



Hochschule Aalen  
UNIVERSITY OF APPLIED SCIENCES

# Lecture 7

## THE SYSTEM AROUND THE MODEL

# Four promises from Lecture 6

| Promise from Lecture 6                                                             | Paid today                                    |
|------------------------------------------------------------------------------------|-----------------------------------------------|
| Resolve <b>mystery 8</b> — same instructions, different tools, different behaviour | Chapters 3–4 build it · chapter 12 retires it |
| Reframe <b>skills</b> as codified prompt patterns                                  | Chapter 7                                     |
| Introduce <b>MCP</b> — the open tool/data standard                                 | Chapter 9                                     |
| Set up <b>evaluation</b> — Act 3 begins                                            | Chapter 13                                    |

## THE CONTRACT

Every debt visible from the first page — and every one paid by the end of the lecture.

# Two blocks, one zoom-out

## Block A — The Orchestration Layer

Chapters 2–10 build the system piece by piece: the zoom-out, the token sequence, CLAUDE.md, memory, skills, the tool loop, MCP, the budget.

## Block B — Lab, then ceremony

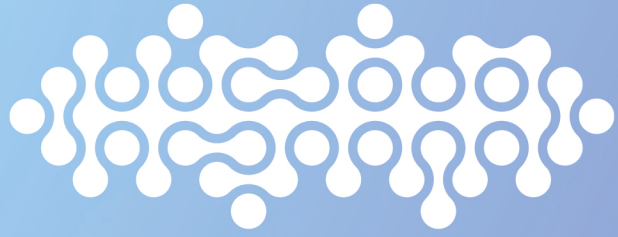
Three exercises on your own Locust files — then the Mystery Board comes down and Act 2 closes.

### TODAY'S DIAGRAM TICK

The **LLM-in-One-Picture** reaches its final Act 2 form: **v1.3** — one version tick per lecture since Lecture 4.

The model is **powerful but passive**; the orchestration layer is **simple but active** — and knowing which job belongs to which is what separates someone who **uses** AI tools from someone who **engineers** with them.

the sentence to carry from here to the exam — chapter 2 earns it

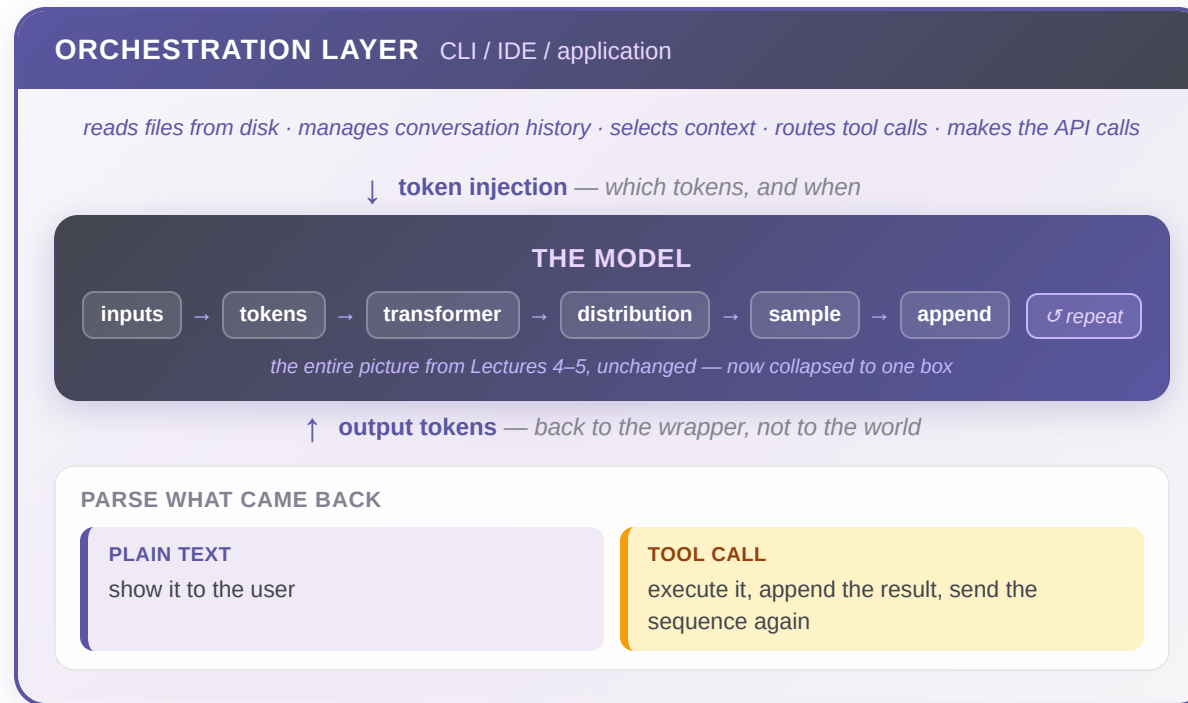


Hochschule Aalen  
UNIVERSITY OF APPLIED SCIENCES

# The Orchestration Layer

1

# LLM-in-One-Picture v1.3



# Two arrows, two jobs

## ARROW IN — TOKEN INJECTION

The orchestration layer assembles a token sequence — from files it loaded, the conversation so far, and tool results — and ships it in.

## ARROW OUT — OUTPUT TOKENS

The model's tokens go back to the **wrapper**, not to the world. It parses them: plain text → display; tool call → execute, append, send again.

## THE WHOLE LECTURE

Everything the model does to the outside world, it does by emitting tokens the wrapper chooses to act on. **It has no other interface. It never did.**

# Powerful but passive, simple but active

|                                 | The model                                                     | The orchestration layer                                          |
|---------------------------------|---------------------------------------------------------------|------------------------------------------------------------------|
| Initiates anything?             | Never — runs only when called                                 | <b>Always</b> — loads files, assembles context, makes every call |
| Reads the filesystem?           | No — does not know what files exist                           | <b>Yes</b> — by rule (file discovery, tool execution)            |
| Executes commands?              | No — only <i>describes</i> a command in tokens                | <b>Yes</b> — decides whether and how to act                      |
| Persists between conversations? | No — nothing carries over inside                              | <b>Yes</b> — files on disk                                       |
| How it fails                    | Probabilistically — distribution shape, attention competition | Deterministically — a rule fired or it didn't                    |

CALL IT OUT

## Inner box or outer box?

**A surprising word choice**

INNER — DISTRIBUTION

**A file got loaded... or didn't**

OUTER — RULES

**It "remembered" a preference**

OUTER — FILESYSTEM LOOP

Decide before you read the tag. Three observations, three diagnoses.

# Two systems, one illusion

## THE LECTURE 3 "MAGIC", DISSOLVED

Two systems were always involved. The **orchestration layer** noticed CLAUDE.md and loaded it — a rule, a dozen lines of code. The **model** then processed those tokens through attention — strong, but never absolute.

## THE MAGIC WAS A DIVISION OF LABOUR

One system put the tokens in the room; the other listened to them, probabilistically.

# Why the wrapper stays invisible

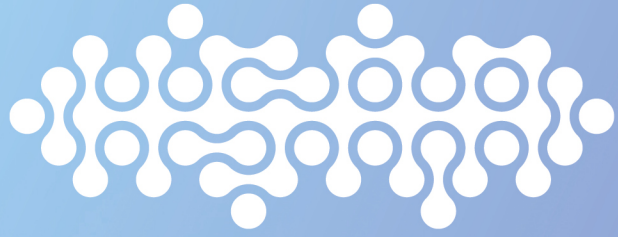
## INTERFACE DESIGN

A chat window presents a single persona — one name, one voice. Every capability gets attributed to that persona: "the AI has memory", "the AI ran the tests".

The persona is a deliberate, reasonable design choice. But it is also why nearly every user's mental model is wrong in the same direction: **everything is credited to the model, and the wrapper — where most engineering decisions live — becomes invisible.**

## V1.3'S JOB

You cannot engineer what you cannot see. Make the wrapper visible, and keep it that way.



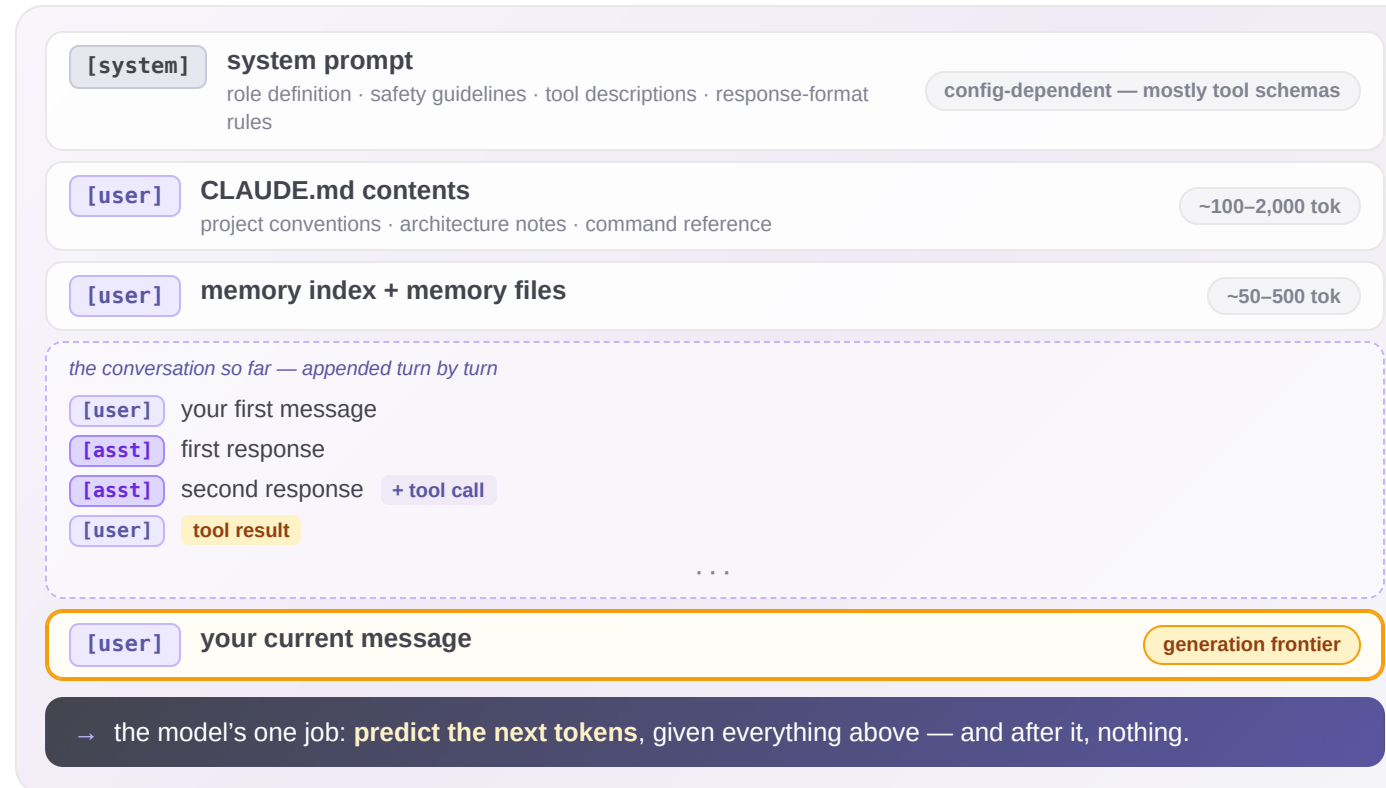
Hochschule Aalen  
UNIVERSITY OF APPLIED SCIENCES

# What the Model Actually Sees

2

# The token sequence, end to end

ONE API CALL — WHAT THE MODEL RECEIVES, TOP TO BOTTOM



# The role rhythm

Strip away the contents and one API call is just a sequence of role-labelled turns — and the role labels are **tokens in the text**, not metadata around it:

[system] [user] [user] [user] [asst] [user] [asst] . . . [user]

## ONE DETAIL

[system] is not a flag on a database row. It is rendered into the sequence as special turn-boundary tokens — the same kind post-training was formatted with. The labels are part of the text the model reads.

# Observation 1: no instruction channel

## OBSERVATION 1

From the model's perspective, there is **no difference** between CLAUDE.md tokens and your question tokens.

At the embedding layer there is no instruction channel (Lecture 5), and there is no instruction channel anywhere else either. The only structural marking is the role label — and a label is influential not because the architecture enforces it, but because **post-training taught the model to weight [system] turns heavily**.

## THE INTERFACE LIES (HELPFULLY)

Your chat window renders the blocks differently — instructions hidden, messages in colour, tool output folded away. The model sees none of it. It receives a flat sequence of tokens.

# Observation 2: the sequence only grows

## OBSERVATION 2

**The sequence only grows.** Every reply, every tool result, every correction, every false start — appended. Never removed, never overwritten.

Your mistakes and the model's mistakes stay in the sequence alongside everything that came after them. Chapter 6 turns this property into the most counter-intuitive practical scenario of the day.

# Observation 3: re-sent on every call

## OBSERVATION 3

**The entire sequence is re-sent on every call.** The model is stateless — a forward pass is a function from a token sequence to a distribution. The continuity you experience is carried entirely by the sequence itself.

## KV CACHE ≠ "THE MODEL REMEMBERS"

The KV cache saves recomputation *within* one generation. It changes what must be recomputed — not what must be sent. The full sequence ships every time.

# Observation 4: one response = many calls

## OBSERVATION 4

What you experience as one assistant response is frequently many API calls.

Each time the model wants to read a file or run a command, generation stops, the wrapper acts, the result is appended, and a fresh call ships the grown sequence back. **Five or ten round trips behind a single apparent "response" is unremarkable.**

## A CONSEQUENCE YOU CAN FEEL

Responses slow down as a session ages — every generated token attends over the whole accumulated sequence. The mechanics of that loop are chapter 8.

# The message array

What the orchestration layer keeps between calls is almost embarrassingly plain — a list with an append method:

```
messages = [  
    { role: system, content: <system prompt> },  
    { role: user,   content: <CLAUDE.md contents> },  
    { role: user,   content: "Fix the failing test in parser.py" },  
    { role: asst,   content: "Let me look." + tool_call(Read, parser.py) },  
    { role: user,   content: <tool result: parser.py contents> },  
    { role: asst,   content: "Done. The off-by-one on line 41 is fixed." },  
]
```

## ONE WRITE, ONE READ

The array supports exactly one write — **append**. Its one read is the API call: serialise the whole array and ship it — not the delta — on every round trip.

Every chapter from here is a rule about **what gets appended to this array, and when.**

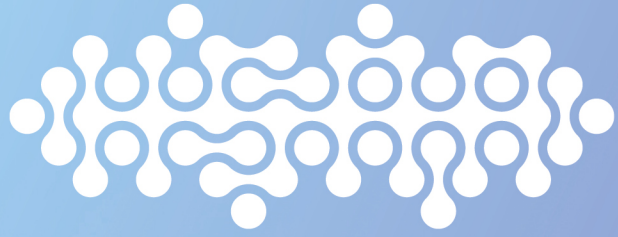
CLAUDE.md loading · memory · skills · tool results · the context budget

# Checkpoint: the token sequence

## HOLD THESE FOUR

1. One sequence, top to bottom — no side channel.
2. It only grows.
3. Re-sent in full on every call.
4. One "response" = many API calls.

Every chapter from here applies them. Next: how CLAUDE.md becomes part of this sequence.



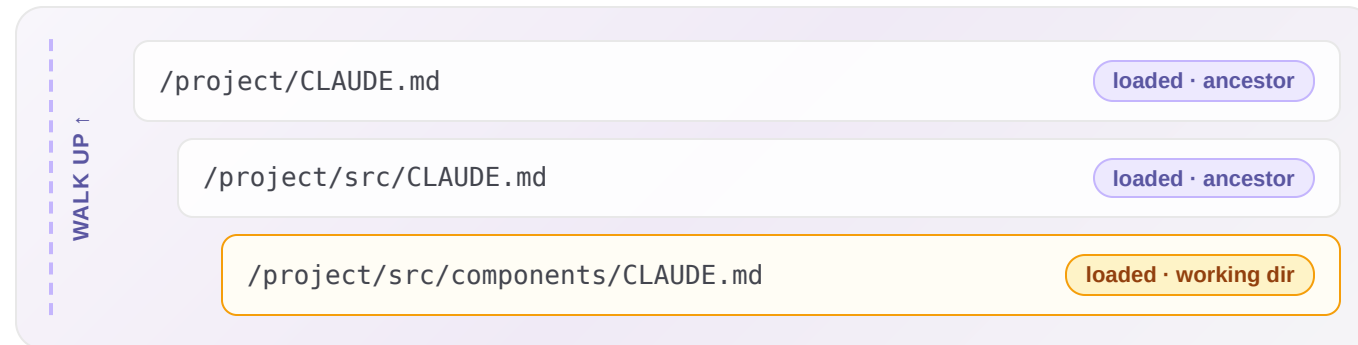
Hochschule Aalen  
UNIVERSITY OF APPLIED SCIENCES

# How Context Enters the Sequence

3

# Rule 1: the ancestor walk

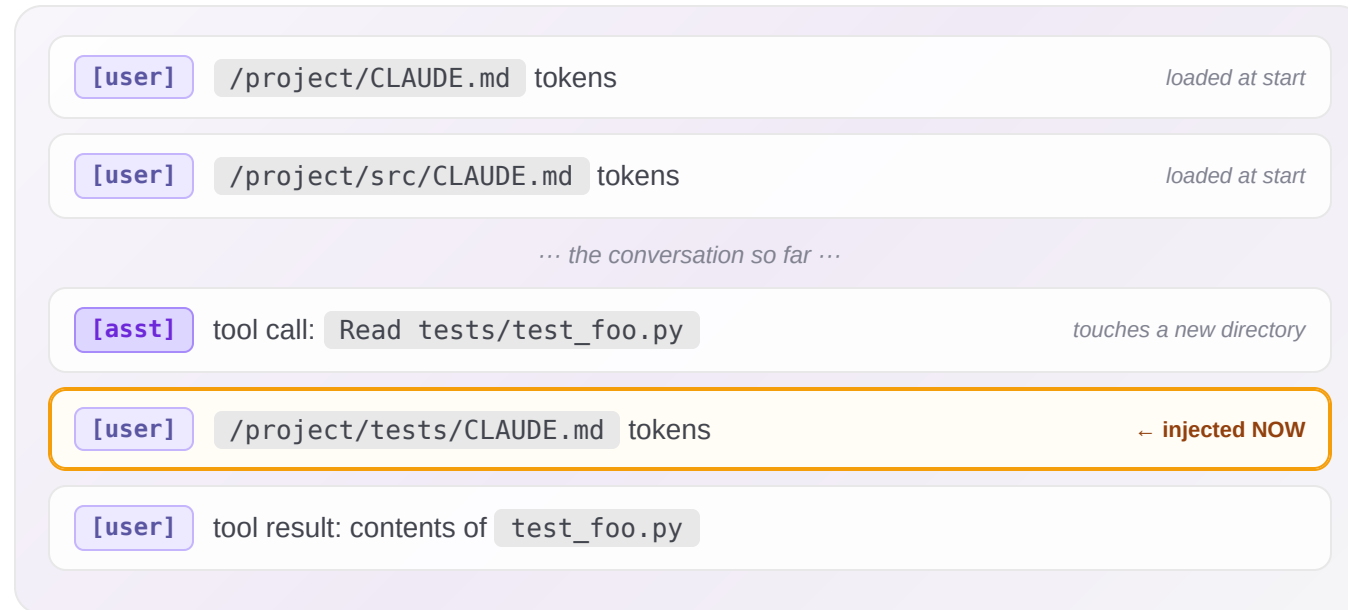
Session started in `/project/src/components/` — the CLI walks **up** the tree to the root, collecting every CLAUDE.md it passes



The walk runs **before your first message** — a dozen lines of ordinary code, no model involved — and hands over one concatenated `[user]` block.

# Rule 2: on-demand injection

RULE 2 — A DIRECTORY'S CLAUDE.MD IS INJECTED THE MOMENT A TOOL CALL FIRST TOUCHES IT



**Once per file, mid-sequence.** A directory's conventions are paid only by the conversations that actually open it.

# The neural network does not decide

## THE LOADING CLAIM

**The neural network does not decide which instruction files to load** — it cannot ask for more context. The orchestration layer decides, by rules as dumb as directory traversal.

## DUMB IS A FEATURE

Because the rules are mechanical, you can know — exactly, without running any model — which instruction files entered a given session. A neural decision would be one you could neither predict nor audit.

# Never loaded vs loaded but outcompeted

| Failure mode           | What happened                                                     | Which half                          | The fix                                                     |
|------------------------|-------------------------------------------------------------------|-------------------------------------|-------------------------------------------------------------|
| Never loaded           | Neither rule fired — the file sits where the walk never went      | <b>Orchestration</b> (this chapter) | Move the file up the tree, or start where the walk finds it |
| Loaded but outcompeted | Tokens are in the sequence, but attention competition eroded them | <b>Model</b> (L5 §16.1 + L6)        | Shorten, front-load, back with deterministic enforcement    |

## BOTH WEAR THE SAME FACE

"The model ignored my CLAUDE.md" — but the two fixes have nothing in common, which is exactly why the diagnosis matters.

# Memory: the read-write loop



## THE PYTEST ANECDOTE, DISSOLVED

**Memory is a read-write loop mediated by the filesystem — the persistence lives on disk, not in the model. Each conversation starts with a model that has never met you.**

# CLAUDE.md vs MEMORY.md

| Aspect                | CLAUDE.md                         | MEMORY.md + memory files                            |
|-----------------------|-----------------------------------|-----------------------------------------------------|
| Who writes it         | A human — deliberately            | The model — during conversations                    |
| Where it lives        | In the repo, version-controlled   | Outside the repo — per user, per machine            |
| Shared with the team? | Yes — everyone who clones gets it | No — it follows you, not the project                |
| Reliability           | Human-reviewed, diffable          | Unverified — possibly stale or wrong from the start |

## "WORKS ON MY MACHINE", AI-ERA VARIANT

Same repo, same CLAUDE.md, same model — different behaviour. Their memory directories differ, so the sequences differ, so the distributions differ.

# What can go wrong with memory

## Stale

The project moved on; the memory didn't. May's truth, asserted in June — in the privileged early positions.

## Born wrong

The model writes memory by sampling, exactly as it hallucinates in chat. "Team adopted httpx" — when it was only discussed.

## Unverified

Nothing checks a memory against the codebase. Attention can't tell "team convention" from "thing it concluded one Tuesday".

## THE POSTURE

Treat memories as **soft context** — useful for preferences and continuity — and verify any load-bearing technical fact against the repository.

# The mid-conversation edit

YOU EDIT CLAUDE.MD MID-SESSION AND RE-READ IT — HERE IS THE SEQUENCE AFTERWARDS



**No overwrite, only accumulation.** Your edit changed the file on disk — but the model sees the sequence, and the sequence can only grow. v1 and v2 now coexist.

# State machines forget; sequences remember

```
# a state machine  
x = 5  
x = 10  
print(x) # 10 – the 5 is gone
```

```
# a token sequence  
CLAUDE v1 @ pos 500  
CLAUDE v2 @ pos 10500  
generate(...) # both vote
```

## THE DEEPEST COUNTER-INTUITION

State machines forget; **token sequences remember everything, including what you wanted forgotten.**

# Start a new conversation

## THE ONLY TRUE OVERWRITE

Fresh sequence. The orchestration layer reads CLAUDE.md from disk as if for the first time; only v2 is loaded; **v1's tokens do not exist anywhere**, because the message array they lived in is gone.

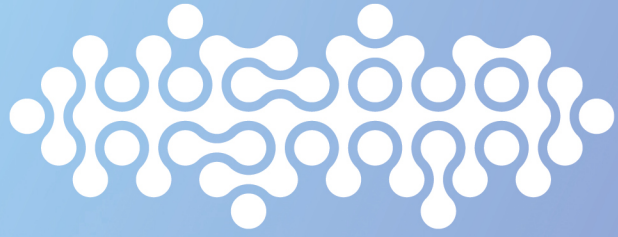
The folklore — "when in doubt, start a new chat" — finally gets its mechanism. Restarting isn't free: it forgets the bad and the good together. Judging when accumulated context has stopped earning its place is **chapter 10**.

# Forward pointer: retrieval (Act 3)

## THE SMALLEST EXTERNAL KNOWLEDGE STORE

A directory of text files, read at startup and injected as context, is the minimal implementation of external knowledge.

Act 3 meets the same idea again, scaled up, under the name **retrieval (RAG)** — where an algorithm replaces the filesystem walk in deciding what gets injected. The architecture is identical: knowledge outside the model, selected and spliced in by the orchestration layer.



Hochschule Aalen  
UNIVERSITY OF APPLIED SCIENCES

# From Context to Action

4

# Skills: codified prompt patterns

## LECTURE 6'S PROMISE, KEPT

A **skill** is a prompting pattern — a few-shot or chain-of-thought pattern — captured as a file on disk, injected into the token sequence **only when it is triggered**.

```
.claude/skills/error-handling/SKILL.md
---
name: error-handling
description: Use when writing or changing HTTP error handlers
---
Follow the project's error-handling pattern. Two references: ...
```

# CLAUDE.md vs Skill: a timing rule

| Aspect          | CLAUDE.md                   | Skill                            |
|-----------------|-----------------------------|----------------------------------|
| When it loads   | Every session, at start     | Only when triggered              |
| Token cost      | Paid in every conversation  | Paid only when used              |
| What it carries | Always-relevant conventions | One task's pattern               |
| Until needed    | —                           | Sits on disk, at zero token cost |

## THE WHOLE DIFFERENCE

Same transport as CLAUDE.md — text spliced into the sequence. The entire difference is a **timing rule** inside the orchestration layer.

# Always-on → on-demand → workflows



## THE ARC

The orchestration layer getting progressively smarter about **when** to inject **which** tokens. The layer stays simple; its rules get more discriminating.

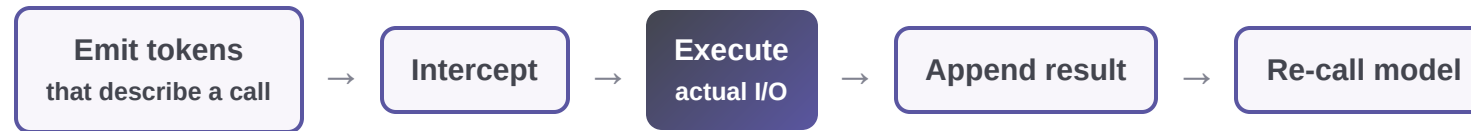
# The loop that acts

## THE CHANNEL THAT FLOWS BACK OUT

Everything so far moved text **into** the sequence. This is the channel that flows back **out** — tokens that act on the world.

How can a model that only emits tokens nonetheless read files and run tests? It cannot. The orchestration layer does — on the model's behalf, when the model's tokens ask it to.

# The mechanism, in five steps



## NO NEW MACHINERY

Step 1 is a next-token prediction. Step 3 is actual I/O — the model is idle. A sixth step hides between 2 and 3: **the human permission gate**.

# "Fix the bug in stats.py" — four API calls

"FIX THE BUG IN STATS.PY" — ONE RESPONSE, FOUR API CALLS

[user] "Fix the bug in stats.py"

[asst] "Let me read the file."

`tool_call(Read, stats.py)`

↓ *orchestration executes Read — one full API round trip ends here*

[user] **tool result** stats.py contents

~500 tok

[asst] "The comparison in `percentile()` looks inverted. Let me check the tests."

`tool_call(Read, test_stats.py)`

↓ *orchestration executes Read*

[user] **tool result** test\_stats.py contents

~300 tok

[asst] "Confirmed. Fixing it."

`tool_call(Edit, stats.py, ...)`

↓ *orchestration executes Edit*

[user] **tool result** "File updated successfully"

[asst] "Done. The inverted comparison in `percentile()` is fixed." — no tool call, so the loop exits and this reaches your screen.

# Act–observe–act



## CHAIN-OF-THOUGHT THAT ACQUIRED HANDS

Some intermediate tokens now trigger execution, and the world's replies return as context. A failed command is not an exception — **it's just more tokens**, and the next attempt is conditioned on the exact failure text.

# Division of labour in the loop

| In the loop                        | The model                           | The orchestration layer                |
|------------------------------------|-------------------------------------|----------------------------------------|
| <b>Contributes</b>                 | The decision, as tokens             | Every action on the world              |
| <b>Sees</b>                        | Only the token sequence             | The filesystem, the network, the shell |
| <b>Carries state between calls</b> | Nothing                             | The whole message array                |
| <b>Fails by</b>                    | Distribution problem (L6's toolkit) | Software problem (a debugger)          |

## THE POLARITY, SHARPEST

The layer that can do everything decides nothing; the model that decides everything can do nothing.

Every capability that touches the world belongs to the **orchestration layer**; the model contributes exactly one thing — **the decision, expressed as tokens.**

the model has no hands. it has no eyes. it has tokens.

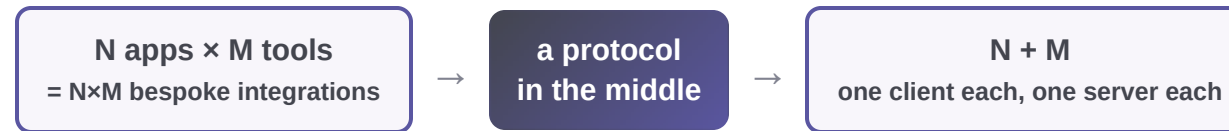
# Checkpoint: skills and tool calls

## FOUR TO KEEP

1. Skill = codified prompt pattern, loaded on-demand. 2. Tool loop = five steps, all append-only. 3. "Fix the bug" = four API calls. 4. The boundary: model decides (tokens), wrapper acts (I/O).

Next: the standard interface, and the one window everything shares.

# Why a standard: $N \times M \rightarrow N + M$

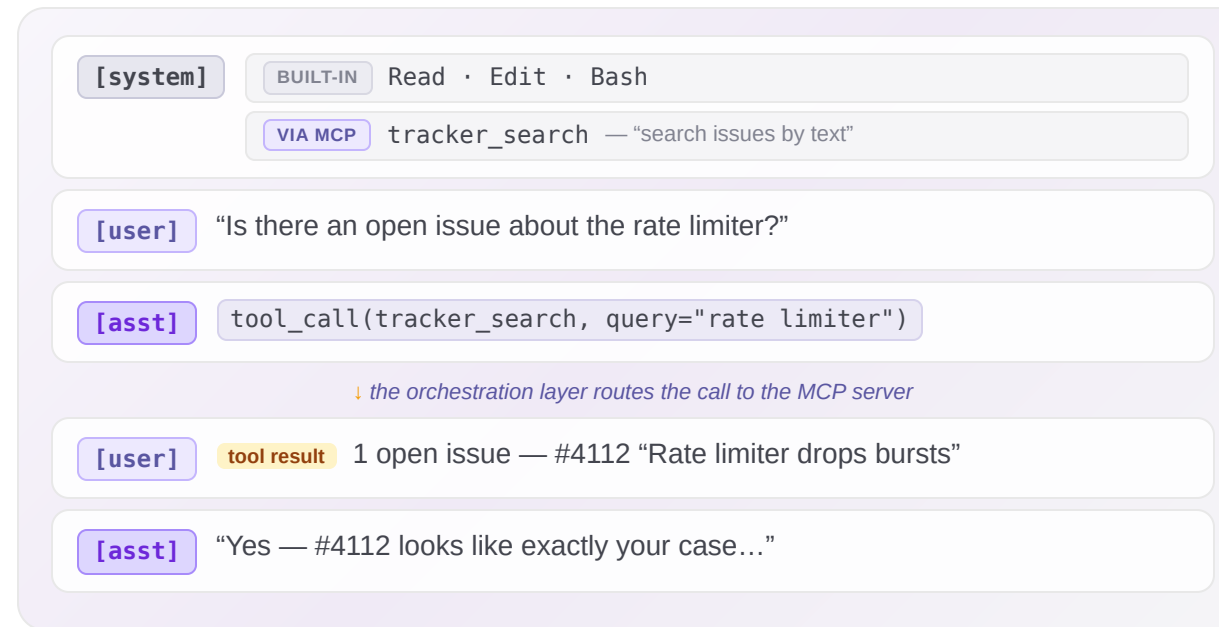


## NOT AN AI PROBLEM

An interface-explosion problem, solved by software engineering's standard technique. LSP did it for editors × languages; **MCP** does it for AI tools × data sources.

# MCP: indistinguishable from native

THE CHAPTER 8 LOOP, UNCHANGED — ONLY THE EXECUTOR BEHIND THE TOOL GOT SWAPPED



## TO THE MODEL

**An MCP tool is indistinguishable from a native one.** The standard lives entirely in the orchestration layer; the model needs no MCP support.

# Four enforceable decisions

| Decision wired at the interface | Mechanism                | What it gives you                      |
|---------------------------------|--------------------------|----------------------------------------|
| Which tools exist               | The catalogue you expose | The model cannot call what isn't there |
| What each may do                | Permissions / scopes     | A read-only tool can't write           |
| What shapes are legal           | Schema validation        | Malformed calls rejected before I/O    |
| What gets audited               | Logging at the boundary  | Every action is on the record          |

## L6'S ENFORCEMENT LAYER HAS A STREET ADDRESS

The permission gate you've seen all semester — "may I run this command?" — is the orchestration layer holding a call at this interface.

# Capability = tool surface, not intelligence

## THE SECURITY THESIS

**What an AI system can actually do is determined by its tool surface, not by its intelligence.** The model cannot exceed the tools the orchestration layer wires up.

Every tool you add widens what a hijacked token sequence can do. A model that emits a destructive call because an attacker buried an instruction in a web page is not malfunctioning — it is completing the most plausible continuation.

## THE DEFENCE

Never a sterner CLAUDE.md. **A narrower tool surface.** Enforcement, not influence.

# Everything competes for the same window

| Context source                                                 | Typical size (tokens)                                       | Who controls it               |
|----------------------------------------------------------------|-------------------------------------------------------------|-------------------------------|
| System prompt — role, guidelines, <b>built-in tool schemas</b> | config-dependent (often five figures); MCP schemas deferred | The vendor                    |
| Instruction files — the CLAUDE.md hierarchy                    | ~100–2,000                                                  | <b>You</b>                    |
| Memory files                                                   | ~50–500                                                     | The model                     |
| Conversation history                                           | grows without bound                                         | Everyone                      |
| Tool results — files, command output                           | ~100–10,000+ each                                           | The task                      |
| Current prompt                                                 | ~50–5,000                                                   | The user                      |
| <b>Response space</b>                                          | whatever remains                                            | <b>Nobody — the remainder</b> |

## ONE WINDOW, NOT SEPARATE POOLS

**Everything competes for the same finite window** — and being *in* the window is necessary; being *attended to* (per head, per query position) is the real currency.

HANDS UP

## Your Locust CLAUDE.md — how many tokens?

< 200 · 200–500 · 500–1,000 · > 1,000

That cost is paid in every conversation you start in that repo. Hold the number — the lab measures it for real.

# Why is there a maximum at all?

## NOTHING IN THE ARITHMETIC IMPOSES A CEILING

Attention multiplies a sequence of *any* length; the feed-forward network runs per token; the residual stream does not care if it carries a hundred tokens or a million.

## SO WHERE DOES THE SIZE COME FROM?

The maximum context window is nowhere in the transformer's arithmetic — it is a **declared** limit. Three constraints from Lecture 5 force the declaration.

# Three constraints from Lecture 5

| Constraint          | Why it bounds the window                                                                                                                    |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| Position            | A learned table runs out of rows; rotary/relative schemes drift past the trained length — the model no longer knows <i>where</i> a token is |
| Quadratic attention | Every token scores every other → an $N \times N$ grid per head, per layer. The published number is the length they paid to <i>train</i>     |
| KV cache            | Stores a key+value per token at serving time; grows with the conversation and must fit in accelerator memory                                |

## READ HONESTLY

The vendor's number means **"trained, affordable, and supported up to here."** A property of the deployment as much as the architecture.

# Window sizes: a short history

| Generation       | Year      | Context window                    |
|------------------|-----------|-----------------------------------|
| GPT-2            | 2019      | ~1K tokens                        |
| GPT-3            | 2020      | ~2K tokens                        |
| GPT-4 era        | 2023      | 8K/32K at launch; 128K with Turbo |
| Current frontier | 2024–2026 | 200K – 1M tokens                  |

THREE ORDERS OF MAGNITUDE IN SEVEN YEARS — AND THIS CHAPTER STILL EXISTS

**Window growth bought headroom, not automatic immunity.** More room helps only if the added context is relevant and well placed; lost-in-the-middle is an empirical, model/task-dependent U-shaped effect.

# When the window fills

## Sliding window

Drop the oldest messages. Cheap, predictable, and brutal — turn three is simply gone.

## Keep N recent turns

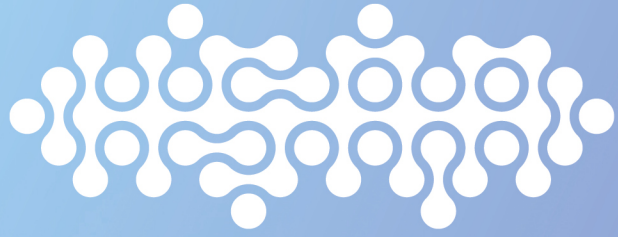
Preserve the system prompt + instructions at the head; the middle of the session is lost.

## Summarisation

Compress older turns into a digest — but the digest is generated text, with opinions.

### EVICTON IS A SYSTEM DECISION

When a long session "forgot" something, the orchestration layer **evicted** it — not a memory failure inside the weights. Act 3 names this discipline: *context engineering*.



Hochschule Aalen  
UNIVERSITY OF APPLIED SCIENCES

# The System, Completed

5

# The influence layer, revisited

INFLUENCE = ANYTHING THAT ENTERS THE TOKEN SEQUENCE

Instruction files, memories, skills, tool results, prompts — all the same kind of thing: text spliced in, competing for attention.

THREE RULES, NOW DERIVABLE

**Keep it concise** (the budget). **Front-load** (position + lost-in-the-middle). **Be specific** (distribution shape).

# The enforcement layer, revisited

## ENFORCEMENT = DETERMINISM BOLTED ON FROM OUTSIDE

Linters, tests, CI, schema validators. The output re-enters the model as tool-result tokens — a gate on the way out, a feedback signal on the way back in.

## THE DUAL NATURE

A failing test is **deterministic on the way out** and **influence on the way back in**.

Determinism cannot be **prompted** into the model; it has to be **bolted on from outside.**

a better influence layer narrows the distribution — only the enforcement layer truncates it

# The human layer, revisited

## HUMAN = AUTHOR OF EVERY WRAPPER DECISION

Walk back through chapters 2–10: every load-bearing decision was a human decision expressed as configuration — CLAUDE.md content, memory curation, skill codification, tool permissions, budget policy.

The model is powerful but passive, the wrapper simple but active — **and the activity is scripted**. The orchestration layer has no judgement of its own; every rule it follows is a rule a human wrote down.

# The three-layer model, completed

| Layer              | Lecture 6 description | Act 2 mechanism                                                   | Where in v1.3             |
|--------------------|-----------------------|-------------------------------------------------------------------|---------------------------|
| <b>Influence</b>   | Probabilistic         | Files & patterns become tokens; attention shapes the distribution | Inside the token sequence |
| <b>Enforcement</b> | Deterministic         | Exit codes & validators; results re-enter as tokens               | In the wrapper's world    |
| <b>Human</b>       | Semantic              | Decides what "right" means; authors every config decision         | Outside both              |

## CARRY THIS INTO ACT 3

Every property is now derivable, not asserted. Influence = chs 3–7. Enforcement = ch8's feedback loop. Human = every chapter's configuration decision.

# Likely, catches, judges

ASSERTED IN L6, DERIVED TODAY

The **influence layer** makes the model *likely* to do the right thing; the **enforcement layer** *catches* it when it does not; the **human layer** *judges* what "right" means.

Likelihood is attention arithmetic and distribution shape. Catching is determinism injected from outside. Judging is the one part that was never next-token prediction to begin with.

# Checkpoint: from mechanism to framework

## THREE LAYERS, FULLY DERIVED

**1. Influence** = tokens in the sequence (chs 3–7). **2. Enforcement** = determinism from outside the model (ch8 loop). **3. Human** = author of every wrapper decision.

Every mystery on the board is now answerable somewhere inside this picture. Which means the board has served its purpose.

THINK — PAIR — SHARE

## Same instructions, two tools, different behaviour. Why?

You've had the mechanism for two chapters now. Name the differences before the reveal.

# Mystery 8, fully resolved

WHY DO DIFFERENT TOOLS BEHAVE DIFFERENTLY FROM THE SAME INSTRUCTIONS?

Four concrete differences — each a mechanically different token sequence:

## Different files

CLAUDE.md vs AGENTS.md vs  
copilot-instructions

## Different placement

[system] vs [user] — different  
learned weight

## Different loading

start vs mid-conversation —  
different position

## Different context

different memories, skills, tool  
results

THERE WAS NEVER ANYTHING ELSE TO EXPLAIN

**Same weights, different wrapper, different token sequence, different behaviour.**

# The Mystery Board comes down (1/2)

| # | Mystery                  | Resolved | Mechanism, in one line                                                |
|---|--------------------------|----------|-----------------------------------------------------------------------|
| 1 | Can't count letters      | L4       | Tokenisation hides characters inside multi-character tokens           |
| 2 | CLAUDE.md inconsistent   | L5+L6+L7 | Injected as tokens (L7); attention competition, drift (L5+L6)         |
| 3 | English prompts better   | L4+L5    | Token efficiency + attention-competition effects                      |
| 4 | Curated > paste-all      | L5       | Distractor tokens compete with what matters                           |
| 5 | Hallucinated APIs        | L6       | Pattern completion, no ground-truth check (flat OR confidently wrong) |
| 6 | Vague prompts → nonsense | L6       | Under-specified task; model fills gaps from learned priors            |

# The Mystery Board comes down (2/2)

| #  | Mystery                              | Resolved | Mechanism, in one line                                             |
|----|--------------------------------------|----------|--------------------------------------------------------------------|
| 7  | CLAUDE.md compliance inconsistent    | L6       | Attention competition + positional effects                         |
| 8  | Different tools, different behaviour | L6+L7    | Different files, placement, loading, context → different sequences |
| 9  | Few-shot works                       | L6       | In-context learning via induction heads                            |
| 10 | Step-by-step helps                   | L6       | CoT uses the auto-regressive loop as a scratchpad                  |
| 11 | Voting improves accuracy             | L6       | Self-consistency: variance reduction over reasoning paths          |

## THE BOARD IS CLOSED

Every mystery from Act 1 has a mechanistic explanation.

Every mystery from Act 1 has a mechanistic explanation — there is **no magic**, and there never was.

AND IT REACHES PAST THE ELEVEN

Even the question the board never listed — **why is the context window finite?** — falls the same way: position, quadratic cost, KV cache. Architecture and deployment, not magic.

# The question changes

## ACT 2 ASKED

*Why does the system behave this way?* — answered by naming the mechanism.

## ACT 3 ASKS

*How do we engineer reliable software work on top of that behaviour?*

The question changes from **explanation** to **engineering** — budgets turned into policies, anecdotes upgraded to measurements, "it seems better" upgraded to a number.

# What Act 3 holds

## **Context engineering**

The budget, turned into a deliberate discipline.

## **AGENTS.md**

The cross-tool instruction-file standard.

## **Retrieval (RAG)**

External knowledge, selected by algorithm.

## **Validation + CI**

The enforcement layer, built out.

## **SWE-bench**

Act 3's first stop — measuring what these systems can do (contested — see L8).

### **NAMES ONLY — FOR NOW**

Each gets its own full treatment. Bring your redesigned CLAUDE.md next week.

# What you'll do in the lab

## THREE EXERCISES — ON YOUR OWN LOCUST FILES

1. **Context budget calculator** — measure your real token numbers, fill the tenant table.
2. **Buried-instruction experiment** — test the positional story empirically.
3. **CLAUDE.md redesign** — the capstone of Act 2.

Deliverables: a context-budget policy with numbers · an experimental record · a before/after CLAUDE.md with a rationale for every change.



Hochschule Aalen  
UNIVERSITY OF APPLIED SCIENCES

# Questions?

THE SYSTEM AROUND THE  
MODEL