
- Choose a framework, or none, and wrap the loop in a harness with budgets, guardrails, state and telemetry.
- Run agents with evals, tracing, cost per success, security and governance, including the EU AI Act.
- Put agents to work building and testing software, with a human gate at each station that needs one.
- Roll an agent out, or stop it, on evidence a steering meeting can check.

Written by AI agents, on purpose

This book was written by a team of AI agents. Claude, a model made by Anthropic, orchestrated more than thirty specialized agents that researched, drafted, verified against primary sources, drew the figures and built the pages you are reading. A human, Yann, directed the work through short multiple-choice decisions and is accountable for what is published. A book about agents in production is also a test of agents in production: if you find an error, we want to hear about it. The Afterword, "How this book was made", shows the pipeline, the numbers and the mistakes caught and possibly missed.

How it handles facts and stories

Every specific claim (a number, a date, a version, a legal deadline) links to its source, and the website lists every source by chapter ([SITE URL]). The companion workbook holds the exercises, solutions, running project and the Claude Kit appendix. Illustrative numbers are labeled as such. One conflict of interest: the agents that wrote this book run on Claude, and the book discusses Anthropic's products, research and data alongside everyone else's. Chapter 19, the one vendor-specific chapter, was added at the director's request and is held to the same sourcing rule. For all vendors: primary sources, vendor-reported figures labeled as such, and no product recommendation without a decision rule behind it. Agents have no lived experience, so this book does not pretend to: stories are either cited public incidents or clearly labeled composite cases. Where the human director adds his own experience as a product builder working on a platform of around 100 microservices and an internal RAG assistant, it appears in a box marked "Yann's note". This edition has one, in Chapter 13, drafted from his interview answers and approved by him; more may come in later editions.

Online extras

Free on the website ([SITE URL]):

- Field notes 2026: the full dated digest behind Chapter 20, with a source and an evidence label on every entry.
- Appendix B, the tool and MCP server design checklist, and Appendix C, the threat model template, each with a PDF.
- For every chapter: the questions readers ask, a self-check quiz and every source.
- The glossary.

How this edition stays current

This is a living edition with free updates through the end of 2027. Fast-moving claims are marked "(as of late 2026)" so you can see what to re-check.

Copyright and notices

Agents in Production: The practitioner's playbook for agentic frameworks, MCP, skills and AI automation. 2026 to 2027 edition. Ebook edition, 2026.

Copyright © 2026 Yann. All rights reserved. No part of this book may be reproduced, stored or transmitted in any form without written permission from the publisher, except for brief quotations in reviews, and the checklists and templates, which readers may copy for use inside their own organization.

AI authorship. Anthropic did not commission, review or endorse this book. Protection of AI-generated text varies by jurisdiction; rights are claimed in the selection, arrangement and human-authored contributions.

Disclaimer. This book is for information and education. It is not legal, financial, tax, security or other professional advice. Facts, prices, versions and regulations were checked as of November 2026 and change quickly; verify them before you act. The author and publisher accept no liability for decisions made on the basis of this book.

Trademarks. Product and company names (Claude, ChatGPT, Gemini, Copilot, n8n, Zapier, Make and every other one in this book) are trademarks of their owners. They are used only to identify products, and no affiliation or endorsement is implied.

ISBN (ebook): to be assigned. Published by [Publisher name].

Set in Bricolage Grotesque, Source Serif 4 and JetBrains Mono, all under the SIL Open Font License 1.1. Figures drawn by the agent team.

Found an error? Every correction makes the next edition better: [errata contact to be added].

1

What an agent actually is

A definition precise enough to design, budget and argue with

Builder Leader Solo Ops Claude

Ask five people what an "agent" is and you get five answers, each implying a different budget, risk profile and team.

Four parts and a harness

An agent is **a model that runs in a loop, choosing tools, with a managed context, inside a harness**. Strip any of the four parts and you have something else. Strip the harness and you have a demo.

The model

The model is the decision maker: on each turn it reads the context and returns a final answer or a tool call. It does not "remember" or "act"; your code does, when the model asks. When a run fails, find the first step in its trace that went off course and ask, in order:

1. **Was the needed information in the context?** If not, context failure: fix retrieval or context management.
2. **Was it there, but the model chose the wrong tool or arguments?** Model failure: fix instructions, tool descriptions or model choice, and add the case to your eval set.
3. **Was the call sensible, but the tool misbehaved?** Software failure: fix the code, add a unit test.

This needs every model and tool call traced with its inputs.

Choose the model as a pair: model and reasoning effort. Launch tables report maximum effort; shipped defaults are often lower (Chapter 19 has a dated example). Pin both values in config and compare pairs on cost per successful task.

The tools

A tool is a name, a description, a parameter schema and a function the model can ask you to run. Most risk lives here: a read-only search and a "refund customer" tool look alike in a schema and are not the same kind of object (Chapter 3).

The loop

Send the context to the model; if it returns a tool call, validate it, run it, append the result; if it returns a final answer or a stop condition is hit, exit; otherwise repeat. This is usually called ReAct, after the 2022 paper, and it is the default loop in most SDKs.

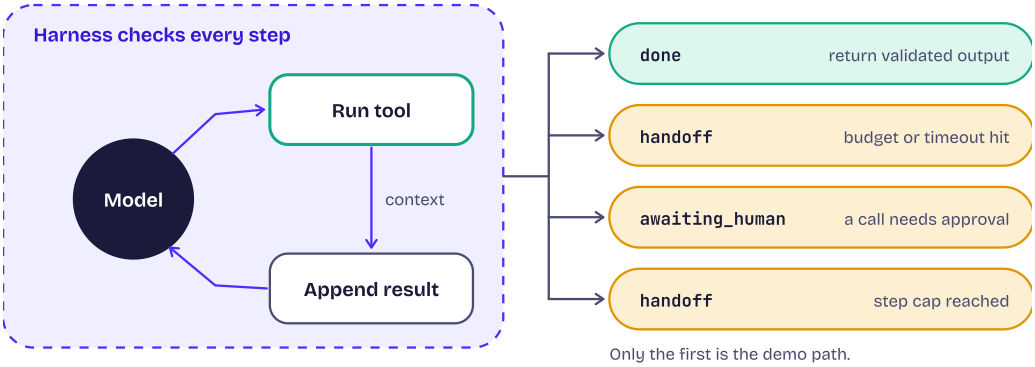
The context

The context is everything the model sees on a turn: instructions, tool definitions, retrieved documents, history, tool results, memory. Recall degrades as it grows, a pattern Chroma's research named **context rot** (Chroma, Jul 2025); managing it is what Anthropic calls context engineering (Anthropic, 2025-09-29), so a loop that appends every result forever gets worse as it runs. Memory is not a fifth part: it is a source of context, read in by the loop and written through tools (Chapter 5).

The harness

Three statuses, four ways a run ends

A demo handles one exit. A production agent treats all four as normal outcomes.



Three statuses, four exits. Done is the demo path; a handoff (budget, timeout or step cap) and awaiting a human are normal outcomes the harness must handle.

> **Strip the harness and you have a demo.**

The harness is the production wrapper, with four jobs:

- **Budget**: caps on steps, tokens, money and wall-clock time.
- **Guardrails**: allowed tools, tools that need human approval, blocked inputs and outputs.
- **State**: checkpoints so a run can resume after a crash, and durable waits for approval.
- **Telemetry**: traces of every model and tool call, the data your evals run on.

Framework defaults are a start, not a design (Chapter 6). The whole thing in one sketch; `call_model` and `run_tool` are placeholders for your model client and tool registry. This is the shape; Chapter 8 has the version to copy.

```
import time

MAX_STEPS, MAX_COST_USD, TIMEOUT_S = 10, 0.50, 120

def run_agent(task: str, tools: dict, ctx: list) → dict:
    ctx.append({"role": "user", "content": task})
    cost, start = 0.0, time.monotonic()
    for step in range(MAX_STEPS):
        reply, call_cost = call_model(ctx, tools)
        cost += call_cost
        ctx.append(reply.as_message())
        trace("model_call", step=step, cost=call_cost)
        # done only on a validated submit_* call (Chapter 8)
        if reply.submit:
            return {"status": "done", "output": reply.submit}
        if cost > MAX_COST_USD:
            return {"status": "handoff", "reason": "budget"}
        if time.monotonic() - start > TIMEOUT_S:
            return {"status": "handoff", "reason": "timeout"}
        if not reply.tool_calls:  # free text is not done
            return {"status": "handoff", "reason": "no submit"}
        # approval pauses the whole turn (Chapter 8)
        pending = [c for c in reply.tool_calls
                   if needs_approval(tools, c)]
        if pending:
            checkpoint(ctx)
            return {"status": "awaiting_human", "pending": pending}
        for call in reply.tool_calls:
            tool = tools.get(call.name)
            if tool is None:
                ctx.append(tool_error(call.id, "unknown tool"))
                continue
            ctx.append(safe_run(tool, call))
        checkpoint(ctx)
    return {"status": "handoff", "reason": "step cap"}

def safe_run(tool, call):
    try:
        return tool_result(call.id, run_tool(tool, call.args))
    except Exception as e:
        return tool_error(call.id, f"{type(e).__name__}: {e}")
```

A run ends in one of three statuses: done, a handoff with its reason (budget, timeout, no submit, step cap), or awaiting a human. A demo has one. A production agent treats the other two as normal outcomes. On "awaiting human", persist the checkpoint and the pending calls under a run ID; for long waits, use durable execution rather than a held worker (Chapter 7).

Anti-pattern

The harness as an afterthought

The common pilot has no step cap, no cost cap and no traces, "until launch". By then prompts were tuned against unbounded runs, and a 10-step cap breaks tasks that quietly took 25. Put the harness in the first commit with generous numbers, and tighten them with data.

Choosing the first numbers

1. **Start generous**: step cap about three times what a good run needs.
2. **Run your eval set**: 30 to 50 real tasks; record steps, cost, outcome.
3. **Set the step cap from successful runs only**, just above the slowest (around the 95th percentile). Longer runs are usually loops or tasks for a human.
4. **Derive the cost cap from value**: one attempt must cost well below what a success is worth (Chapter 14).
5. **Set the timeout from the caller**: seconds for chat, minutes for a background job.

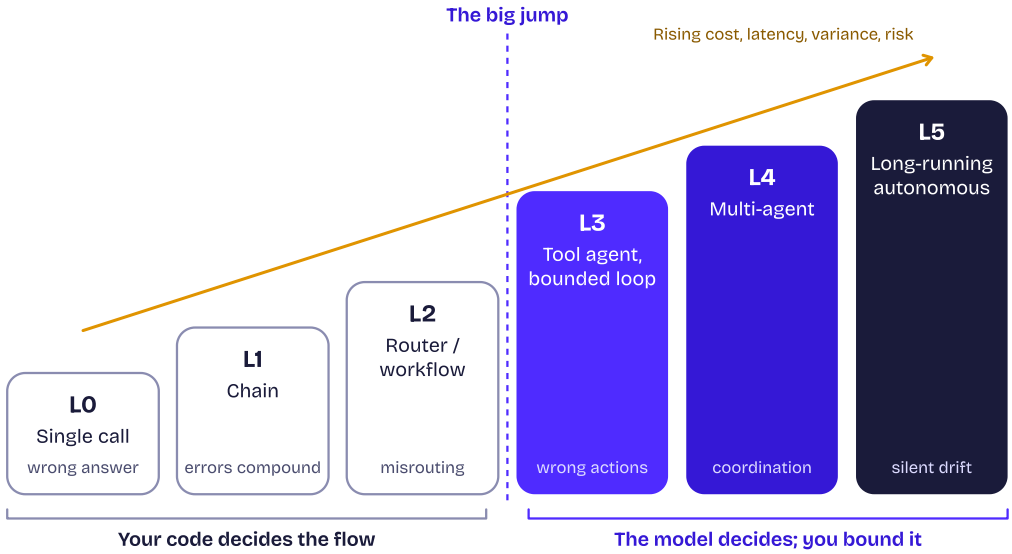
A per-run cap stops one run, not a bug that triggers thousands; add a daily spend limit and per-user rate limits too (Chapters 8 and 10).

The autonomy ladder

Each step up adds capability, and also cost, latency, variance and risk.

The autonomy ladder

Each rung adds capability, and adds cost, latency, variance and risk



Climb one rung at a time. Below L3 your code decides the flow; from L3 up the model decides and you can only bound it.

Level	Name	Who decides the next step	Model calls per task	Example	Main risk
L0	Single call	Your code	1	Classify a ticket	Wrong answer on one input
L1	Chain	Your code, fixed sequence	Fixed	Extract, draft, check tone	Errors compound
L2	Router / workflow	Your code, model picks branches	Small, known per branch	Route a ticket to billing, tech or sales	Misrouting
L3	Tool-using agent, bounded loop	The model, inside your caps	Variable, up to the cap	Investigate a ticket with search and CRM, max 10 steps	Wrong actions, loops, cost spikes
L4	Multi-agent	Several models, delegating	Sum of every loop	Orchestrator splits research across workers	Coordination failures
L5	Long-running autonomous	The model, over hours or days	Open-ended	Agent that works a backlog	Drift, unbounded spend

Up to L2 you can price a task before you ship. From L3 you only know the ceiling, and because each step re-sends the growing context, cost rises faster than the step count.

- **Climb one rung at a time.** A well-built L2 router beats a fragile L3 agent on almost every metric finance cares about.
- **L2 to L3 is the big jump.** Below L3 your code decides the control flow and you can draw it. From L3 the model decides and you can only bound it: you need trajectory evals, and the harness stops being optional.
- **L4 and L5 multiply everything**: calls, context and hiding places for errors (Chapter 2).

Decision rule

Which rung to start on

Start at the lowest level that can plausibly do the task. Steps you can write down in advance: L1 or L2. Steps that depend on what you discover: L3 with a step cap you can justify. Never start at L4 or L5; let L3's failures tell you what extra agents must fix.

Where real deployments sit

The most useful public evidence as of late 2026 is the UC Berkeley-led study "Measuring Agents in Production" (MAP): 20 case studies plus a survey of 306 practitioners, 86 of whom reported on agents in production or pilot (arXiv).

- **Short leashes (survey).** 68% of deployed agents run 10 steps or fewer before a human intervenes; 47% fewer than 5.
- **Humans judge (survey, 31 respondents to this question).** 74% depend primarily on human evaluation; 52% use LLM-as-judge.
- **Custom code (case studies).** 17 of 20 teams built on direct model API calls, while 61% of survey respondents use a framework.
- **Weak measurement (case studies).** 75% evaluate without formal benchmark sets.

Most production agents are L2 workflows or tightly capped L3 agents with a human close by. Only one of the 20 case studies let its agent explore without constraints, inside a sandbox with strict CI checks. If a first project promises L5, ask for the L3 version with a step cap, a review queue and a date for eval results.

Agent, automation, chatbot, copilot: vocabulary that ends meetings

The test is who decides the next step and who acts.

How we got here

Agents were defined long before LLMs

The word "agent" has a research history older than most of today's frameworks. In 1995, Michael Wooldridge and Nicholas Jennings proposed a "weak notion of agency" with four properties: autonomy (acting without direct human intervention), social ability, reactivity to a changing environment, and pro-activeness, meaning goal-directed behavior on the agent's own initiative (Wooldridge and Jennings, 1995). What changed after 2023 is that a general-purpose language model can fill the decision-making role, so the old definition finally describes software that teams can ship.

Lesson for agents: define agency by observable behavior (autonomy, reaction, initiative), not by which model sits inside.

Term	Who decides the next step	Who acts	Example
Automation	Your code (rules or flowchart)	Your code	A workflow with an LLM step that classifies emails
Chatbot	The user, turn by turn	The model answers; nothing else happens	FAQ assistant over documentation
Copilot	The user, with model suggestions	The user accepts or rejects	A draft reply the support agent edits
Agent	The model, within bounds	The model, through tools, with approvals	Investigates a ticket and proposes a resolution

- An automation with an LLM call inside is still an automation. Calling it an agent raises expectations and lowers rigor.
- A chatbot becomes an agent the moment it can call tools that change something, or chain tool calls on its own.
- A copilot is often the best first version of an agent: every accept or reject becomes eval data.
- "Autonomous" is a rung, not a type. Say "L3 with approval on writes". An "AI agent" node in n8n or Zapier that loops over tools is L3, whatever the canvas looks like (Chapter 13).

Composite case: the assistant that became an agent

A mid-size software company built an internal documentation assistant: fixed retrieval, cited answers, L1. Questions like "why did this customer's import fail yesterday" led the team to add tools one at a time: a second document source, an ownership lookup, a read-only pipeline-status query. After the third, the system chose which tools to call and in which order.

Nobody had decided to build an agent; the team had one anyway, with the harness of a chatbot. A question that triggered a long chain of rephrased searches was slow, expensive and invisible until each step was traced. The fixes were a step cap, a per-question budget, traced tool calls, read-only credentials, and an eval set from questions that had gone wrong.

In production **Notice the moment you cross L2**

The move from chatbot to agent rarely happens in a design review; it happens one tool at a time. Make the second tool added to any assistant trigger a harness review: caps, tracing, credential scope, eval set.

Checklist

- [ ] Every AI feature has its autonomy level (L0 to L5) written down.
- [ ] Every agent has a step cap, cost cap and timeout in config.

- [ ] Every agent handles three statuses: done, handoff with a reason, awaiting human.
- [ ] Every model call and tool call is traced, not only the final answer.
- [ ] Write tools require approval or have a documented reason not to.
- [ ] Adding a tool to an existing assistant triggers a harness review.

Chapter 1 on one page

What an agent actually is

An agent is a model in a loop, choosing tools, with a managed context, inside a harness. Name its rung on the ladder before you build it.

Key takeaways

- The harness (budget, guardrails, state, telemetry) is what makes an agent production-ready
- L2 to L3 is the big jump: the model takes over the control flow
- Real deployments are short and supervised; systems become agents one tool at a time

Do this on Monday

- Pull one failed run's trace and find the first step that went off course
- Count steps on your last 30 successful runs and set the cap just above the slowest
- Ask how many tools your oldest assistant has now, and who reviewed the last one added

68%
of deployed agents in the survey run 10 steps or fewer before a human

17 of 20
case-study teams built on direct API calls (61% of survey respondents use a framework)

2

Workflows before agents

The cheapest, most reliable agent is often the one you did not build

Builder Leader Solo Ops Claude

Composite case: a software company spends two weeks building four cooperating agents to triage support tickets, at forty seconds per ticket, until someone draws the real process on a whiteboard: five boxes, two branches. Rebuilt in an afternoon as a workflow with two model calls, it is faster, cheaper and easier to test.

A **workflow** is a system where your code decides the sequence of steps, even if some steps call a model; in an agent (Chapter 1), the model does. The difference is who controls the flow.

The core rule: if you can draw the flowchart, build the workflow

If you can draw the flowchart before the task starts, build the workflow. Use an agent only for the parts where the next step depends on what you discover along the way.

How we got here

Workflows have been executable for fifty years

Composing small, predictable steps is an old idea. Unix pipes, credited to Doug McIlroy and implemented by Ken Thompson (Bell Labs, Unix history), first appear in the Version 3 Unix manual of February 1973 (Oral history of Unix). BPMN 2.0 later made process diagrams executable (Prologue). Each generation learned that an explicit flow is easier to test, audit and change than behavior hidden inside one clever component.

Lesson for agents: a model call is one more stage in a pipeline; keep the pipeline explicit wherever you can draw it.

In a workflow, every path is code you wrote: testable, logged, priced, explainable. In an agent, the model chooses the path at runtime: you can bound it, not enumerate it. That trade pays only when you need the flexibility. Anthropic's "Building effective agents" draws the same line and advises starting with the simplest solution, adding complexity only when needed (Anthropic, 2024-12-19). Most production "agents" are workflows with a short agentic section, and that is the healthy shape.

To test whether you can draw it, take five real inputs and write down the steps a competent human follows for each. Same steps, or two or three known variants: a workflow. A different path through different systems every time: maybe an agent.

Anti-pattern

Building the agent to find out what the workflow is

Teams that do not know the process yet build an agent, hoping the model will figure it out. It will, differently each time, and you spend weeks reverse-engineering traces. Interview the human who does the task today, draw the flowchart, build that. Keep the agent for the boxes labeled "it depends".

The pattern catalog

In the first four patterns your code owns the flow. Orchestrator-workers sits on the border. The last two are agents. The first five definitions follow Anthropic's guidance.

Pattern	Shape	Use when	Avoid when	Relative cost	Typical failure
Prompt chaining	Fixed sequence of calls with code checks between them	Clear steps (extract, draft, check)	Steps depend on each other unpredictably	Low	Early error flows into later steps
Routing	Classify, send to a specialized path	Distinct input categories	Categories overlap or keep changing	Low	Confident misrouting
Parallelization	Independent calls at once (sectioning) or same call N times (voting)	Independent subtasks, high-stakes judgments	Shared state	Medium, N times the calls	Votes agreeing on the same wrong answer
Evaluator-optimizer	Generator plus critic in a loop	Checkable quality criteria	First draft is good enough	Medium to high	Endless polishing
Orchestrator-workers	One model decomposes and delegates	Subtasks unknown in advance	You can list the subtasks	High	Bad decomposition
Agent loop (ReAct)	Reason, call a tool, read, repeat	Path depends on tool results	You can draw the flowchart	Bounded only by caps	Loops, wrong tools, invented parameters
Plan-and-execute	Write a plan, execute, revise	Long tasks where a visible plan helps review	Short tasks	Medium to high	Following a bad plan faithfully

What the table does not say:

- **Chaining** is reliable only because of the gates: plain code between calls checks schema, required fields and length.
- **Routing** is your main cost lever (small models for easy cases). Always give the router an "unsure" category that goes to a human or a stronger path; a router forced to choose will choose confidently and wrongly.

- **Voting** detects uncertainty cheaply: disagreement between runs is a signal.
- **Evaluator-optimizer** needs a stop condition and an iteration cap. The critic always finds something.
- **The agent loop** (named after the ReAct paper) needs a full harness. Arize describes agents that keep polling a job's status instead of waiting, turning one task into hundreds of API calls while every log line looks healthy (Arize).
- **Plan-and-execute** gives humans a plan to review before anything runs: a natural approval point.

The decision tree: single call, workflow, agent, multi-agent

Walk this tree for every new use case. Stop at the first "yes".



> **Stop at the first yes.** Each box further down costs more and is harder to test, so it needs a failing eval case to justify it.

Step 4 asks about contexts, permissions and parallelism, not "roles": splitting does not pay because the task "feels like" a team. Default to the highest box that passes your evals, and move down only with a failing eval case that better prompts, context or tools cannot fix. Write that case down: it justifies the extra cost.

The determinism budget: where to put code and where to put the model

Every LLM call adds variance, and wrong outputs can look right. Spend that variance only where the model buys something code cannot.

> **The model proposes, the code disposes.**

Put in code (computable right answer)	Put in the model (language and judgment)
Permissions and policy: who sees what, which refunds need approval	What is this message about, how urgent does it sound?
Arithmetic, dates, SLA deadlines	Which of these documents answers the question?
Lookups by known keys: tier, contract, owner	Draft a reply in the right tone
Routing on structured fields, validation, retries, idempotency	Summarize this thread for a human

The model proposes, the code disposes. The model returns structured output (a category from an enum, a draft); code validates it, applies business rules and decides. A model that says "priority: P1" for a free-plan customer does not override the rule that free plans have no P1 queue. The test: "if the model gets this wrong, would a unit test have caught it?" If yes, it belongs in code. "Refunds above 500 euros need manager approval" is a rule, not a prompt line: prompts are not enforcement.

Worked example: support ticket triage, built three ways

The task. A B2B SaaS company receives support tickets by email and chat. For each one the system assigns a **category** (billing, technical, access, feature, other), a **priority** from P1 (outage for a paying customer) to P4 (question), stored as an integer 1 to 4, a **queue**, and optionally a **draft reply** from the knowledge base. P1s must reach on-call within minutes, wrong P1s wake people at night, and no reply goes out without a human. Available: a CRM, a searchable knowledge base, a status page of open incidents.

Version A: the workflow

Five boxes: classify, look up the customer, apply priority rules, route, draft. Two need a model.

```
CATEGORIES = {"billing", "technical", "access", "feature", "other"}

def workflow_triage(ticket: Ticket) → Triage:
    # Model call 1: small, fast model, strict JSON schema
    # (category, urgency_signal, confidence).
    c = classify(ticket.text, schema=ClassifySchema)
    if c.category not in CATEGORIES or c.confidence < 0.6:
        return to_human(ticket, reason="low_confidence")

    # Code: lookups by key.
    customer = crm.get_customer(ticket.sender_email)
    if customer is None:
        return to_human(ticket, reason="unknown_sender")
    incident = status.open_incident_for(customer.region)

    # Code: business rules return an int, 1 (outage) to 4.
    inputs = PriorityInputs(
        category=c.category, urgency=c.urgency_signal,
        tier=customer.tier, open_incident=incident is not None,
    )
    priority = priority_rules(inputs)
    queue = QUEUE_MAP[(c.category, priority)]

    # Model call 2, only where a draft helps.
    draft = None
    if c.category in {"billing", "access"}:
        docs = kb.search(ticket.text, top_k=4)
        draft = draft_reply(ticket.text, docs, customer.name)

    return Triage(c.category, priority, queue, draft,
                  confidence=c.confidence, inputs=inputs,
                  needs_review=True)
```

At most two model calls per ticket. Priority comes from rules, so a P1 always means the same thing, and every branch is testable with unit tests plus labeled past tickets (100 to 200 across categories, with rare costly cases such as real P1s over-represented). You can price it before writing code: tickets times tokens per ticket times price.

Where the confidence number comes from. A confidence the model writes into its own JSON is a hint, not a probability; the 0.6 is illustrative. Group labeled tickets by stated confidence, measure accuracy per group, and set the threshold where accuracy meets the business bar, probably per category. If stated scores do not track accuracy, use agreement between two or three runs, or token log-probabilities where your provider exposes them.

What it cannot do: investigate. "The export has been broken since your release on Tuesday, see attached log" gets a category and a priority, but nobody checks whether Tuesday's release touched exports.

Version B: the agent

One model in a bounded loop with tools `search_kb`, `get_customer`, `get_open_incidents`, `search_recent_tickets`, `read_attachment` and `submit_triage`. The prompt holds categories, priority policy and queue map; the harness caps the run at 12 steps and a fixed cost; every tool except `submit_triage` is read-only.

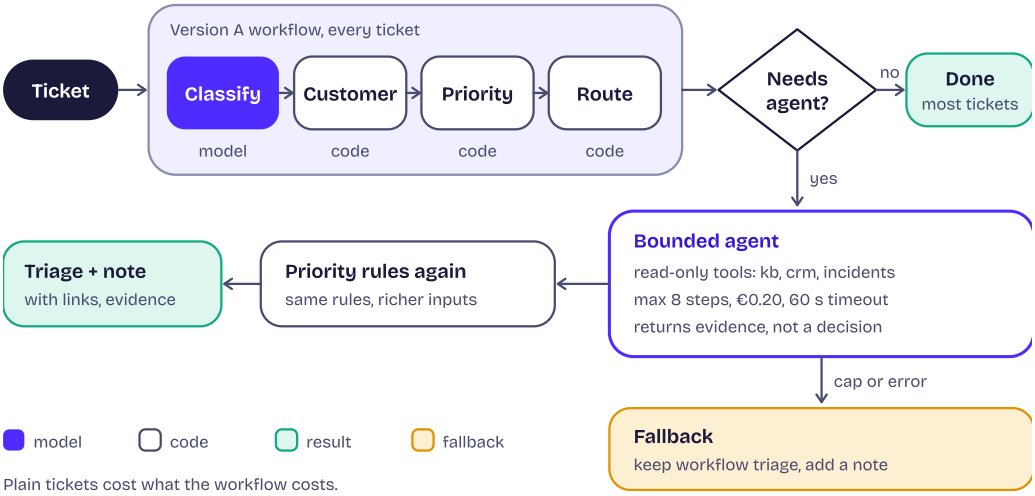
On the Tuesday export ticket, the agent reads the log, finds three similar recent reports, notices an open incident, and proposes a P2 with links: what a senior support engineer does. The cost: 3 to 12 model calls per ticket, each re-sending a growing context, with variable latency. Priority is now the model's reading of a prose policy, so similar tickets can get different priorities, and tests must cover trajectories. On a plain "how do I change my invoice address?" ticket, the agent still searches and checks. Most tickets are plain.

Version C: the hybrid

Run the workflow for every ticket. Send only tickets that need investigation to a bounded agent, and let the agent propose, not decide.

The hybrid: a deterministic spine with a bounded agentic section

The agent proposes evidence. Code applies the rules and has the last word.



Plain tickets cost what the workflow costs.

> **Spine first, agent where it depends.** Every ticket runs the workflow; only tickets that need investigation reach a bounded, read-only agent, and code still sets the priority.

```
from dataclasses import replace

def triage(ticket: Ticket) → Triage:
    # Version A, unchanged.
    base = workflow_triage(ticket)
    needs_investigation = base.category == "technical" and (
        base.priority ⩽ 2 or ticket.has_attachment
        or base.confidence < 0.8
    )
    # Most tickets end here.
    if not needs_investigation:
        return base

    result = run_agent(
        task=Task(id=ticket.id,
                  input=investigate_prompt(ticket, base)),
        model=SMALL_MODEL,
        # kb, crm, incidents, tickets
        tools=READ_ONLY_TOOLS, flags=FLAGS,
        budget=Budget(8, 0.20, 60),
    )
    # A handoff keeps the workflow triage, with the reason.
    if result.status ≠ "done":
        return replace(base, note="investigation incomplete: "
                       + result.reason)
    # A validated Finding: the submit_finding argument.
    finding = result.output

    # Keep only evidence that points at real records
    # (ticket, incident, log line).
    evidence = verify_evidence(finding.evidence)

    # The agent proposes; code applies the same rules
    # to richer inputs.
    inputs = replace(base.inputs, evidence=evidence)
    p = priority_rules(inputs)
    return replace(base, priority=p, inputs=inputs,
                   queue=QUEUE_MAP[(base.category, p)],
                   note=finding.summary, links=finding.links)
```

`verify_evidence` guards against the nastiest failure: evidence that is wrong but plausible. The agent must cite each claim with a record ID; code fetches each record and drops claims whose ID does not exist or does not match (an "open incident" that is closed). Unverified hunches can appear in the summary but never move a ticket to P1.

Plain tickets cost what the workflow costs, and a P1 still means what the rules say. If the agent fails or runs out of budget, the ticket keeps its workflow triage with a note: graceful degradation, not an outage.

Tuning the trigger. `needs_investigation` decides both cost and coverage. Log it on every ticket and watch two weekly rates: **missed investigations** (not investigated, later escalated by a human: too tight) and **wasted investigations** (the finding changed nothing: paying for no value). Change one condition at a time.

The comparison

These are uncached design estimates, not benchmarks. Call counts understate the gap: a classification reads one ticket, while each investigation step re-sends a growing context. With illustrative token counts (about 1,200 input tokens per classification and about 2,800 per ticket for Version A once drafts are counted; a six-step investigation growing from 5,000 to 11,000 tokens, about 48,000 in total), Version C costs about 1.85x Version A when 5% of tickets are investigated, and about 4.4x at 20%. The trigger, not the model, sets the bill.

Google's illustrative ADK 2.0 refund example on mock APIs points the same way: deterministic nodes cut tokens per run from 5,152 to 2,265 and time from 7.2 to 5.7 seconds against a plain LLM agent (Google Developers Blog, Jul 2026, vendor claim).

Dimension	A. Workflow	B. Agent	C. Hybrid
Model calls per ticket	1 to 2, fixed	3 to 12, chosen by the model	1 to 2 for most; up to 2 + 8 when investigated
Relative model cost	1x	1.5x to 12x by call count, more with context growth	About 1.85x at 5% investigated, 4.4x at 20% (illustrative)

Dimension	A. Workflow	B. Agent	C. Hybrid
Latency	Low, predictable	Variable, capped	Low for most tickets
Priority consistency	Deterministic	Varies between similar tickets	Deterministic, richer inputs
Novel, multi-system tickets	Human investigates	Yes	Yes, for routed tickets
Behavior on failure	Human queue	Hits a cap; no triage without a fallback	Falls back to workflow triage
How you test it	Unit tests plus labeled set	Trajectory and outcome evals, human review	Workflow tests plus a small trajectory set

The verdict

Default to Version A: a working system, a labeled dataset and a baseline cost within days. Ship Version C when enough tickets that A handled badly for lack of investigation exist to build an eval set. Ship Version B only if most tickets need investigation, volume is low, and you accept that priorities become judgments.

The hybrid is the general shape of good agent systems: a deterministic spine for predictable work, bounded agentic sections where discovery pays, and code with the last word on decisions that matter. Chapter 17 takes this system to production.

Decision rule

Hybrid placement rule

Put the agent where the flowchart says "it depends", give it read-only tools and a small cap, make it return evidence rather than decisions, and let code apply the rules. If its output must be acted on directly (a write to a customer record), put a human approval or a deterministic validation between the agent and the action.

Anti-patterns

- **The agent for a cron job.** A nightly report built as an agent rediscovers the plan every night, skips a metric as "not relevant today", and reports stop being comparable. Fetch and compare in code, then one model call writes the summary.
- **Multi-agent theater.** A "CEO agent" delegates to "manager", "analyst" and "QA" agents. Each hand-off loses context, costs multiply per hop, and nobody can tell which agent caused a wrong output. Usually a prompt chain wearing costumes: apply step 4 of the tree.
- **Agent as error handler.** Wrapping a brittle integration in an agent "so it can figure out what went wrong". Fix the integration: agents sometimes read an HTTP 500 with no explicit failure text as "processed" (Arize).
- **Same big model everywhere.** Classification and extraction usually run fine on a small model. Route by task and evaluate each route.

Per-track notes

- **Builder.** Build Version A first, even if you are sure you will need an agent. Keep the workflow-agent interface typed, with Chapter 8's three statuses: done, handoff, awaiting human.
- **Leader.** Use the decision tree as a review gate: every agent proposal names the step it fails, with an example. Ask for cost per successful task and what happens at the cap.
- **Solo.** Workflows fail loudly and predictably, so you debug fewer evenings. Keep each step's cost plus your review time in a spreadsheet: that sum, on the retry case, is your pricing floor (Chapter 13).
- **Ops.** n8n, Zapier or Make flows are already workflows: routing is a switch after a classification step, chaining is LLM steps with filters, an IF node on the confidence field sends low scores to a human queue, the investigation is an AI agent node behind a condition, with read-only tools, a low iteration limit and an approval step before any customer message.

Checklist

- [ ] We drew the flowchart from at least five real inputs before choosing an architecture.
- [ ] We wrote down which step of the decision tree our design fails, with an example.
- [ ] Every decision with a computable answer is in code, not in a prompt.
- [ ] Every model output is validated, and every classifier has an "unsure" path.
- [ ] Every agentic section has caps, a timeout and read-only tools unless writes are approved, and falls back to a workflow result or a human.
- [ ] We have a workflow baseline for cost and quality to compare any agentic design against.

Chapter 2 on one page

Workflows before agents

If you can draw the flowchart, build the workflow. Use an agent only for the boxes labeled "it depends".

Key takeaways

- Walk the decision tree from the top and stop at the first yes
- The model proposes, the code disposes
- Ship the workflow first; move to the hybrid when real tickets show the need

Do this on Monday

- Sit with the person who does the task today for an hour and note every "it depends"
- Search your prompts for thresholds, amounts and "must": each is a rule to move into code
- Price the workflow version: tickets times tokens per ticket times price

1 to 2
model calls per ticket, workflow triage (design estimate)

3 to 12
model calls per ticket, agent triage (design estimate)

Keep going: the full book

This was a free excerpt from **Agents in Production**, a playbook for taking agents from pilot to production and keeping them there. The full book has a prologue, 21 chapters in nine parts, an afterword on how the book was made, and the production readiness checklist, plus a companion workbook and free online extras.

The full book covers:

- **Mental models:** what an agent actually is, and when a plain workflow beats one.
- **Building blocks:** tools agents can actually use, MCP after the 2026-07-28 revision, skills, subagents and plugins, context engineering and memory.

- **Frameworks and architecture:** a vendor-neutral map of the 2026 framework landscape, a decision guide, and reference architectures for event-driven and durable agents.
- **Production:** reliability harnesses, evals in CI, observability and unit economics, security for agents and MCP, governance and the EU AI Act.
- **Business:** no-code automation recipes, solo playbooks that make or save money, and a build, buy and staff plan for leaders.
- **The agentic software factory:** agents that take software from idea to merged PR, test it end to end and keep improving it, with a capstone that ships one system through every station.
- **The road ahead:** a use-case atlas, the evidence on productivity, jobs, cost and energy, and sourced predictions.
- **Appendix:** the production readiness checklist (59 checks, 19 must pass). Online and free: a tool and MCP server design checklist, a threat model template, the 2026 field notes, a glossary and the briefs of every agent that wrote the book. The companion workbook holds the exercises and their solutions.

Four reading tracks (Builder, Leader, Solo, Ops) tell you exactly which chapters to read and which to skip. Exercises, a running project and self-checks come with full solutions.

Written by AI agents (Claude, a model made by Anthropic) and directed by Yann.

Get it at agentinprod.com