AI Product Judgment

Version 2026 · verified 2026-11-18

Decide what AI should build, how users verify it, when it may act alone, and whether it pays off — problem framing, evals, cost and the business case.

Contents

Mental model 2

Start from the problem 2

Design for verification 4

Build model intuition 5

Trace a wrong answer 7

Choose autonomy deliberately 8

Evaluate before you trust 10

Cost, latency, reliability 11

Make the business case 12

Prove it with a portfolio 14

Further reading 15

Mental model

When anything can be prototyped in an afternoon, “can we build it?” stops being the question and “should we, and how will we know it works?” becomes the whole job. An AI product manager is not paid to use AI; they are paid for decisions about which problem deserves a model, how people check its output, how much it may do alone, and whether quality, cost and reliability hold at scale. Every skill below is a way of making one of those decisions with evidence instead of a demo.

Start from the problem

Step	Output
Watch the work	The real bottleneck, not the assumed one
Name the user	One role, one job, one pain
Set baselines	Today’s numbers, measured or “TBD”
Set targets	One goal, three or four metrics

Step	Output
Only then pick tech	Model, retrieval, agent or none

Hearing “support spends four hours a day on policy questions” and proposing a chatbot skips discovery. Shadowing the agents may show the bottleneck is *finding* the right policy across scattered documents, not writing the reply — which points to a copilot that finds, cites and drafts, not an autonomous bot. Write the goal as an outcome, then attach metrics with honest baselines.

```
Goal: halve time on policy questions,
  without losing accuracy
Time on policy Qs   4 h/day -> 2 h/day
Avg resolution      8 min   -> 4 min
Material errors     5%      -> < 1%
Sent w/o big edits  TBD     -> 80%
```

> **Gotcha:** A problem that a checklist, a search box or a manual step solves cheaply does not need a model. “Could AI do this?” is almost always yes; “should it?” is the decision you own.

Design for verification

Pattern	Why
Cite the source	User can check in one click
Draft, don’t send	Human edits before it counts
Show uncertainty	“Not found” beats a guess
Cheap correction	Edit, dismiss, retry inline
Explain the action	What it will do, and why

AI UX is the design of how people work alongside an output that is sometimes wrong: how they see what the system can do, verify a result, fix a mistake and stay in control. The goal is **calibrated trust** — users rely on it where it is good and check it where it is not. Both the Google PAIR guidebook and Microsoft’s human-AI

guidelines centre on this: set expectations, make correction cheap, and explain behaviour.

```
Copilot flow
1. Agent pastes customer question
2. Copilot finds policy, drafts reply
3. Draft shows the cited passages
4. Agent edits, approves, sends
```

> **Gotcha:** A fast answer that takes the user three minutes to verify is slow. Measure the whole task — question to sent reply — not model response time.

Build model intuition

Term	Product consequence
Token	Unit of cost and of context size
Context window	Max input per call; fitting ≠ using
Tool calling	Model asks; your app runs the tool
Latency	Better models are often slower
Inference cost	Model × tokens × number of calls

Model intuition is knowing where a model works well and where it falls over, and you only get it by testing on your own tasks. Run candidate models on the same set of real questions and keep the **failures** next to the scores: a missed exception, two documents merged wrongly, a tool call where a clarifying question was needed. The failure pattern chooses the model better than one accuracy number.

```
50 support questions, 3 models
M1 fast, cheap   misses key facts
M2 accurate      slower, pricier
M3 good simple   fails multi-doc
Decision: route, don't pick one
```

> **Tip:** Try the bare model before adding custom instructions and plugins. Seeing where it fails unaided tells you what your context actually has to supply.

Trace a wrong answer

Failure	Question	Typical fix
Retrieval	Right document found?	Index, ranking, filters
Context	Right passage reached it?	Chunking, assembly
Generation	Misread good input?	Prompt, conflicts, model

Prompting is the instructions, context engineering is getting the right information and tools in front of the model, and RAG is fetching documents into that context. A wrong answer — “60-day returns” when policy says 30 — can start at any of the three stages, and each needs a different fix. Log every step so you can follow the failure back to where information was lost.

```
{
  "question": "Return after 30 days?",
  "retrieved": ["returns-2023.md#p2"],
```

```
  "context_sent": "…60 days…",
  "tool_results": [],
  "answer": "You have 60 days."
}
```

> **Gotcha:** “Fix the prompt” is the default reflex and often wrong. In the trace above the model faithfully repeated an outdated document; no prompt change fixes a retrieval failure.

Choose autonomy deliberately

Shape	Use when
Single model call	One step, no tools
Fixed workflow	Steps predictable, order known
Agent	Path depends on what it finds
Approval gate	Action is costly or irreversible

Anthropic’s agent guide draws the line: a **workflow** runs model calls along a path you wrote; an **agent** decides its

own next step with tools. Prefer the simplest shape that works — workflows are cheaper, faster and easier to debug. Let the agent research and prepare; keep consequential, state-changing actions behind a human. Revisit the choice as models improve: Cursor's code reviewer shipped as a fixed multi-pass pipeline and later moved to an agentic design.

```
Refund $500 requested
1. Agent checks order + refund policy
2. Policy says: needs manager approval
3. Pause action; open approval request
   (order, policy, recommendation)
4. Manager approves -> refund, log
   Manager rejects  -> no refund, log
```

> **Warning:** Retries plus a non-idempotent action means double refunds. Cap retries, and make every state-changing tool safe to call twice.

Evaluate before you trust

Case type	Expected behaviour
Answerable	Correct, cited answer
Not in the docs	Says so; no guess
Outdated or conflicting	Uses current, flags conflict
High risk	Escalates to a human

An eval is a fixed set of cases, each with defined success, run on every change so you can tell whether a change helped. Before picking metrics, read real outputs and group the failures — that gives you grounded categories. Use code checks where you can (“approval existed before refund”) and expert review where you cannot; if a model grades, calibrate it against people who know the task.

```
Baseline      32/40 pass   80%
After prompt  36/40 pass   90%
Read the diff, not the total:
```

```
fixed 6, broke 2 that passed
4 left: all wrongly approve refunds
```

> **Gotcha:** A higher pass rate can hide a worse product. Break results down by case type, inspect regressions, and test on fresh cases the prompt was not tuned against.

Cost, latency, reliability

Lever	Question
Cost	Spend below the value created?
Latency	Fast enough for the workflow?
Reliability	Safe when a tool fails?
Scale	Holds at 50× the users?

What works for ten users can be too slow, costly or fragile for thousands. Route simple questions to a small model and hard ones to a capable model, cut redundant calls, and fail visibly: when search breaks, say the

source could not be retrieved and offer retry or manual lookup — never invent an answer.

```
50,000 questions/month
All capable  $0.10 x 50k = $5,000
Routed avg   $0.03 x 50k = $1,500
Track: cost per RESOLVED question
  = model + retries + human review
```

> **Gotcha:** Cost per model response can fall while cost per resolved question rises — if the cheap model causes more retries, re-edits or escalations, the savings reappear as staff time.

Make the business case

Audience	Wants to hear
Engineers	Which capabilities, models, trade-offs
Customers	Faster, easier, more reliable work
Leadership	Efficiency or revenue, and the evidence

A model that is accurate, cheap and reliable is still only worth building if its value covers build, run and upkeep. Put time saved, operating cost, measured quality and other costs side by side, then focus where evidence shows the most value — often routine cases, with complex ones kept human. Plan in four-to-eight-week horizons, since cheaper and better models keep changing the answer.

```
Saved  50k x 4 min = 3,333 agent-h/mo
Model  $1,500/mo inference
Other  eng, infra, monitoring, upkeep
Eval   routine: good; disputes: human
Bet    automate routine, keep disputes
```

> **Note:** Hours saved is capacity, not cash. It becomes savings only if the team's work or headcount changes — and adoption, not usage counts, shows whether the workflow actually changed.

Prove it with a portfolio

Project	Shows you can
Cited RAG copilot	Discover, retrieve, eval
Bounded agent	Tools, permissions, handoffs
Measured improvement	Trade cost vs quality with data

Build the smallest prototype that tests the question — an image mock to compare layouts, a working flow to test whether users can verify answers. Ten demos prove you can use tools; one project with evals, failures and decisions proves product judgment. For the agent, break a tool on purpose and show it asks for help instead of continuing on partial data.

```
For every project, make visible:
  Problem   who, what pain, why this
  Product   demo or working prototype
  Design    architecture, model choice
```

```
Evidence  evals, metrics, failures
Decisions trade-offs, what changed
```

> **Warning:** Never invent adoption or impact. If no real user has tried it, say so — interviewers probe numbers, and one made-up metric discredits the rest.

Further reading

- Anthropic — Building effective agents — workflows versus agents, and when to add autonomy.
- Anthropic — Demystifying evals for AI agents — graders, task design and reading transcripts.
- Google PAIR — People + AI Guidebook — trust, explanation and error handling in AI UX.
- Microsoft — Guidelines for Human-AI Interaction — 18 tested guidelines for AI behaviour.
- Cursor — Building Bugbot — a pipeline-to-agent migration measured by resolution rate.