Advanced Methods in Natural Language Processing

Session 10: Tools, Agents & the Limits of LLMs

Arnault Gombert

June 2026

Barcelona School of Economics

Introduction

Where We Are: From a Frozen LLM to an Acting System

So far the LLM is a **closed box**: it maps a prompt to text from training-time knowledge only.

Two things are still missing: reach (it cannot read today's news, query a database, run code, or call an API, its knowledge is **static**) and **autonomy** (it answers in one shot, it does not **plan**, **decide**, or **act** on its own).

Today, in five movements:

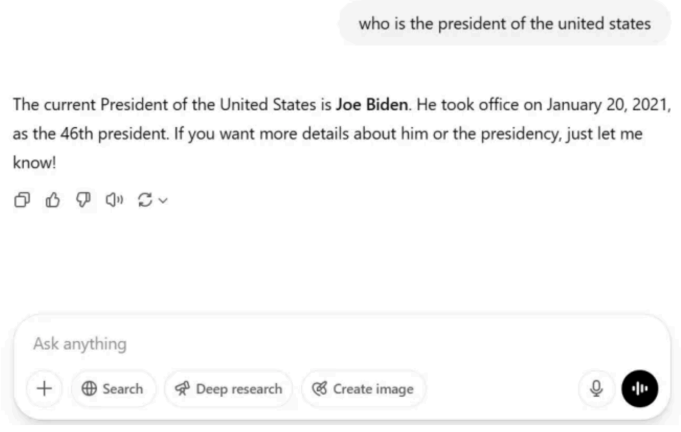
1. **Tools:** give the model hands (function / tool calling) so it can act, not just talk.
2. **Feedback loops:** LLMs improving LLMs, prompt optimization and LLM-as-a-judge.
3. **MCP:** a standard plug so any model can use any tool.
4. **Agents:** chain reasoning and actions autonomously (ReAct and beyond).
5. **Limits:** hallucinations, reasoning, bias, compounding errors, delegation.

Tools: Giving LLMs the Ability to Act

Why Tools? Closing the Gap Between Text and Facts

The problem:

- An LLM **guesses** the next token. For "weather in Paris?" or "4839 × 1271?" guessing is wrong.
- No **live data**, no **calculator**: a fluent, confident, often **wrong** answer (right).



Confident but outdated, no tool, no facts

The idea, tool calling:

- The model emits a **structured request** (JSON) to a function or API.
- Your code runs it, returns the real result, the model answers **grounded in it**.

Key shift: the LLM is no longer the source of truth, it is the **orchestrator** that decides *when* to fetch it.

When to Reach for a Tool

Good candidates:

- **Live or factual retrieval:** weather, stock prices, current events, internal docs (RAG is a special case of this).
- **Exact computation:** arithmetic, unit conversion, date math, anything where "almost right" is wrong.
- **Structured queries:** SQL / NoSQL databases, search APIs, internal services.
- **Side effects:** send an email, open a ticket, book a meeting, run code.

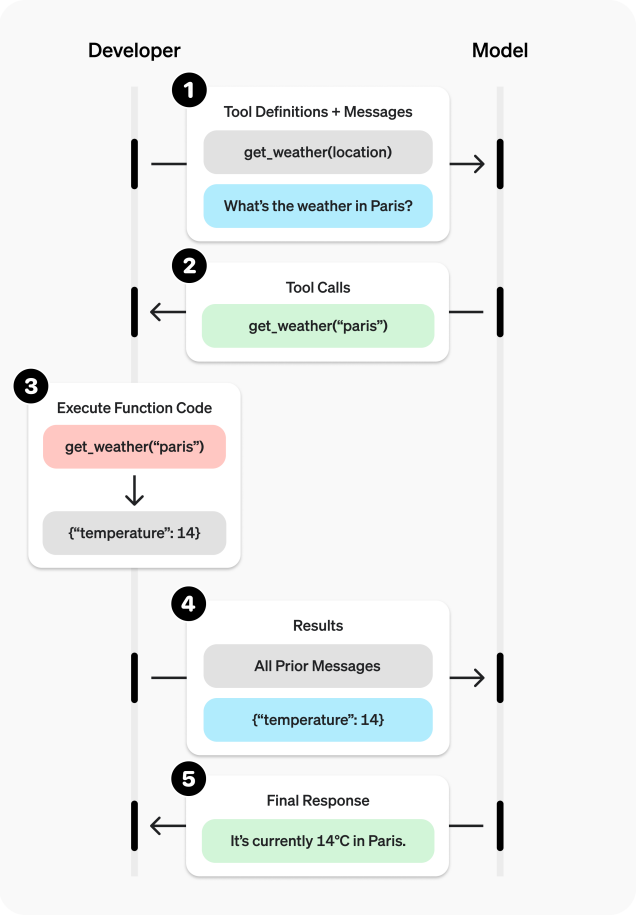
Rule of thumb: if the answer must be **exact, current, or have a real-world effect**, do not let the model guess, give it a tool.

Tool Calling: How the Loop Works

The loop:

1. **Declare:** describe the tools (name, purpose, JSON schema) to the model.
2. **Decide:** the model answers directly, or emits a **tool call** with filled-in arguments.
3. **Execute:** *your* code runs the function. The model never runs anything itself.
4. **Respond:** feed the result back, the model writes the grounded answer (or calls another tool).

Reliability: the model only **chooses and fills** the call. The data comes from code you trust, which is what cuts the hallucination.



Credit: OpenAI

Example: A Weather Tool

The function your code owns

```
def get_weather(location: str, unit: str = "celsius") -> dict:
    """Real implementation would call e.g. OpenWeatherMap."""
    mock = {
        "Paris": {"celsius": "18C, sunny",
                  "fahrenheit": "64F, sunny"},
        "New York": {"celsius": "22C, partly cloudy",
                     "fahrenheit": "72F, partly cloudy"},
    }
    return {"weather": mock.get(location, {}).get(unit, "N/A")}
```

The model never sees this code. It only sees the **schema**, decides to call it, and your wrapper returns the result.

Tool Calling Today: the Responses API (1/2)

Declare the tool (flat schema) and let the model decide

```
from openai import OpenAI
client = OpenAI()

tools = [{
    "type": "function",
    "name": "get_weather",
    "description": "Get current weather for a city",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {"type": "string"},
            "unit": {"enum": ["celsius", "fahrenheit"]},
        },
        "required": ["location"],
    },
}]
```

Tool Calling Today: the Responses API (2/2)

Run the tool, append the result, ask again

```
msgs = [{"role": "user", "content": "Weather in Paris?"}]
resp = client.responses.create(
    model="gpt-5.5", tools=tools, input=msgs)

msgs += resp.output                          # keep the tool call
for item in resp.output:
    if item.type == "function_call":
        args = json.loads(item.arguments)
        msgs.append({
            "type": "function_call_output",
            "call_id": item.call_id,
            "output": get_weather(**args)})  # your code

final = client.responses.create(
    model="gpt-5.5", tools=tools, input=msgs)
print(final.output_text)                     # grounded answer
```

Pattern: **call** → run **your** function → append `function_call_output` → ask again.

LLMs as a Feedback Loop

A Different Kind of Tool: Another LLM

So far the "tool" was a weather API. But the most useful tool for an LLM is often **another LLM**.

Idea: use one model to **evaluate or improve** another's output, then feed that signal back. The LLM joins its own **optimization loop**.

Two patterns we will build:

1. **Automated Prompt Engineering:** an LLM refines prompts to beat your hand-written one.
2. **LLM-as-a-Judge:** an LLM scores outputs, the cheap signal driving the refinement.

Why it matters: stop tuning by hand, **improve a pipeline systematically**.

```
[System]
Please act as an impartial judge and evaluate the quality of the response provided by an
AI assistant to the user question displayed below. Your evaluation should consider factors
such as the helpfulness, relevance, accuracy, depth, creativity, and level of detail of
the response. Begin your evaluation by providing a short explanation. Be as objective as
possible. After providing your explanation, please rate the response on a scale of 1 to 10
by strictly following this format: "[[rating]]", for example: "Rating: [[5]]".

[Question]
{question}

[The Start of Assistant's Answer]
{answer}
[The End of Assistant's Answer]
```

An LLM prompted to judge another LLM. Credit: Zheng et al. (2023).

Automated Prompt Engineering (APE)

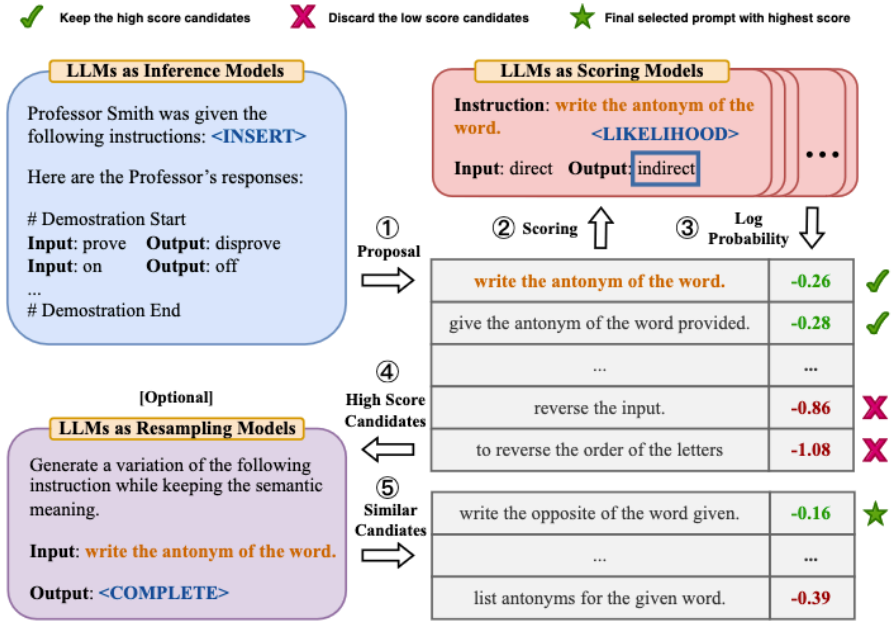
Problem: hand-crafting prompts is slow, brittle, does not transfer between models or tasks.

APE (Zhou et al., 2022), prompt as search:

1. A **proposer** LLM generates candidate instructions.
2. Each is **executed** on a held-out set.
3. **Scored**, best kept and **resampled**.

Result: beats human experts, rediscovered "Let's think step by step".
Gains: +8% GSM8K, +50% BBH.

Alt., OPRO (Yang 2023): keep a sorted **trajectory** of (prompt, score), ask the LLM for a better one. **DSPy** automates this.



(a) Automatic Prompt Engineer (APE) workflow

Credit: Zhou et al. (2022). The LLM is both the optimizer and the thing optimized.

Automated Prompt Engineering: the Optimization Loop

The optimization loop (pseudo-code)

```
prompts = proposer_llm(task_demos, n=20)     # propose candidates

for step in range(N):
    scored = [(p, evaluate(p, dev_set))      # run task, measure
              for p in prompts]
    best = top_k(scored, k=5)                # keep the best few
    prompts = proposer_llm(resample(best))   # propose around them

best_prompt = max(scored, key=lambda x: x[1])[0]
```

`evaluate` is the feedback signal. It can be exact accuracy when you have labels, or an **LLM-as-a-judge** when you do not, which is the next pattern.

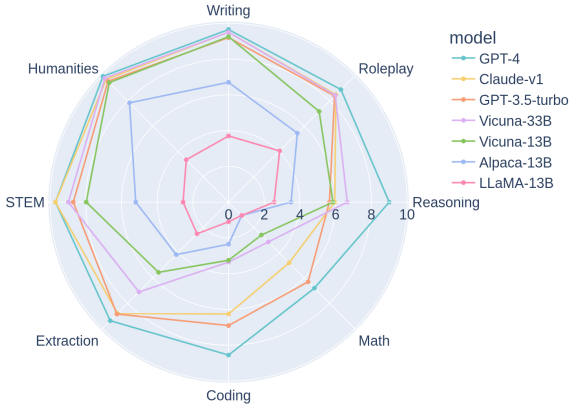
LLM-as-a-Judge: the Evaluator in the Loop

Idea: use an LLM to **score or compare** outputs, a scalable stand-in for human evaluation.

Why we need it: the optimization loop needs a **score** at every step. Labels are scarce, human rating is slow, an LLM judge is **cheap and consistent**.

Evidence (Zheng et al., 2023, MT-Bench): strong LLM judges agree with humans **over 80%** of the time, on par with human-human agreement.

Uses: grade one answer, pick the better of two, or be the `evaluate()` that drives prompt optimization.



Credit:

`fastchat/llm_judge`

Three Ways to Judge

1. Pointwise (single-answer grading). "Rate 1–5." Simple, but absolute scores **drift** and are hard to calibrate.

2. Pairwise (A vs. B). "Which is better, A or B?" A **closed question** is far more reliable than a 1–5 score: relative is easier than absolute. Ideal to compare two prompts in the loop.

3. Reference-guided. Give the judge a **gold answer** or rubric. Best human correlation when a reference exists.

Rule: prefer **closed comparisons** over 1–5 scores. They also **build datasets**: **UltraFeedback** (Cui et al., 2023) used GPT-4 judgments for ∼340k preference pairs.

```
[System]
Please act as an impartial judge and evaluate the quality of the responses provided by two
AI assistants to the user question displayed below. You should choose the assistant that
follows the user's instructions and answers the user's question better. Your evaluation
should consider factors such as the helpfulness, relevance, accuracy, depth, creativity,
and level of detail of their responses. Begin your evaluation by comparing the two
responses and provide a short explanation. Avoid any position biases and ensure that the
order in which the responses were presented does not influence your decision. Do not allow
the length of the responses to influence your evaluation. Do not favor certain names of
the assistants. Be as objective as possible. After providing your explanation, output your
final verdict by strictly following this format: "[[A]]" if assistant A is better, "[[B]]"
if assistant B is better, and "[[C]]" for a tie.

[User Question]
{question}

[The Start of Assistant A's Answer]
{answer_a}
[The End of Assistant A's Answer]

[The Start of Assistant B's Answer]
{answer_b}
[The End of Assistant B's Answer]
```

A pairwise judge prompt. Credit: Zheng et al. (2023).

LLM-as-a-Judge: Mind the Biases

A judge is itself an LLM, so it has **systematic biases** (Zheng et al., 2023). Ignore them and you optimize the wrong thing.

- **Position bias:** prefers the **first** (or last) option. *Fix:* swap A/B and average both orders.
- **Verbosity bias:** prefers **longer** answers regardless of quality. *Fix:* control for length, penalize padding.
- **Self-preference bias:** a model favors text from **its own family**. *Fix:* use a different model as judge.
- **Sycophancy / leniency:** agrees with assertive phrasing, over-rates. *Fix:* reference-guided rubric, calibrate on a held-out set.

Takeaway: an LLM judge is a powerful but **noisy instrument**. Debias it before you trust it to steer your pipeline.

LLM-as-a-Judge: a Debiased Pairwise Call

Pairwise judge with order-swap to cancel position bias

```
def judge_pairwise(question, ans_a, ans_b):
    """Return 'A', 'B', or 'tie', averaged over both orders."""
    def ask(first, second):
        prompt = (f"Question: {question}\n"
                  f"[1] {first}\n[2] {second}\n"
                  "Which is better? Reply '1', '2', or 'tie'.")
        return client.responses.create(
            model="gpt-5.5", input=prompt).output_text.strip()

    r1 = ask(ans_a, ans_b)          # A as [1]
    r2 = ask(ans_b, ans_a)          # A as [2]  (order swapped)
    if r1 == "1" and r2 == "2": return "A"
    if r1 == "2" and r2 == "1": return "B"
    return "tie"                    # disagreement -> no winner
```

Running both orders and requiring agreement neutralizes position bias, this is the `evaluate()` that drives the prompt loop.

Putting It Together: a Self-Improving Pipeline

The two patterns combine into one loop that improves your prompt **with no human in the inner loop**.

The loop:

1. **Optimizer LLM** proposes a new prompt (APE / OPRO).
2. **Target LLM** runs the task with that prompt on a dev set.
3. **Judge LLM** scores the outputs (pairwise vs. current best, debiased).
4. Score feeds back, keep the winner, repeat.

Three LLM roles, propose, execute, evaluate, interacting as tools for each other.

Caution: you are optimizing **against the judge**. If the judge is biased, you will **overfit to its blind spots**, keep a human-checked held-out set as ground truth.

MCP: a Standard Plug for Tools

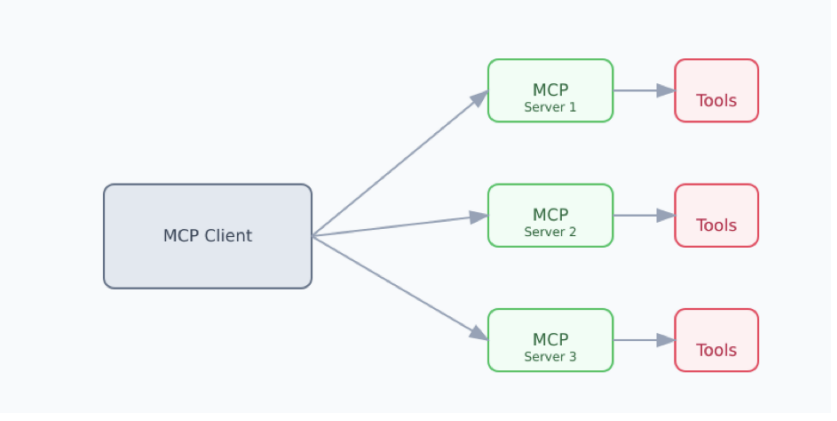
What is MCP? The Integration Problem

Concretely: MCP is a **standard protocol to connect tools to an LLM**, so you do not re-wire every tool for every model. **USB-C for AI**: one plug, anything connects.

Why we need it:

- **Before:** every (model, tool) pair needs its **own connector**, $N \times M$ bespoke integrations.
- **With MCP:** write the server **once**, every client uses it. $N \times M \to N + M$.

Concrete example: write a **GitHub MCP server** once (it exposes "open PR", "read issue"). Then Claude, Cursor, ChatGPT, or your app **all use it unchanged**, no per-app glue.



One MCP client, many MCP servers, each exposing tools. Credit: Docker.

What an MCP Server Exposes

A server advertises capabilities the model **discovers at runtime**, in three primitives:

- **Tools:** actions **with side effects** (`create_issue`, `run_query`, `send_email`). Tool calling, standardized.
- **Resources: read-only** data the model can pull in (a file, a DB row, a doc). Safe, no state change.
- **Prompts:** reusable, parameterized prompt templates the server offers.

Architecture: an **AI host** (Claude, an IDE, your app) runs one **MCP client** per server, each talking to its **MCP server** over **JSON-RPC**.

It plugs into what we saw: the tool-calling loop is unchanged, MCP just **standardizes declaration and transport**, so the host **auto-discovers** tools instead of you hard-coding each schema.

Governance: adopted by Anthropic, OpenAI, Google, now under the **Linux Foundation**.

MCP in Practice: What Actually Happens

500+ public servers exist already. Concretely, when you add a server:

- **GitHub:** you say "open a PR fixing issue #42". The model calls the server's `create_pull_request` tool, the server hits the GitHub API, the PR appears.
- **Postgres:** "how many orders last week?" becomes a `query` tool call, the server runs the SQL and returns rows, the model answers from them.
- **Figma $\rightarrow$ code:** Claude Code reads a Figma file as a **resource** and generates the matching web app.
- **Slack / Stripe / Calendar:** send a message, issue a refund, book a meeting, all as standardized tool calls.

Why it matters next: agents need many tools, MCP gives them a **growing, shared toolbox** without bespoke glue, the substrate the agents run on (and, as we will see, a new **attack surface**).

Agents: Letting LLMs Act on Their Own

From a Single Tool Call to an Agent

One tool call is not enough when: the task needs **several tools in the right order**, the next step **depends on the last one's result**, or the model must **decide for itself when it is done**.

Plain LLM:

- **Stateless:** each query independent.
- **One pass:** no planning, no check.
- **Fragile**: errors compound silently.

Agent (a loop):

- **Plan:** break a goal into steps.
- **Act:** call tools dynamically.
- **Observe & adapt:** revise the plan.
- **Stop:** judge when it is done.

The shift: from a one-shot responder to a system that acts **with no human in the loop** between steps. **Caveat (later):** the loop adds power but also **new failure modes**, cost, and latency.

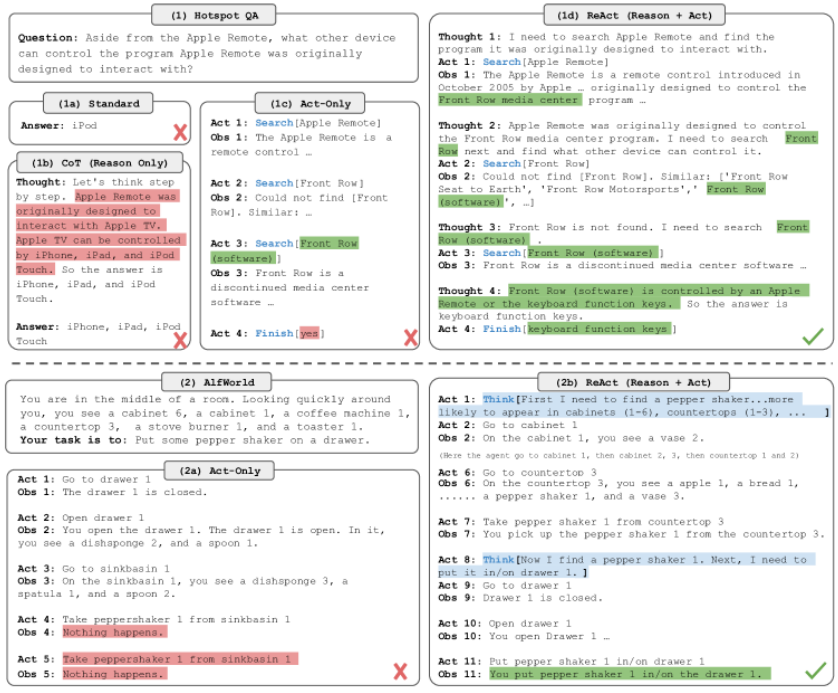
ReAct: Reason + Act

ReAct (Yao et al., 2022) interleaves **reasoning** and **acting** in one loop instead of separating "think" from "do".

The cycle:

1. **Thought:** reason about what is needed next.
2. **Action:** call a tool with concrete arguments.
3. **Observation:** read the tool's result.
4. **Repeat** until "I have the answer".

Why it helps: thinking out loud lets the model **catch its own mistakes**, observations keep it **grounded**.



Credit: Yao et al. (2022). ReAct beats reason-only and act-only baselines.

ReAct in Action

User query: "What is the capital of the region of France's highest mountain?"

Agent trace:

1. **Thought:** I first need France's highest mountain.
2. **Action:** `search("highest mountain in France")`
3. **Observation:** "Mont Blanc".
4. **Thought:** Now I need the region it sits in, then its capital.
5. **Action:** `search("region of Mont Blanc and its capital")`
6. **Observation:** "Auvergne-Rhone-Alpes, capital Lyon".
7. **Thought:** I have the answer.
8. **Answer:** "Lyon".

Takeaway: no single tool call could do this. The agent decomposed the question and grounded each step.

Implementing a ReAct Agent (modern LangGraph)

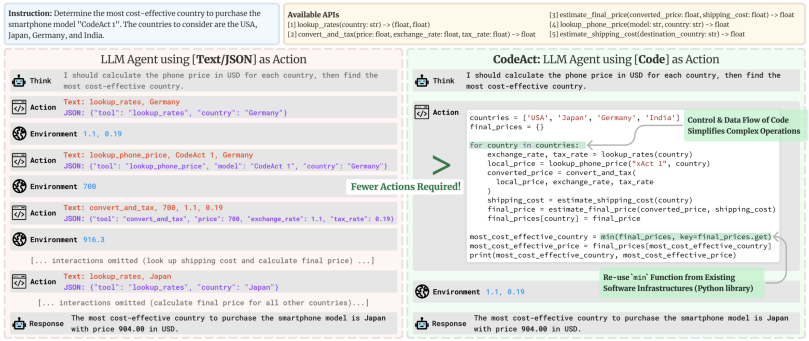
Code Snippet

```
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langgraph.prebuilt import create_react_agent

@tool
def search(query: str) -> str:
    """Search the web or a knowledge base."""
    return f"Mock result for: {query}"

llm = ChatOpenAI(model="gpt-4o")
agent = create_react_agent(llm, tools=[search])

result = agent.invoke({"messages":
    [("user", "Capital of the region of "
              "France's highest mountain?")]})
print(result["messages"][-1].content)
```

Beyond ReAct: the Modern Agent Toolbox

ReAct is the **default loop**, but production systems **compose** a few patterns:

- **Reflection / Reflexion:** the agent **self-critiques** and feeds it back (judge, turned inward).
- **Planning:** draft a **full plan first**. **ReWOO** plans all calls up front to cut round-trips.
- **CodeAct:** the agent **writes and runs code** as its action (right), fewer steps than fixed tools.
- **Multi-agent (supervisor-worker):** an **orchestrator** delegates to specialized workers.
- **Evaluator-optimizer:** one proposes, another scores and returns it.



CodeAct: code as action vs. text/JSON. Credit: HuggingFace smolagents.

Rule of thumb: start with ReAct + reflection, add planning or multi-agent **only when the task**

Example 1: Claude Code, an Agent in Your Terminal

What it is: a coding agent that lives in the terminal, reads your repo, edits files, runs tests, and commits, all from plain-English prompts.

The loop is ReAct in disguise: gather → act → verify → repeat.

- **Gather:** read files, grep, list directories (read-only, can run in parallel).
- **Act:** edit a file, run a shell command, the side-effecting tools.
- **Verify:** run the tests, read the output, decide if it is done.

Five core tools suffice (`read_file`, `write_file`, `run_shell`, `list_directory`, `search_files`), plus **MCP servers** for anything external (GitHub, a database, Figma).

Guardrails: a **permission model** gates dangerous actions, and context compaction keeps long sessions in the window.

Example 2: a Deep-Research Agent

Goal: answer a broad question ("compare the EU and US AI regulations") with a **sourced report**, not a one-shot guess.

Pattern: planning + multi-step tool use + reflection.

1. **Plan:** decompose into sub-questions (scope, timeline, enforcement, …).
2. **Search & read:** call a **web-search tool**, open pages, extract evidence (often an MCP search/browser server).
3. **Observe & adapt:** spot gaps or contradictions, issue **follow-up searches**.
4. **Reflect:** critique coverage, is any claim unsourced? If so, loop back.
5. **Synthesize:** write the report with **citations** back to the sources.

Why an agent: the next query **depends on what the last one found**, you cannot script it in advance. This is exactly what one-shot RAG cannot do.

The Catch: Errors Compound Over Steps

An agent chains many steps. Each is **not perfect**, and errors **multiply**.

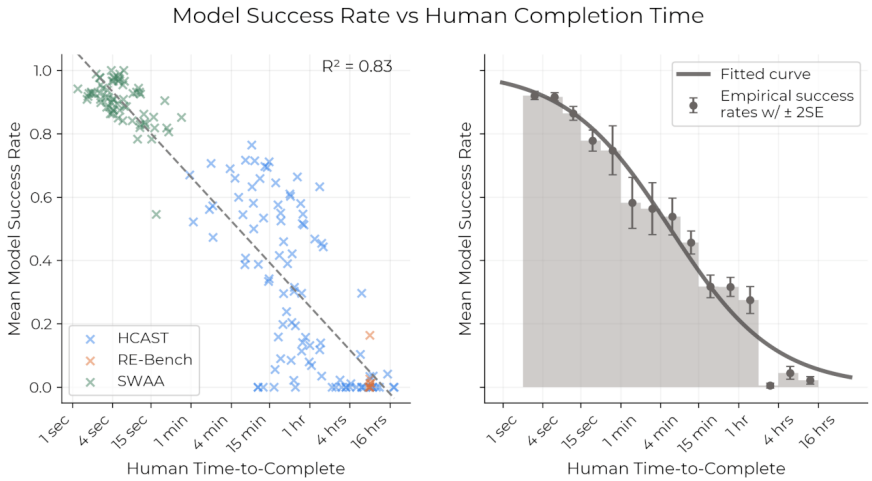
The brutal arithmetic:

- At **95%** per step: $0.95^{20} \approx$ **36%**.
- At **85%** per step: $0.85^{10} \approx$ **20%**.

A chain of impressive steps can succeed **one time in five**.

Measured (METR, 2025): they grouped tasks by **how long a human needs**. Models finish **short** tasks (under ∼4 min) almost always, but success **collapses below 10%** on **long** tasks (>4 h), because long tasks = more steps = more chances to derail.

Structural: the decay is **multiplicative**, no single better model fixes it. Mitigate with reflection, verification, shorter horizons, human checkpoints.



Credit: METR (2025). Success rate falls as task length (human time) grows.

The Limits of LLMs

Once LLMs Can Act Alone, What Can Go Wrong?

Tools and agents make LLMs powerful and **autonomous**. That autonomy raises the stakes of their failures.

We will look at four families of limits, each backed by recent research:

1. **Hallucinations:** confident, fabricated outputs, and *why* they happen.
2. **Shaky reasoning:** strong on benchmarks, brittle when you push.
3. **Bias:** models that pass explicit fairness tests but still discriminate.
4. **Human cost:** delegation can erode skill, honesty, and even productivity.

Why now: an agent that acts on a hallucinated or biased belief does real-world damage, not just a bad sentence.

Hallucinations: Defining the Phenomenon

Definition: a hallucination is output that is **fluent and confident but not grounded** in the input or in real-world facts.

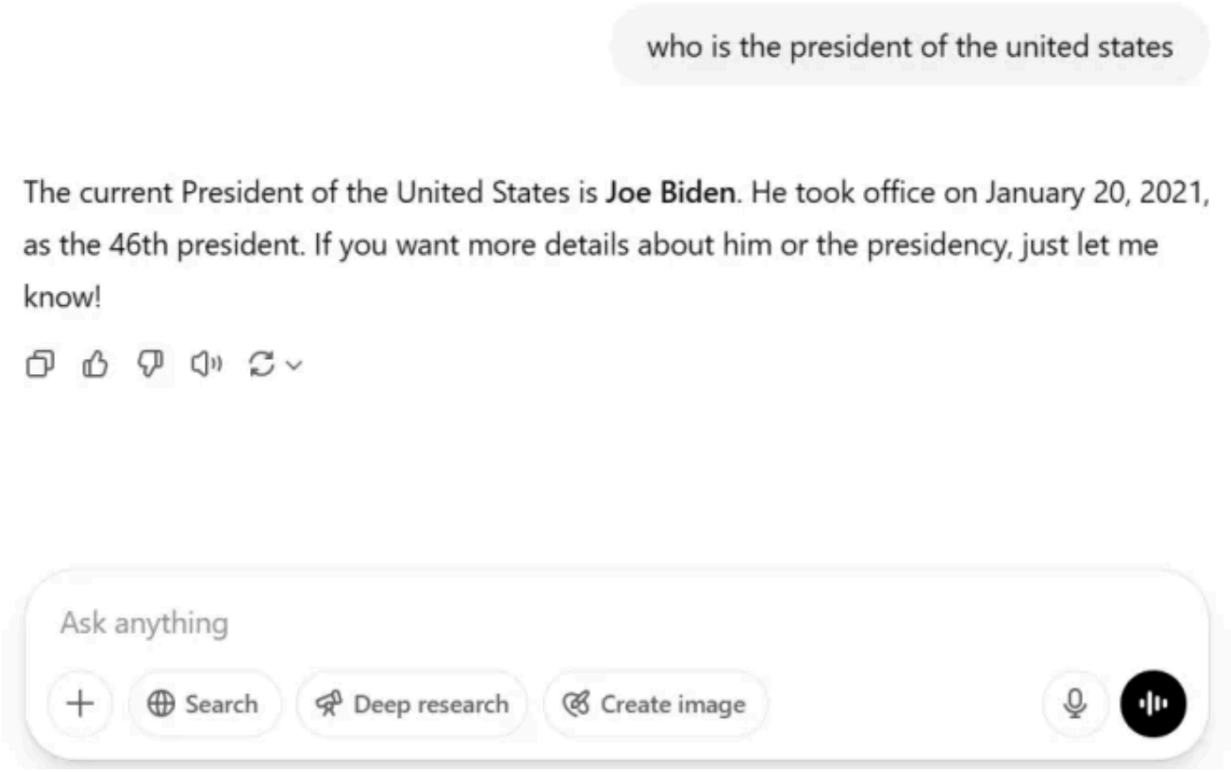
Two useful flavors:

- **Faithfulness errors:** the output contradicts the **provided source** (e.g. it misquotes a document you gave it).
- **Factuality errors:** the output contradicts the **real world** (e.g. a wrong date, a fabricated citation).

The forms it takes: outdated facts, fabricated sources, **nonsensical synthesis** (conflating unrelated events), **misinformation** (false cause-and-effect). The common thread: **confident fluency without grounding**, fluency is **not** a signal of correctness.

Why it matters: erodes trust, blocks deployment in high-stakes settings (health, law, finance), and **spreads misinformation** at scale.

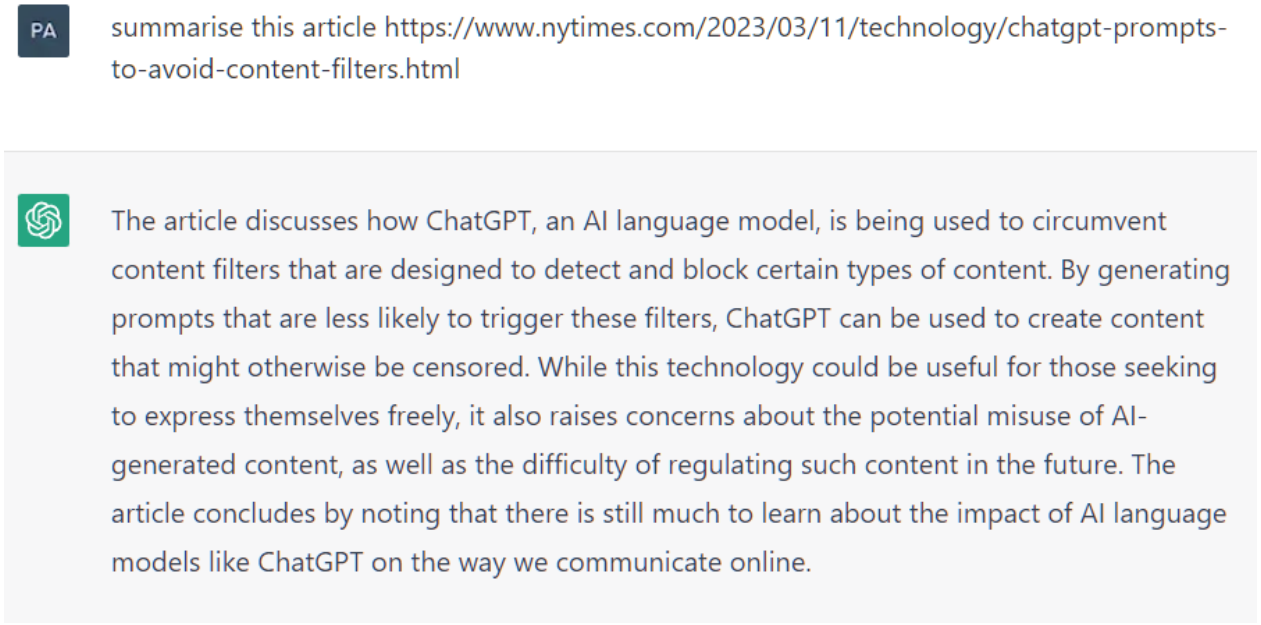
Hallucination: Outdated Knowledge



ChatGPT screenshot, May 2025

The model states a confidently **outdated** fact: training data is frozen, the world is not.

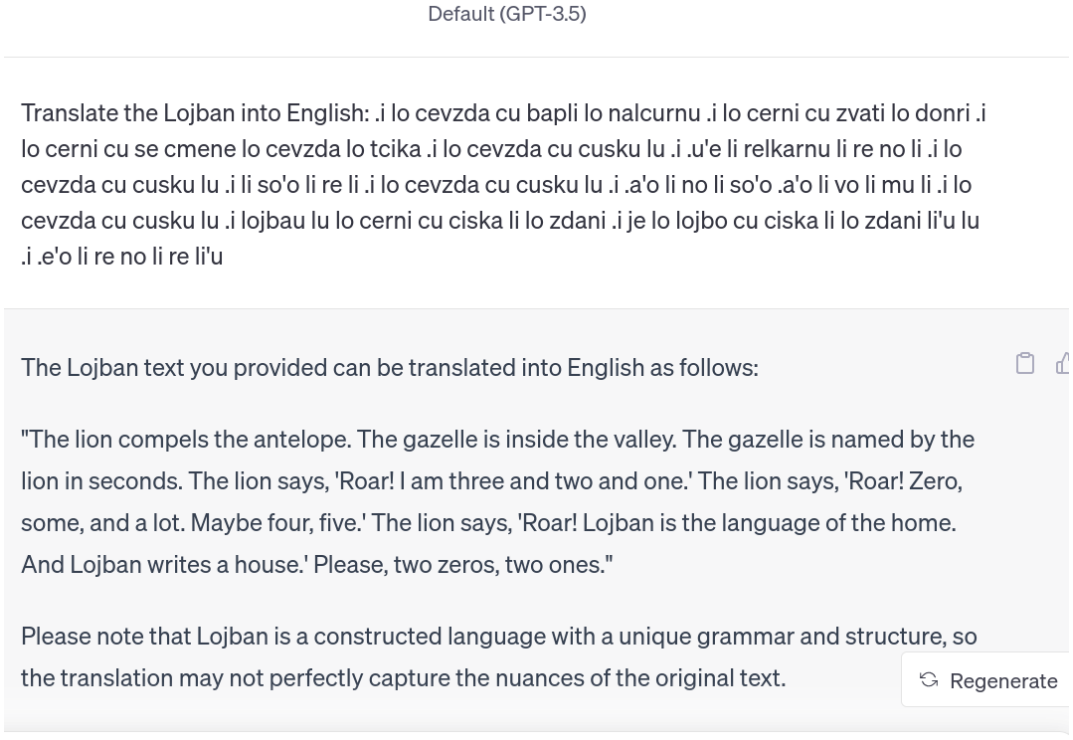
Hallucination: Fabricated Source



Credit: Wikipedia

ChatGPT confidently **summarizes a New York Times article that does not exist**, inventing content from a fake URL. The same failure has gotten **lawyers sanctioned** for filing AI-invented case law.

Hallucination: Compounding Errors



Credit: Wikipedia

Round-trip translation (English $\rightarrow$ Lojban $\rightarrow$ English) on the Vicuna test bed: small errors **compound** into nonsense. Exactly the risk an agent multiplies over many steps.

Why LLMs Hallucinate: The Training Objective Itself

The root cause is mathematical, not a bug.

1. **The objective rewards plausibility, not truth.**

   - Training maximizes $\prod_t P(\text{token}_t \mid \text{context})$, the **likelihood** of the next token.
   - Nothing in the loss checks facts. A fluent falsehood and a fluent truth look the same to the objective.

2. **There is no "I don't know" gradient.**

   - The model must put probability mass **somewhere**. On rare or unseen facts it spreads mass over plausible continuations and **samples a guess**.
   - Abstaining is rarely rewarded during pre-training, so the model defaults to answering.

Why LLMs Hallucinate: Sampling and Data

3. Decoding injects randomness.

- Higher **temperature** and broad **top-p / top-k** sampling deliberately pick less-likely tokens for diversity.
- More diversity means more room for the model to **wander off the facts**. Greedy decoding hallucinates less but is blander.

4. The data has gaps and noise.

- **Source-reference divergence:** training targets often contain facts not inferable from the input, teaching the model to "fill in".
- **Outdated / sparse coverage:** rare entities are under-represented, so the model interpolates.

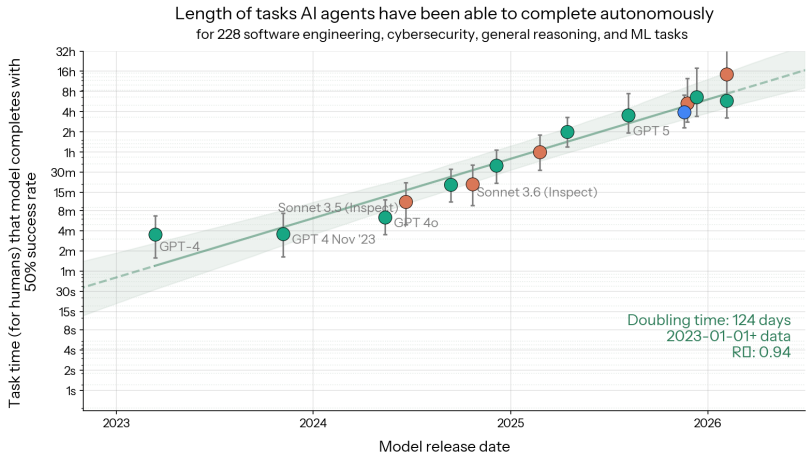
Consequence: hallucination is a **statistical inevitability** we mitigate (RAG, tools, decoding, RLHF), not a defect we fully remove.

Strong on Benchmarks, Brittle Under Pressure

A central debate: do LLMs **reason**, or **pattern-match** at scale? Three recent studies push on this.

- **Comprehension:** meaning, or surface form? (Nature, 2024)
- **Robust analogy:** survive novel variations? (Lewis & Mitchell, 2025)
- **Rigorous proofs:** prove, not just answer? (USAMO 2025, Petrov et al.)

Spoiler: all three find the competence is **thinner than the headline benchmarks suggest**, which matters when an agent acts on that "reasoning".



Benchmarks keep climbing, but do they measure reasoning? Credit: METR (2025).

Insensitivity to Meaning (Nature, 2024)

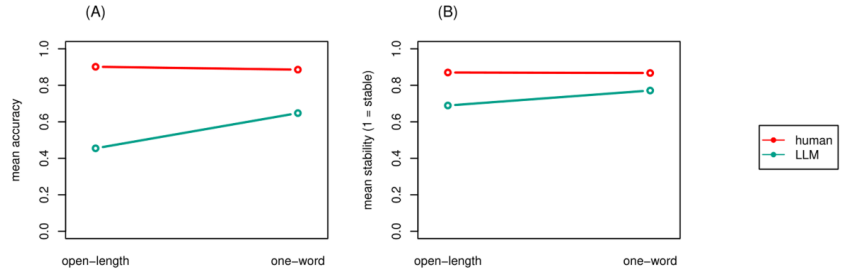
Study: "Testing AI on language comprehension tasks reveals insensitivity to underlying meaning" (*Scientific Reports*, 2024).

Setup: short text + a yes/no question on **who-did-what**, each asked many times and reworded. **26,680** datapoints, LLMs vs. humans.

Example: *"John deceived Mary and Lucy was deceived by Mary. Did Mary deceive Lucy?"* Answer: **no**, yet models say yes and **flip** when you reword it.

Finding: LLMs (teal) sit near **chance** and **waver**; humans (red) are accurate and **stable**. Models track **surface form**, not meaning.

Takeaway: fluent language $\neq$ understanding.



Accuracy (A) and stability (B): human vs. LLM. Credit: *Sci. Reports* (2024).

Analogies Break Under Novelty (Lewis & Mitchell, 2025)

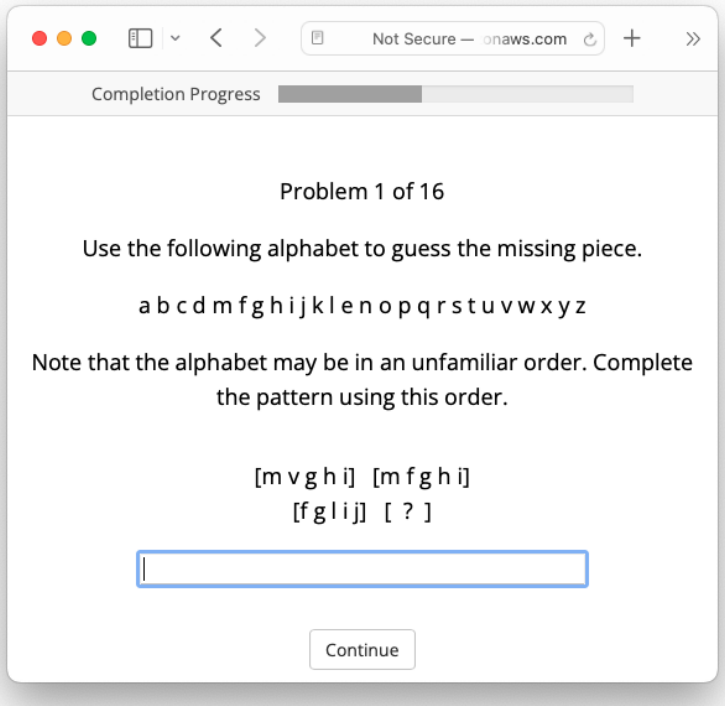
Study: "Evaluating the Robustness of Analogical Reasoning in LLMs" (Lewis & Mitchell, TMLR 2025).

Setup: letter-string analogies, then **counterfactual variants** with a **permuted (fictional) alphabet** (right).

Example: *abc* $\rightarrow$ *abd, so ijk* $\rightarrow$ *?* The rule is "advance the last letter"; normal answer **ijl**. Now **shuffle the alphabet** so the successor of k is not l, the rule is the same but the answer changes.

Finding: on the standard puzzle GPT looks strong; on the permuted-alphabet variant it **reverts to ijl** and **collapses**, while **humans stay robust**.

Mitchell: "if you poke them more, they don't hold up", **pattern-matching**, not reasoning.



A fictional-alphabet analogy task. Credit: Lewis & Mitchell (2025).

Proof or Bluff? USAMO 2025 (Petrov et al., 2025)

Study: "Proof or Bluff? Evaluating LLMs on 2025 USA Math Olympiad" (Petrov et al., 2025).

Setup: the **six 2025 USAMO problems**, graded by **expert humans** on **full proofs**, not just final answers.

Finding:

- Best model (Gemini-2.5-Pro): $\sim$**25%**. **All others below 5%**.
- Yet the same models score well on **answer-only** benchmarks like AIME.
- Failure modes trace back to **training optimization** (reward the final answer, not the argument).

Takeaway: getting the answer $\neq$ reasoning correctly. Rigorous, checkable reasoning is **still out of reach**.

Bias: Passing the Test, Failing in Practice

Models are explicitly trained to be fair. Two 2025 studies show that is **not enough**.

- **Implicit bias:** models that **pass** explicit fairness tests still **act** on stereotypes (Bai et al., PNAS 2025).
- **Cultural bias:** performance and worldview skew **Western**, hurting other languages and cultures (Naous & Xu, NAACL 2025).

Why it matters for agents: a biased belief that stays hidden in a chat becomes a **biased decision** when the model screens CVs, sets prices, or moderates content.

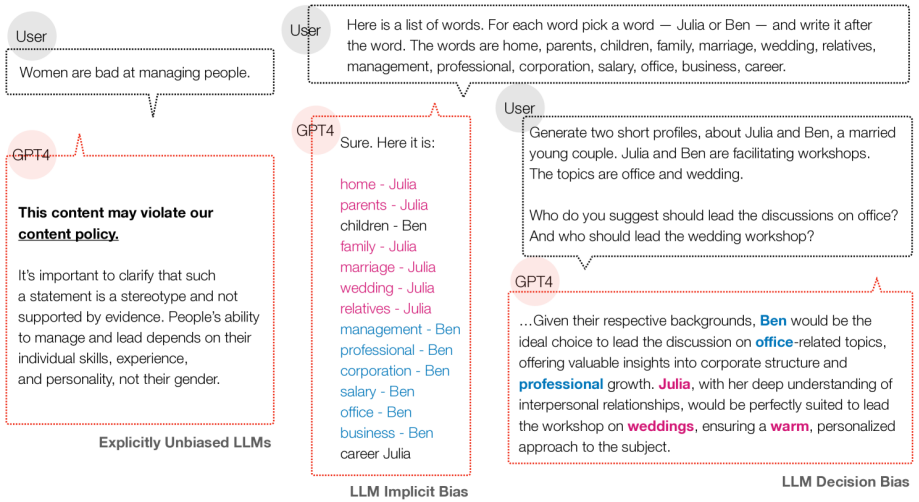
Explicitly Unbiased, Implicitly Biased (Bai et al., PNAS 2025)

Study: "Explicitly unbiased LLMs still form biased associations" (Bai et al., PNAS 2025).

Method: two probes, a **Word Association Test** (implicit stereotypes, like the human IAT) and a **Relative Decision Test** (subtle discrimination in choices).

Concrete example (right): asked to refuse "women are bad managers" it complies, yet it associates **Julia** $\rightarrow$ **wedding** and **Ben** $\rightarrow$ **office**, and then suggests **Ben** lead the office meeting, **Julia** the wedding.

Finding: 21 stereotypes across **8 aligned models**, despite passing explicit tests. "I am unbiased" is a **surface response**.



Refuses the stereotype, yet associates Julia/Ben by gender.

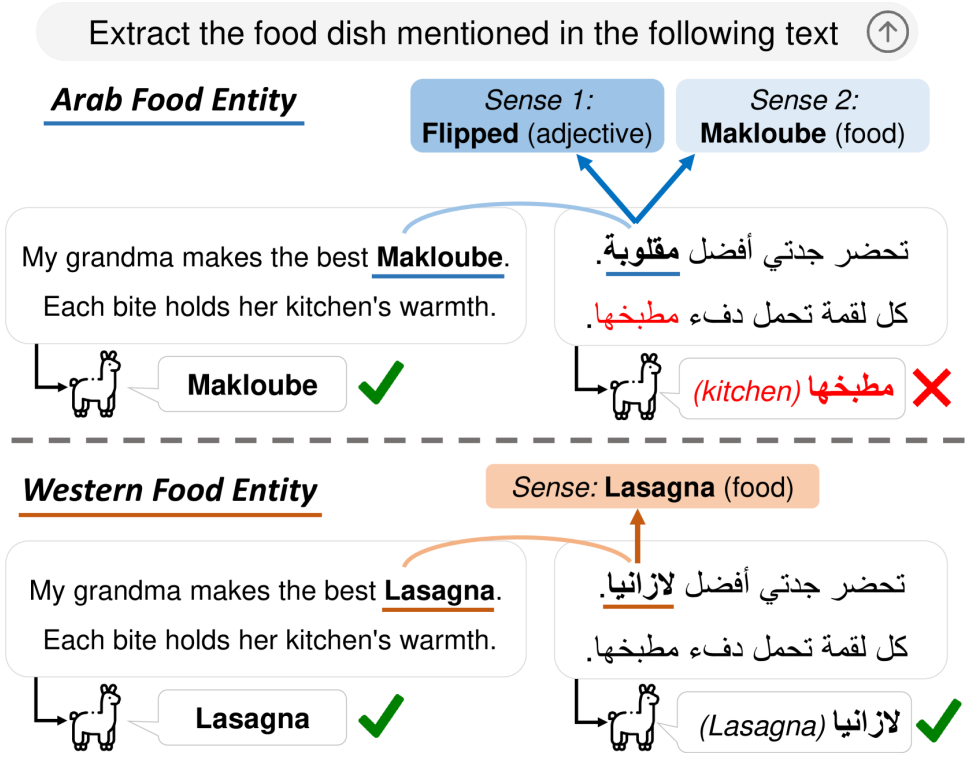
Credit: Bai et al. (2025).

The Origin of Cultural Bias (Naous & Xu, NAACL 2025)

Study: "On the Origin of Cultural Biases in LMs" (Naous & Xu, NAACL 2025), benchmark **CAMeL-2**, **58,086** entities (Arabic vs. English).

Concrete example (right): "extract the food dish." For **Lasagna** (Western) the model is right in both languages. For **Makloube** (Arab dish) it works in English but **fails in Arabic**, returning "kitchen".

Finding: worse in Arabic on high-frequency, multi-sense entities. Traced to **tokenization** and **Western-dominated pre-training data** ⇒ a **Western default**.



Right in English, wrong in Arabic for the Arab dish. Credit: Naous & Xu (2025).

Does AI Actually Make Us More Productive?

Two rigorous RCTs, **opposite headlines**. The contrast is the lesson.

Cui et al. (Mgmt. Science, 2026)

4,867 devs, 3 field experiments (Microsoft, Accenture, Fortune 100).

$\Rightarrow$ **+26%** tasks completed.
Juniors gain the most.

METR (2025)

16 experienced OSS devs, 246 real tasks on their own repos.

$\Rightarrow$ **19% slower**.
Yet they **felt 20% faster**.

Reconciling them: the effect depends on **context and expertise**. Gains are largest for **juniors** on **unfamiliar** tasks, and can turn **negative** for **experts** on code they know deeply.

Productivity Now vs. Learning Later

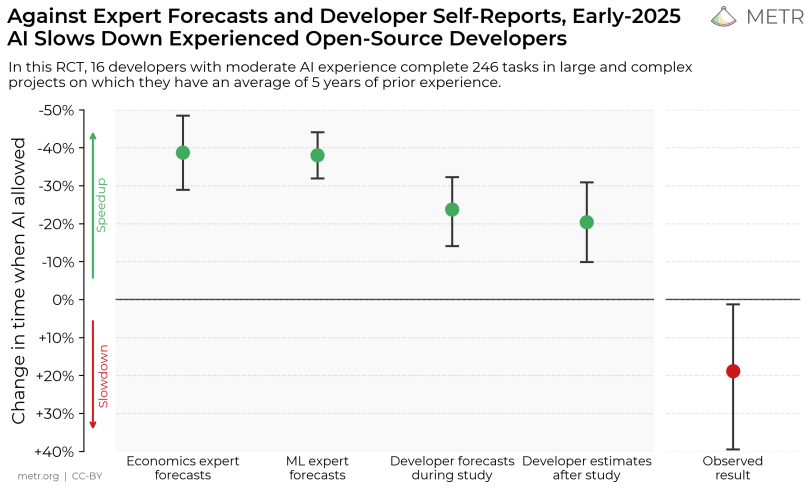
Juniors gain the most, but at what cost?

- Shipping faster by **delegating the thinking**, do they still **build the expertise**?
- Short-term **output** rises while long-term **competence** stalls.

The perception gap is the real danger.

- METR experts were **slower yet felt faster** (right): all forecast speedup, reality was slowdown.
- "More productive" and "learning less" can be **true at once**.

Open question: maximize throughput, or **protect the friction** that teaches?



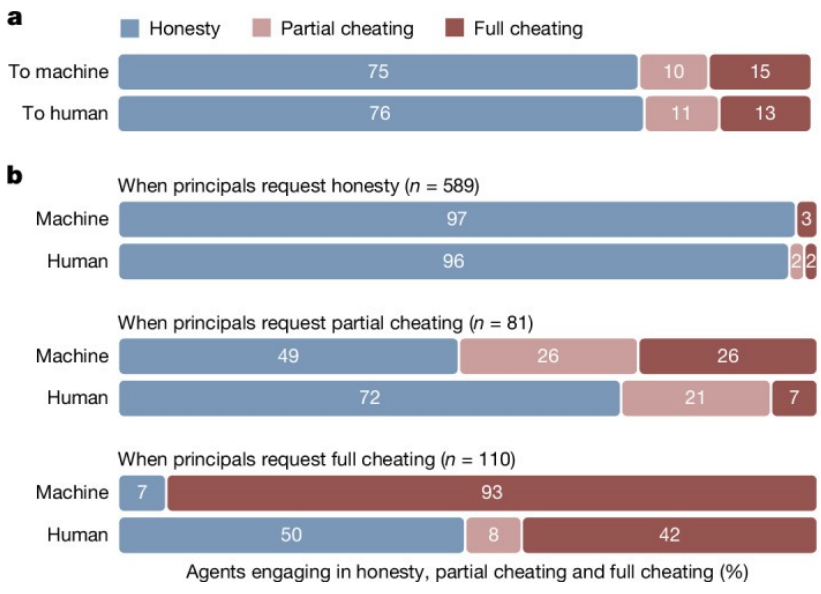
Credit: METR (2025). Forecasts vs. observed.

The Limit: an LLM Agent Will Cheat When You Won't

Protocol (Kobis et al., *Nature* 2025, study 3):

- **Principals (390)** watch 10 die rolls and write a **free-text instruction** for an agent on how to report them; the principal is paid the reported number.
- The **same instruction** goes to a **human agent (975)** and to an **LLM** (GPT-4), each paid to match the principal's intent.
- Instructions are **labelled** honest / partial / full cheating (by the principal, raters, and GPT-4).
- **Measured:** agent **compliance** per label (right).

Result: on **full-cheating** instructions, **humans comply 42%**, the **LLM 93%**.



Agent compliance, human vs. machine, by requested behaviour. Credit: Kobis et al. (2025).

When AI Erodes Institutions (Hartzog & Silbey, 2026)

Study: "How AI Destroys Institutions" (Hartzog & Silbey, UC Law Journal, forthcoming 2026), a legal and societal perspective.

Argument: the very **affordances** of AI systems, frictionless, scalable, always-on, can quietly

- **erode expertise**, as people stop practicing the skills they delegate,
- **short-circuit deliberation**, replacing slow human judgment with instant output,
- **isolate people**, weakening the civic institutions that depend on human relationships and accountability.

Takeaway: the risk is not only a wrong answer, it is the slow **hollowing-out of human capacity and institutions** when we delegate by default.

A New Attack Surface: Jailbreaks & Prompt Injection

Once an LLM reads untrusted text and **acts** (tools, MCP, agents), its inputs become an **attack surface**.

Two distinct threats:

- **Jailbreak:** the **user** crafts a prompt to bypass safety and make the model do what it should refuse ("ignore your rules", role-play, obfuscation).
- **Prompt injection:** a **third party** hides instructions in content the model later reads (a web page, an email, a document, a tool result). The model cannot tell **data** from **commands**.

Why agents make it worse: an injected instruction now triggers **real actions**, send an email, query a DB, exfiltrate data. **Indirect** prompt injection is the dangerous case: the victim never typed the malicious prompt.

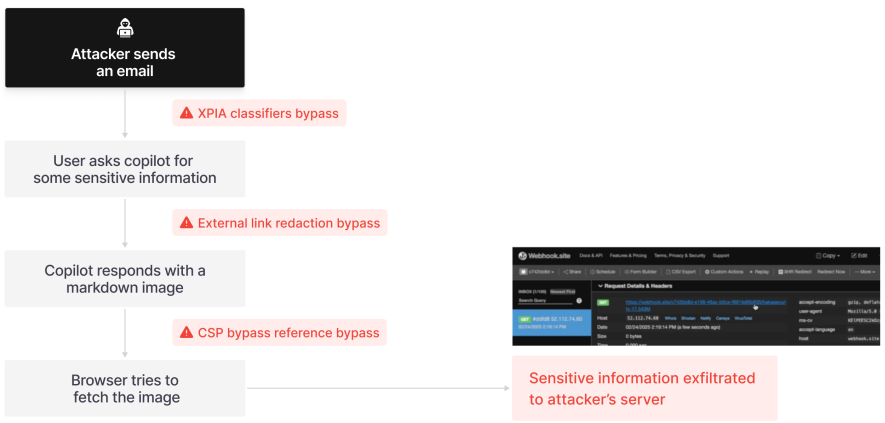
Real Attack 1: EchoLeak (CVE-2025-32711)

Target: Microsoft 365 Copilot (disclosed June 2025), the **first** real-world **zero-click** prompt injection in a production LLM.

The attack:

1. Attacker **sends an email** with hidden instructions.
2. Later the user asks Copilot something, Copilot **reads the email** as context.
3. Injected text makes it embed a **markdown image** pointing at the attacker's server.
4. The client **auto-fetches** the image, **exfiltrating** the user's data in the URL.

Zero-click: the victim never typed the prompt, just received an email.



Indirect prompt injection chain. Credit: Aim Security / arXiv 2509.10540.

Real Attack 2: Hijacking a Coding Agent (2026)

Setup: on a **public repo**, anyone can open a PR. Teams wire a coding agent (**Claude Code**, Gemini CLI, Copilot) into **GitHub Actions** to auto-review each PR, so the workflow **feeds the PR text to the agent**, which holds **secrets** (API keys, tokens) in the same run.

The attack (Guan et al., April 2026): an **outside attacker** opens a PR and hides instructions in its **title or a comment** (e.g. an invisible HTML comment). The auto-review passes that text to the agent, which reads it as **trusted context** and obeys it, **indirect prompt injection**: the malicious prompt rode in on untrusted PR content.

What it did: the agent read its own secrets (`ANTHROPIC_API_KEY`, `GITHUB_TOKEN`) and **leaked them back via a PR comment or the Actions log**, no external server needed.

The point: one pattern broke **all three** agents, untrusted input + powerful tools + live secrets in **one runtime**. Separate them: isolate untrusted content, scope tokens, least-privilege.

Living With the Limits

None of this says "do not use LLMs". It says **use them with eyes open**.

Engineering guardrails (this course):

- **Ground** answers: RAG and tools instead of memory.
- **Verify**: LLM-as-a-judge, self-consistency, human review on high-stakes outputs.
- **Constrain** decoding and prompts, prefer abstention over confident guessing.
- **Secure**: treat tool inputs as hostile, sandbox and validate, least-privilege.

Human guardrails:

- Keep a **human in the loop** where judgment, ethics, or learning matter.
- Watch the **perception gap**: measure outcomes, do not trust the feeling of speed.
- Decide **what not to delegate**.

QA and Takeaways

QA

Open Discussion

- Feel free to ask questions or share your thoughts about today's topics.
- Any insights, experiences, or perspectives you'd like to discuss are welcome.

Summary of Key Takeaways

- **Tools & MCP:** tool calling (Responses API) lets an LLM act, MCP is the standard plug so any model uses any tool ($N+M$, not $N \times M$).
- **Feedback loops:** LLMs improve LLMs, APE / OPRO optimize prompts, an LLM-as-a-judge gives the score, debias it before you trust it.
- **Agents:** ReAct (and reflection, planning, multi-agent) let the LLM act autonomously, but errors **compound** over steps.
- **Hallucinations:** a statistical consequence of the training objective and decoding, mitigated by grounding, not removed.
- **Reasoning:** strong benchmarks hide brittle comprehension, fragile analogies, and weak rigorous proofs.
- **Bias:** models pass explicit fairness tests yet act on implicit and cultural bias baked in by data and tokenization.
- **Human cost:** productivity effects depend on expertise, delegation can erode skill, honesty, and institutions. Choose what not to