PART 1 OF 3 · THE MECHANICS

What Exactly Is an AI Agent?

Clausey

The mechanics nobody on your feed actually explains. No autonomy hand-waving — just the moving parts, and the single API call they're all built on. By the end, you can whiteboard one.

Statelessness · The message array · The loop · Tools · The economics

Clausey

An agent is... a model, in a loop, with tools.

Four words doing all the work — and every turn of that loop is a single API call to the model. Unpacking that one call is the rest of this deck.

1

The model

A large language model — the reasoning. Turns text into the next step.

2

The tools

How it reaches past its own text to act on the world.

3

The loop

Your code calling the model again and again — the engine.

4

The goal

What you put in the prompt; how it knows it's done.

Most explainers stop at this sentence. We're going to keep going.

SECTION 01 / 04

The Model

01

The brain that does the reasoning — and the one surprising thing it can't do.

1 · The Model | 2 · The Tools | 3 · The Loop | 4 · The Goal

Clausey

The “model” is a large language model.

An LLM is a text predictor trained on a huge amount of writing: give it text, it returns the most plausible continuation using probabilities about word distributions. That's the whole brain — it turns its input into text. Everything else an agent does is wiring your code adds around that one ability.

Context window = how much text it can read at once. The hard ceiling everything else in this deck works around.

Closed — call their API

you hit the provider's servers

ANTHROPIC · OpenAI · Gemini

Open — run it yourself

you host it, hit your own API

LLaMA by Meta · MISTRAL AI_ · Qwen · deepseek

Either way, your agent never touches the model directly — it just calls an API.

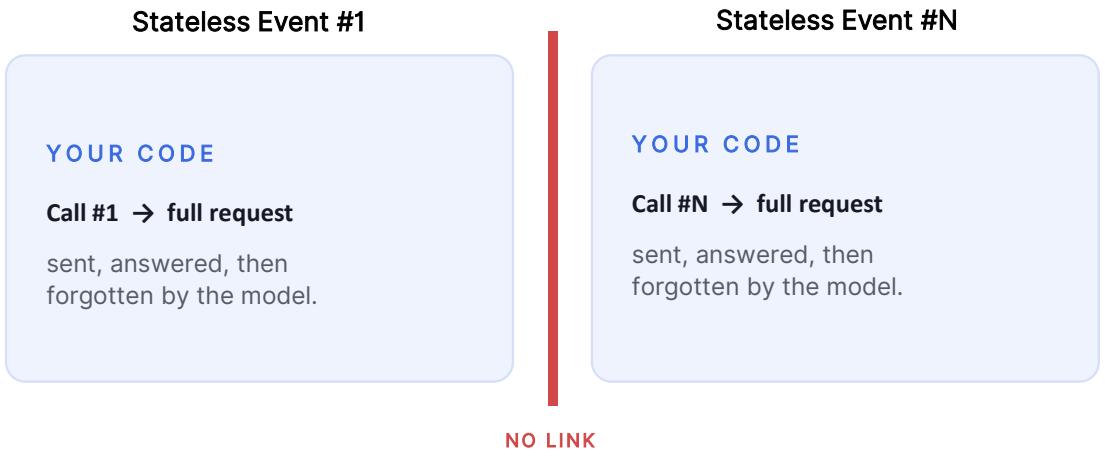
New to LLMs? Watch how they actually work →

THE FOUNDATION

Clausey

Every call is “stateless.”

That's the one piece of jargon worth knowing. Stateless means the model keeps no memory between calls: you send a message, it answers, and it instantly forgets the whole exchange. Ask a tenth question and it has no idea the first nine happened — unless you send them again.



THE ANALOGY

A brilliant doctor with total amnesia. Every patient meeting starts cold — so you bring the entire case file, every time.

Accept that the provider keeps nothing, and the rest becomes a logical consequence.

What you actually send...

A system prompt, then a messages array of alternating turns — each made of typed content blocks. That payload is the model's entire universe.

- **system**

  standing instructions — who the model is, what it can do

- **messages[ ]**

  ordered, alternating user / assistant turns

- **content blocks**

  text · image · tool_use · tool_result

```
{
  system: "You are a data analyst...",
  messages: [
    { role: "user", content: [
        { type: "text", text: "..." } ] },
    { role: "assistant", content: [
        { type: "tool_use",
          name: "query_db",
          input: {...} } ] },
    { role: "user", content: [
        { type: "tool_result",
          content: "..." } ] }
  ],
  tools: [ ... ]
}
```

Above is Anthropic's shape. Shapes vary by provider — the pieces don't.

There is nothing else. Everything the model "knows," someone put here. Let's take the three pieces one at a time.

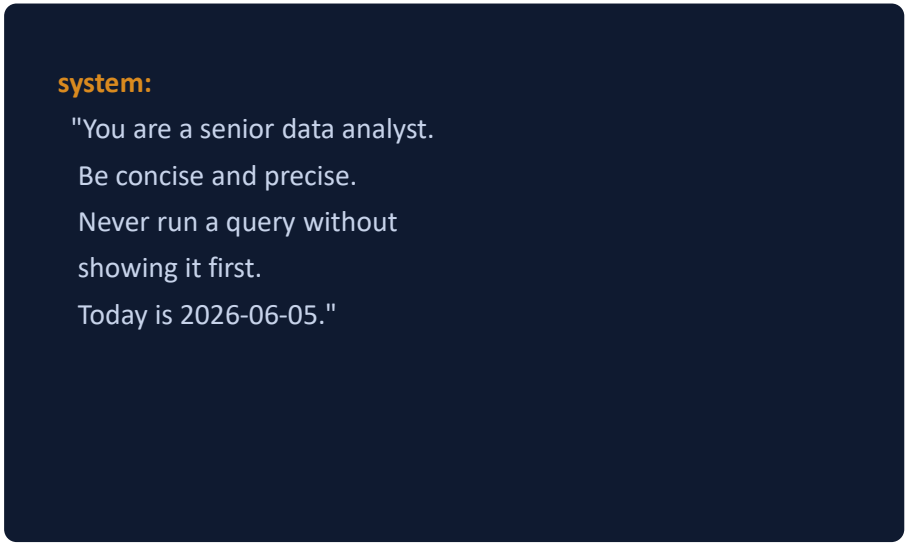
WHAT YOU SEND · SYSTEM

Clausey

system: your standing instructions.

Set once, at the very top of every call. The system prompt frames who the model is, how it should behave, and what it's allowed to do — its persona, its rules, its background. It does not change from turn to turn.

Like a job description. Written once, read before every task, the same each time — not part of the back-and-forth.



SAME EVERY CALL

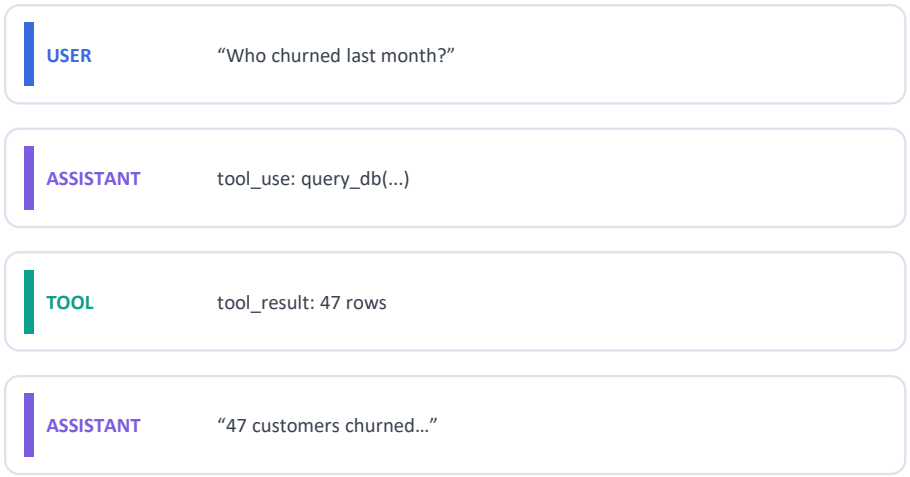
It's just text you write — and because it never changes, it's the first thing caching reuses. (More soon.)

Clausey

messages: the conversation, in order.

An ordered list of turns. Each turn carries a role — user or assistant — and the order is the timeline of what happened. This array is the conversation; there is no other copy of it anywhere.

Like a script. The role is the character name; the order is the scene. Read top to bottom, it's the whole story so far.



Re-sending this whole array every call is the only reason the model "remembers" the conversation.

content blocks: a message isn't just text.

Inside each message, the content is a list of typed blocks. Every block declares a type — and one message can hold several at once. This is how a single conversation carries words, images, tool requests, and tool outputs in one structure.

type:

"text"

Plain words — a question, an answer, a thought.

type:

"image"

A picture the model can look at and reason about.

type:

"tool_use"

The model asking to run a tool, with arguments.

type:

"tool_result"

What your code sent back after running it.

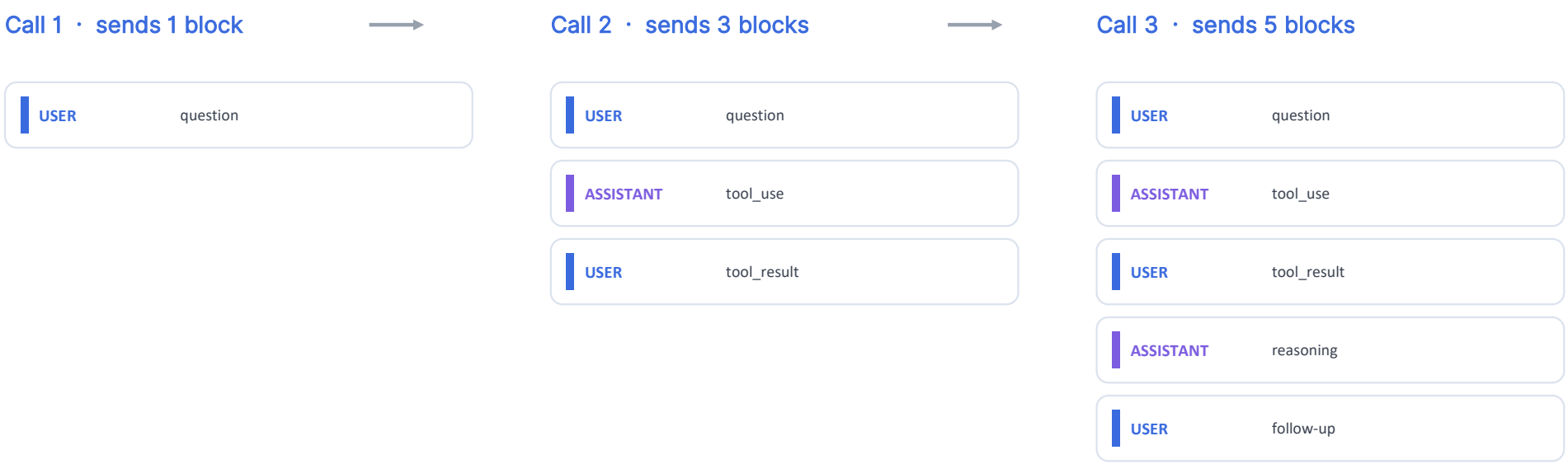
Same container, four block types — that's how tools and images ride inside the conversation.

HOW MEMORY WORKS

Clausey

Memory lives in your code.

The model “remembers” only because your client re-sends the entire array on every call. Delete a message before sending and, to the model, it never happened.



Who decides what goes in that file? That's what turns a chatbot into an agent.

SECTION 02 / 04

The Tools

02

How a text-only model reaches out and changes the world.

1 · The Model | 2 · The Tools | 3 · The Loop | 4 · The Goal

HOW TOOLS WORK

Clausey

The model asks. Your code does the work.

A model can only produce text — it can't touch a database or call an API. A “tool call” is just four steps your code runs around it:

1

You list the tools

In the request, you describe what's available and what each one does.

2

The model asks

It emits a tool call — the tool name plus JSON arguments — then stops and waits.

THEN STOPS

3

Your code runs it

Real code you wrote executes the actual operation:

4

Result goes back

You append the result to the array and call the model again with it in hand.

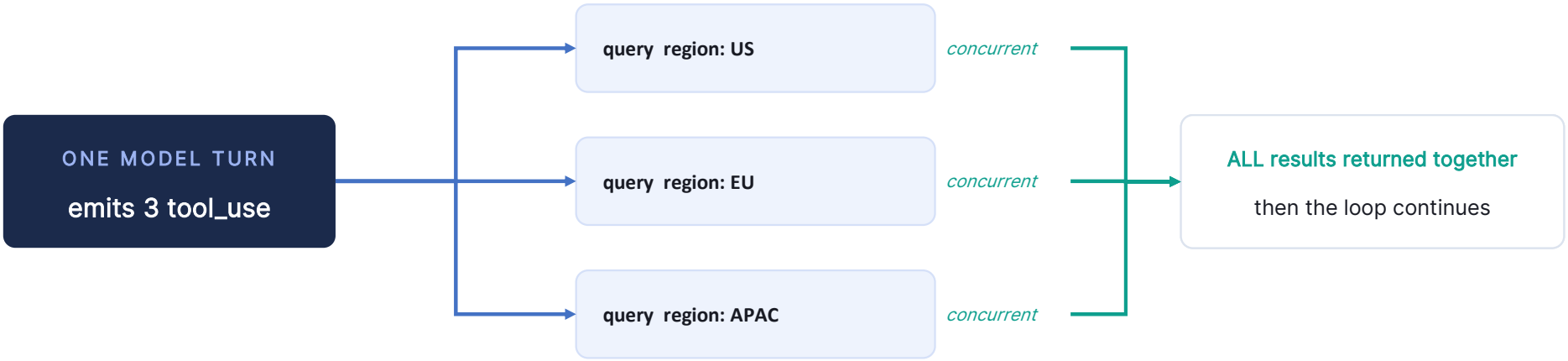
The model never has its hands on production — it can only ask, and your code decides what's wired up.

INTRA-TURN PARALLELISM

Clausey

One turn, many tool calls.

A single model turn can emit multiple tool requests at once. Your runtime executes them concurrently — but must return all results together, in the next turn, before the model continues.



Parallelism inside one loop. Running many agents at once is a different, larger problem — Part 2.

SECTION 03 / 04

The Loop

03

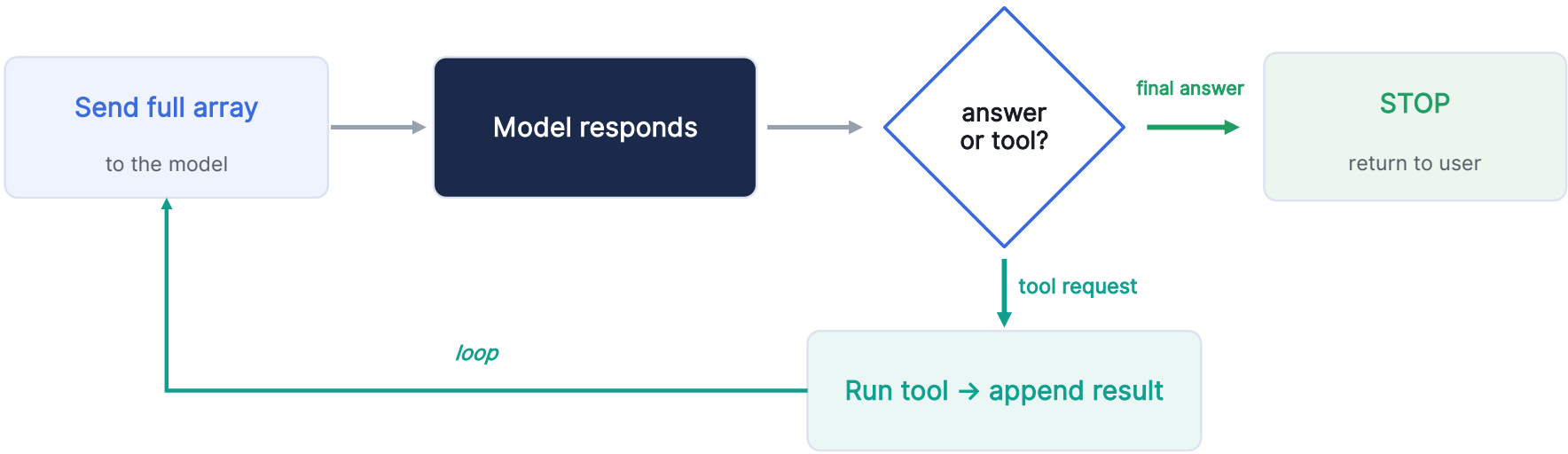
The rhythm that turns one answer into ongoing work — and what it costs.

THE LOOP

Clausey

The agent is the loop.

Send the array → the model returns a final answer (stop) or a tool request (continue) → run the tool, append the result, send again.



Guardrail: *a max-iterations limit force-stops the loop so a confused model can't churn forever.*

Note: *when a tool errors, the result still goes back and the model retries — it doesn't crash the loop.*

TRACED, STEP BY STEP

Clausey

One question, three iterations.

“How many enterprise customers churned this quarter?”

Watch the array grow. “Thinking” is just accumulated context — nothing happens you can't point to.

Iteration 1

USER	the question
ASSISTANT	tool_use: query_db

Can't answer cold → requests data, halts.

Iteration 2

USER	the question
ASSISTANT	tool_use: query_db
TOOL	result: raw rows
ASSISTANT	tool_use: filter tier+date

Reads data, needs to refine → second request.

Iteration 3

USER	the question
ASSISTANT	tool_use #1
TOOL	result #1
ASSISTANT	tool_use #2
TOOL	result #2
ASSISTANT	“47 churned.”

Array now holds everything → final answer.

YOU CONTROL MEMORY

Clausey

You decide what it remembers.

Your code rebuilds the array before every call — so you choose exactly what the model sees. Three common ways to handle a history that keeps growing:

Keep everything

Send the full history every call.

- user
- assistant
- tool
- ...

Total recall — but slower and pricier as it grows.

Drop old turns

Keep only the last few messages.

- *older*
- *older*
- recent
- recent

Fast and cheap, but forgets early context. (A form of compaction.)

Summarize

Replace old turns with a short digest.

- digest of older
- recent
- recent
- recent

Keeps the gist, loses detail. (Also compaction.)

Trimming the array is compaction; making what you re-send cheaper is caching — the two cost mechanisms, just ahead.

THE PRICE OF STATELESSNESS

Clausey

Why long agents get slower.

Every call reprocesses the full history. Per call, cost grows linearly with length. But across a run of n turns, the early tokens are processed n times — so cumulative work grows roughly quadratically.

linear

cost of a single call

~quadratic

cumulative cost of the run

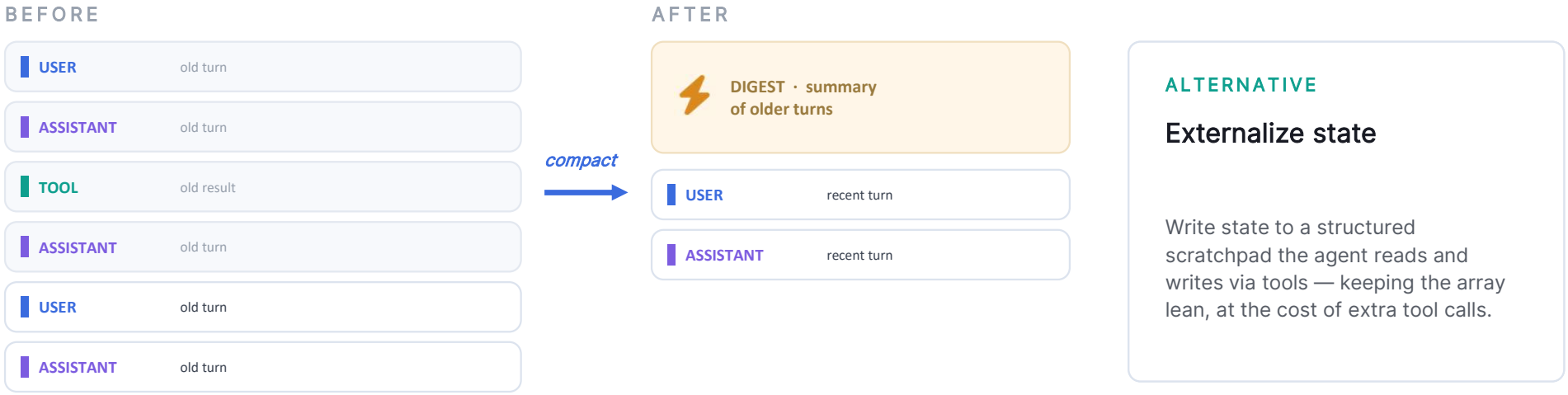
Not a bug — the direct price of statelessness. The next two mechanisms exist to manage it.

MECHANISM 1 · COMPACTION

Clausey

Compaction buys headroom.

This is “drop old turns” and “summarize” from the memory slide, done deliberately: near the context limit, fold older turns into a compact digest and keep recent turns verbatim. Honest tradeoff — summaries are lossy.



ALTERNATIVE

Externalize state

Write state to a structured scratchpad the agent reads and writes via tools — keeping the array lean, at the cost of extra tool calls.

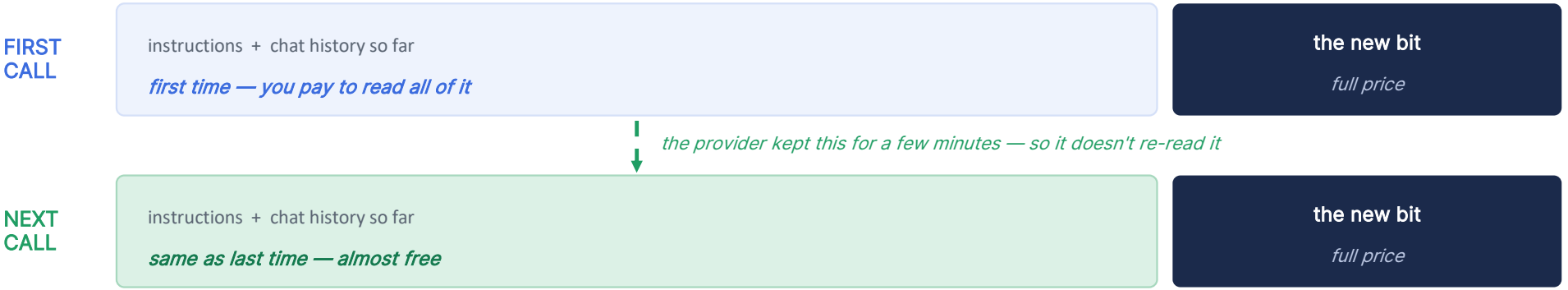
Compaction trades fidelity for headroom. A real tradeoff — not a free win.

MECHANISM 2 · CACHING

Clausey

Caching: pay once for the part that repeats.

Every call re-sends the same long beginning — your instructions, tool list, and the conversation so far — then a little new text. Re-reading that beginning costs money every single time. So the provider offers a deal: it remembers the processed beginning for a few minutes, and barely charges you to re-read it. *Like a barista who knows your usual — you only say what's new today.*



It's a discount, not memory. You still ship the whole array every call. Caching only makes those tokens cheaper — it does not extend the context window and does not remember anything for you.

SECTION 04 / 04

The Goal

What the agent is working toward — and how it knows it's finished.

04

1 · The Model | 2 · The Tools | 3 · The Loop | **4 · The Goal**

Clausey

A goal is what makes it an agent.

The goal is just what you ask for, in the prompt. The loop then runs on its own until the model decides the goal is met — it stops asking for tools and returns a final answer. That self-directed stopping is the line between an agent and a plain chatbot.

Done when…

the model returns a final answer with no tool call attached. The loop sees that and stops on its own.

The guardrail

a max-iterations cap. If the model never declares victory, the cap stops it from looping — and billing — forever.

No goal, no agent — just a chatbot. The goal is the reason the loop exists.

THE WHOLE MACHINE

The whole machine, one diagram.

Clausey

Model

reasoning — reads the array, decides the next move

Tools

actuation — the model's reach into the real world

Loop

control flow — send, decide, execute, append, repeat

Message array

memory — the one growing object your code builds

Compaction + caching

economics — what makes long runs survivable

That's an agent, completely. Five parts, no magic. If you can draw this from memory, you understand agents better than most of the feed.

THE LIMITS

You've got the whole machine. It has two limits.

Clausey

An array your code builds, a loop that calls a model, tools it can request, and two cost mechanisms — that's a complete, working agent you could ship. But one loop running in one process runs into two hard walls:

LIMIT 1

It can't outgrow one window.

One agent has one finite context window. A job too big to fit — or one that needs many things happening at once — can't run in a single loop.

→ PART 2 · AGENTS WORKING TOGETHER

LIMIT 2

It can’t outlive its process.

The array — the agent's entire mind — lives in one running process’s memory. A crash, timeout, or routine deploy erases it. There's no resume.

→ PART 3 · DURABLE EXECUTION

THE SERIES

This was Part 1 of 3.

Clausey

If you can now explain why calls are stateless, why memory is a policy, and why the loop is the whole engine — you're ahead of most of the conversation.

01

What Exactly Is an AI Agent?

The anatomy of a single agent — down to the API call.

02

How AI Agents Work Together

Orchestration: sub-agents as context compression, coordination, aggregation.

03

Why Most Agents Die in Production

A structural failure teams overlook: durable execution — keeping agent state alive when the process dies.

Follow for Parts 2 and 3. The machinery only gets more interesting from here.