Modern AI Systems: Agents and System Optimizations

Agent Design I: context management and tool design

Juncheng Yang

Harvard John A. Paulson School of Engineering and Applied Sciences

HARVARD UNIVERSITY MaDSys

Measurement and Design of Systems Lab

Logistics

- Assignment 1 due this Sunday
  - Share your repo
  - Fill in Google form
- Assignment 1 sharing sign up this Thursday

Plan for today

- Agent loop (ReACT)
- Tool designs
- Context management
  - prompt engineering
  - context engineering

The ReAct loop

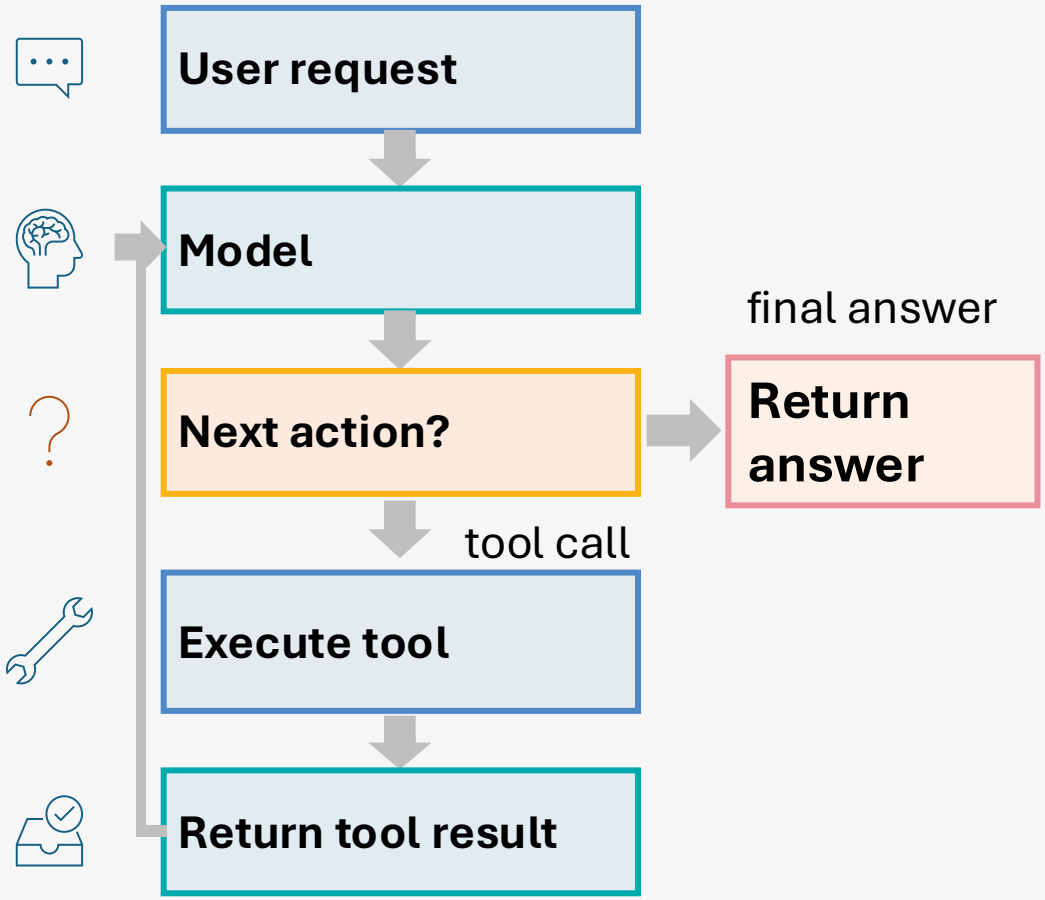
Reason and **Act** in one loop

Thought	Action	Observation
What do I know, what is missing, what next?	Emit a structured call the environment can run.	The environment answers with facts.

Reason only	Nothing ever checks the reasoning; small errors compound into confident hallucination.
Act only	Tool calls with no plan behind them; the model repeats itself.
ReAct	Thoughts steer the next action; observations correct the next thought.

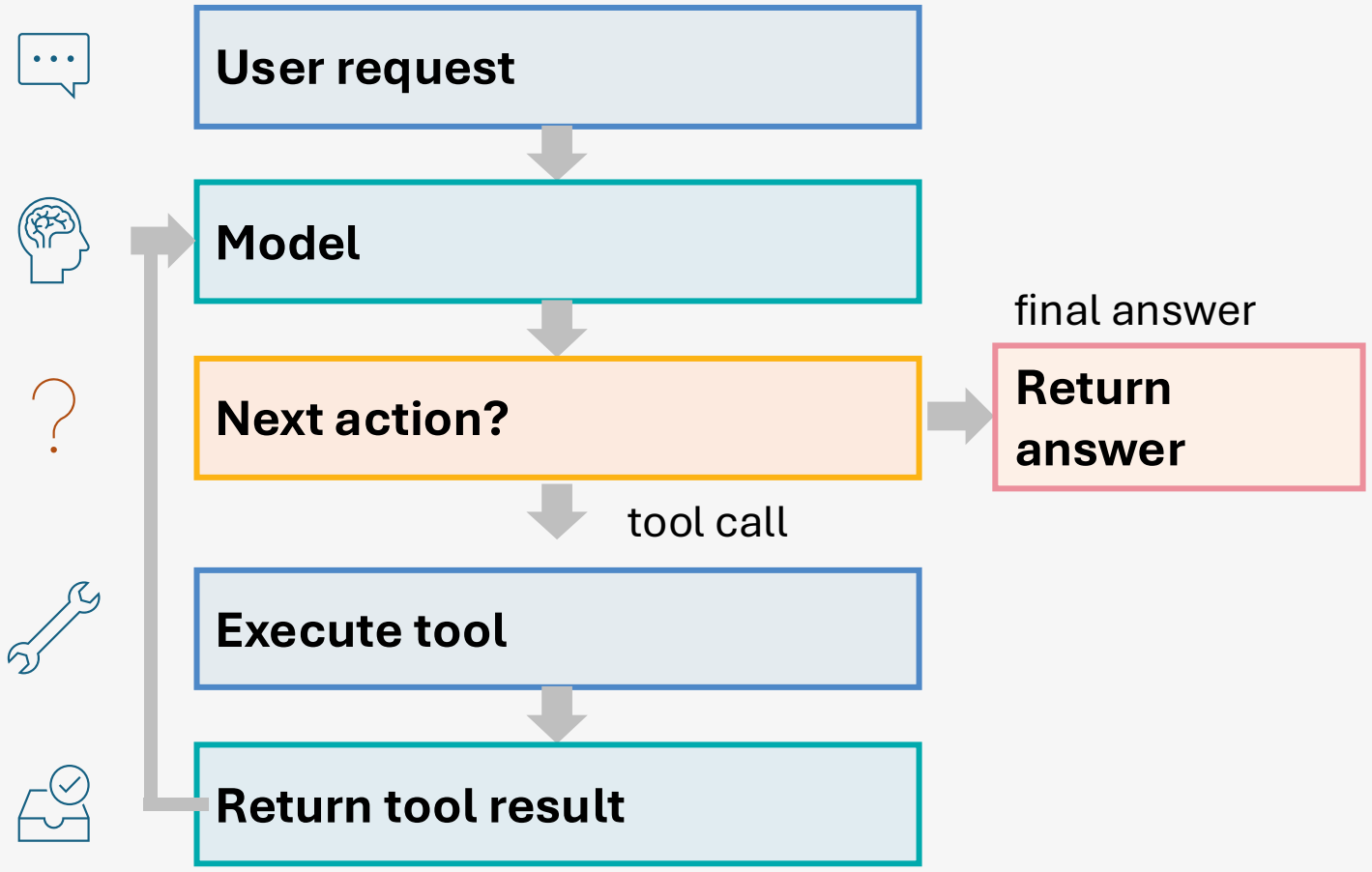
Repeat until the observations answer the question — or a stopping rule ends the run.

The agent loop




```
messages = [
    {"role": "system", "content":
        "Solve the task using the tools."},
    {"role": "user", "content": user_request},
]
for step in range(MAX_STEPS):
    response = call_model(messages, tools)
    messages.append(response.message)
    if response.final_answer:
        return response.final_answer
    for call in response.tool_calls:
        args = validate(call.arguments)
        result = execute_tool(call.name, args)
        messages.append({
            "role": "tool",
            "tool_call_id": call.id,
            "content": serialize(result),
        })
raise RuntimeError(
    "Agent exceeded its step limit")
```

The agent loop



What you will do as an agent harness designer

Write tool definitions

Assemble the context

Validate request, check permission and run it

Optionally process the result and manage context

Components of a coding agent

Three components, one context window — design them together

01

Model + agent loop

The LLM proposes an action, sees the result, and repeats until done.

→ An agent is a model plus a loop

02

Tools and guardrails

Search, read, edit, run tests and shell — behind permissions, sandboxing and approval.

→ Tools set capability; guardrails keep it safe

03

State: context and memory

What is in the window now, plus what the agent carries forward between steps.

→ Assemble it; keep what outlives the turn

What Tools Should We Provide to Agents?

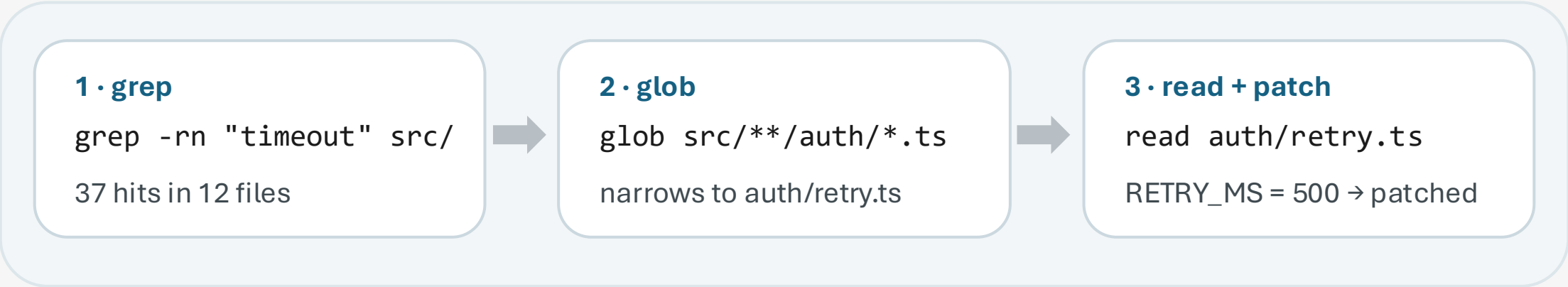
- Search / Read
- Edit / Write
- Execution
- Others

How Should An Agent Find Relevant Code?

Tool search and read: glob and grep

- Agent loops over ls, glob and grep — nothing to build or keep in sync
- Always reflects the live repo
- Cost: burns turns and context; lexical matching misses paraphrase

Example — “fix the login timeout”

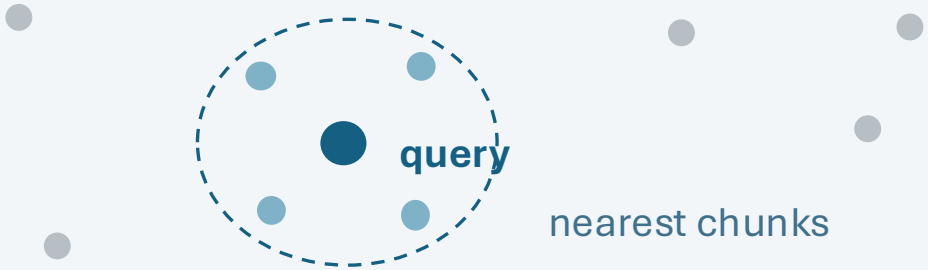


Tool search and read: semantic search

- Semantic: embed code chunks, retrieve by meaning
- Structural: file tree plus ranked symbols (Aider's repo map)
- Fast at query time; cost is building and re-syncing the index

Example — "where do we rate-limit?" returns throttle.ts on the first query

Embedding search



Same meaning, different words — vectors land close

Structural search — repo map

```
aider/coders/base_coder.py:
  class Coder:
    def apply_updates(self)
aider/commands.py: class Commands
```

Ranked outline of files, classes and signatures

Tool search and read: extrinsic signals

- Evidence outside the source: stack traces, failing tests, git history
- Highest-precision signal when debugging
- Only helps when such a signal exists

Example — crash report on checkout

1 · read the signal

`stack trace: cart.ts:88`

3 frames named

2 · open those frames

`read tax.ts:20-60`

rounding in applyTax

3 · patch + re-run

`pytest tests/tax.py`

green, no search used

Tool search and read: language server (LSP)

- Queries the compiler’s index: definitions, references, symbols, types
- Resolves imports, overloads and renames that grep only guesses at
- Cost: a live server per language; slow cold start

Example — rename charge() to capture()

1 · find references

`textDocument/references`

42 call sites, 7 files

2 · rename the symbol

`textDocument/rename`

aliased imports included

3 · verify

`type check`

no unresolved symbols

Search and read: a comparison

Approach	Finds code by	Best when	Main cost

In practice — start with grep; reach for an index or LSP when names run out

Search and read: a comparison

Approach	Finds code by	Best when	Main cost
glob / grep	literal names and paths	the name is known or guessable	turns and context per search
Semantic index	meaning of embedded chunks	wording won't match the code	building and re-syncing the index
Extrinsic signals	traces, failing tests, git history	a report already points at code	only exists for some tasks
Language server	the compiler's symbol graph	edits must follow references	a live server per language

In practice — start with grep; reach for an index or LSP when names run out

How Should An Agent Edit Code?

Tool edit: whole-file rewrite

- Model regenerates the whole file; the tool overwrites it
- No patch syntax to get wrong — every edit applies cleanly
- Cost: output scales with file size; untouched code can drift

Example — add a retry to the API client

1 · read the file

`read client.ts`

120 lines into context

2 · regenerate

`write client.ts`

full file, 4 lines changed

3 · result

`diff vs previous`

applies on the first try

Tool edit: unified diff / patch

- Model emits a patch; a separate tool applies it
- Compact, reviewable, the format humans and CI already use
- Cost: hunks fail when line numbers or context drift

Example — bump the retry limit in two files

1 · read for anchors

`read client.ts, api.ts`

line numbers in hand

2 · emit the patch

```
@@ -41,3 +41,3 @@
```

6 lines, two hunks

3 · apply

`git apply patch.diff`

both hunks applied

Tool edit: search and replace

- Model emits an exact anchor string plus its replacement
- Output is proportional to the change, not the file
- Cost: fails on ambiguous, stale or whitespace-off anchors

Example — raise the retry count

1 · locate the anchor

`grep -n "retries = 3"`

one unique match

2 · replace the block

`retries = 3 → 5`

one call, no rewrite

3 · verify

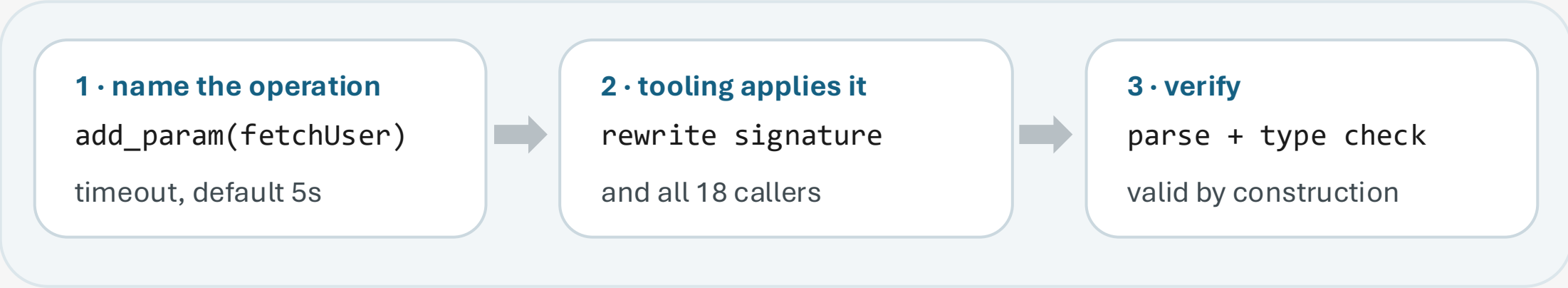
`read back the block`

rest of file untouched

Tool edit: structured / AST edits

- Model names an operation on a symbol: rename, add parameter
- Immune to formatting; cannot produce invalid syntax
- Cost: per-language tooling; limited set of expressible edits

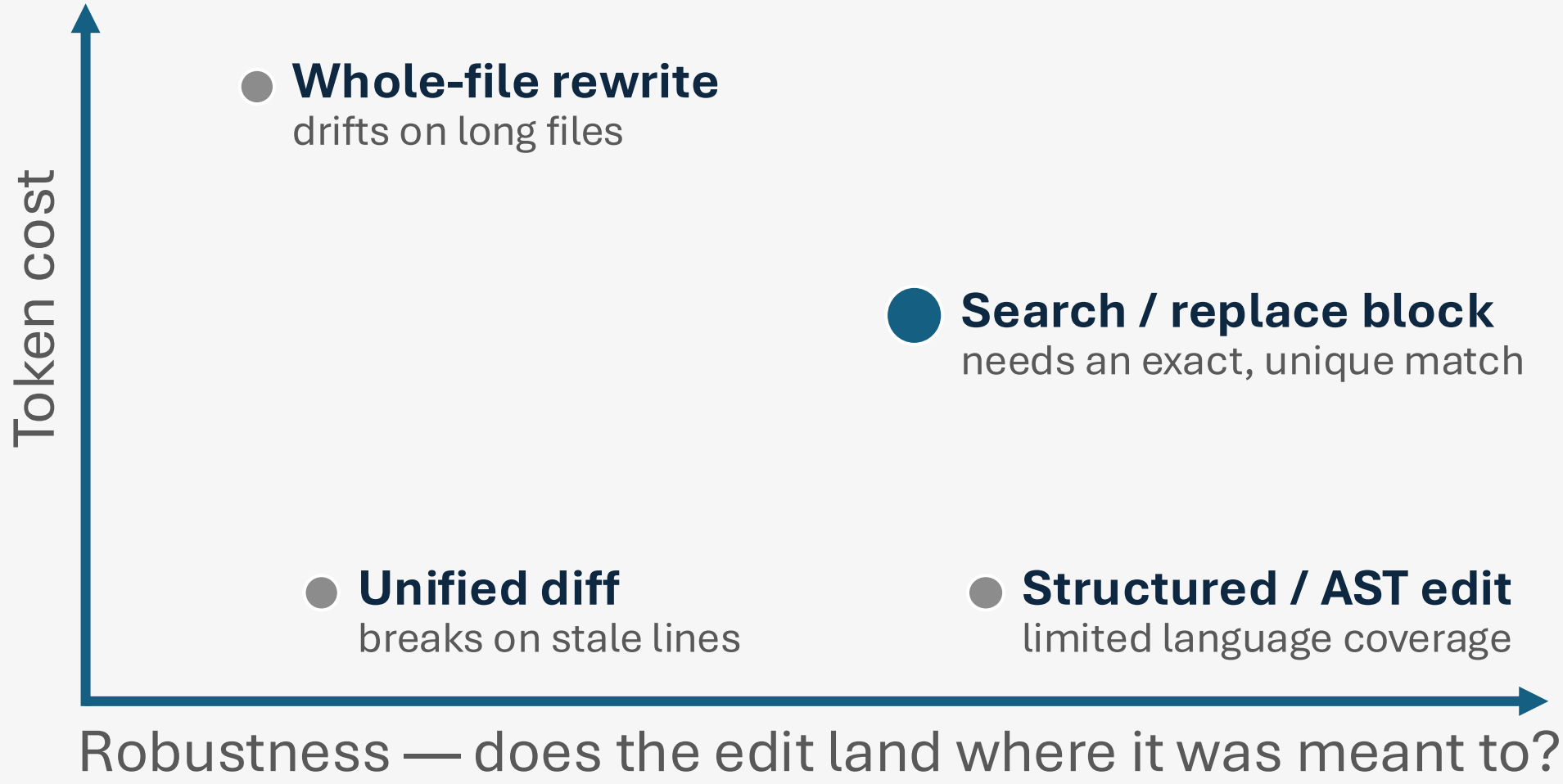
Example — add a timeout parameter to fetchUser()



Abstract syntax trees https://www.youtube.com/watch?v=lCrC-qojPpU
https://pierce.dev/notes/how-language-servers-work

Hands up = score

How agents apply edits



Tool execution: effectively a shell

The most powerful but also the most dangerous tool.

- **Composition** — one call can chain, filter, loop and retry
- **Reach** — filesystem, network, package managers, installed binaries
- **Persistence** — files and processes outlive the individual call
- **Blast radius** — whatever the shell can do, the agent can do

Tool execution: things you have to think about

Isolation — where the code runs, and what it can reach

Failure signal — errors the agent can actually act on

Output — enough but not too many

Duration — long-running and background processes

Budgets — when to stop retrying and hand back

Five more tools that are commonly used

A. Memory and notes — write findings to a file; plans survive compaction

B. Subagents and delegation — exploration returns a shortlist, not 40 file reads

C. Web search and fetch — read the installed source first, then the web

D. Browser and visual feedback — screenshot the render and feed it back

E. Asking the user — one cheap question beats a confident wrong patch

Tool design best practices

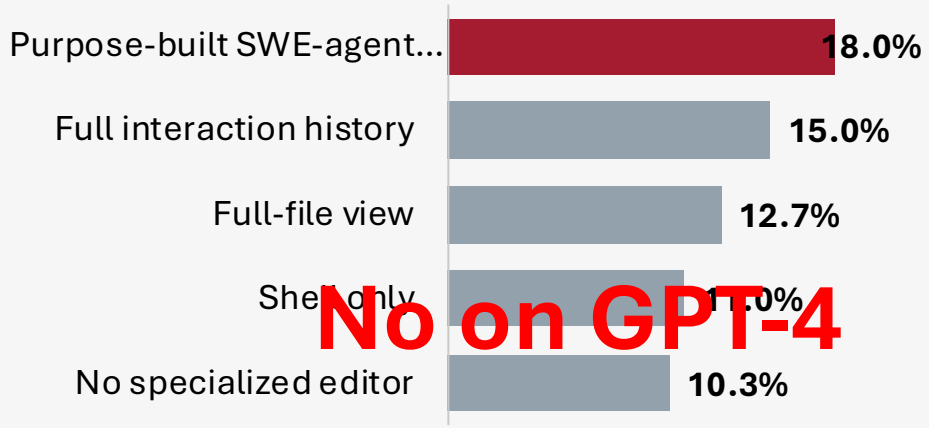
Few, orthogonal tools Overlapping tools make the model dither between them.

Purpose-built, not generic `search_orders(customer_id)` beats a raw `run_sql` tool.

Token-cheap output Truncate, paginate, summarize (tool design is context design).

Errors that teach Return the fix hint, not just “invalid argument”.

Idempotent by default Safe to retry whenever possible.



Can we just provide the shell tool?

Find out when you work on the assignment!

Context Management

Prompt engineering

Context engineering

People love coining terms. I use them so you'll know what they mean when you encounter them, but that doesn't mean I agree or like them.

From Prompts to Agent Systems

2023–2024

Prompt Engineering

"How should we instruct the model?"

Wording, examples, output formats

2025-2026

Context Engineering

"What should the model see?"

Retrieval, memory, history, compaction

2026

Harness Engineering

"How should the agent operate?"

Loop, tools, sandbox, retries, stopping

Now -> Next

Goal Engineering

"What counts as success?"

Objectives, criteria, constraints, tradeoffs

Prompting Engineering

- Zero-shot
- Few-shot
- Role & persona
- Chain-of-thought
- Tree-of-thoughts
- Self-consistency
- Prompt chaining
- Meta prompting

Zero-Shot Prompting: Instruction Only

Instruction only

Use it when

- The task is common and well known
- The format fits in words

Move on when

- The rule is tacit → few-shot

Example

```
You are a support triage assistant.
Classify the ticket as one of:
billing | bug | feature | other
Reply with JSON only: {"label": ...}
If unclear, use "other".

Ticket: "Crashes when I export a PDF."
→ {"label": "bug"}
```

Few-Shot Prompting: Instruction + Examples

Instruction plus a few worked examples.

Use it when

- The output shape is unusual
- The rule is easier shown than said

Move on when

- The task needs steps → chain-of-thought

Example

```
Classify the ticket. Reply with JSON
only.

"Charged twice this month."
→ {"label": "billing"}
"Can you add dark mode?"
→ {"label": "feature"}
"asdfgh"
→ {"label": "other"}

"Crashes when I export a PDF."
→  ?
```

Role & Persona Prompting: Give It a Role

Use a role to guide the behavior.

Use it when

- Tone or judgement matters
- The audience is specific

Move on when

- Needs steps → chain-of-thought

Example

```
You are a senior SRE reviewing a
postmortem for an exec audience.
Flag evidence gaps; never speculate.
Be direct: at most five bullets.

→ "Timeline omits the 04:12 alert
   ack — evidence gap, not a cause."
```

Chain-of-Thought: Make the Steps Explicit

Reasoning steps first, answer last.

Use it when

- The task genuinely has steps
- Errors come from skipped work

Move on when

- Several paths exist → tree-of-thought

Example

```
Work through it in numbered steps,
then end with: ANSWER: <value>

Q: 3 features/week for 4 weeks,
then 5/week for 2 weeks. Total?

1. 3 × 4 = 12
2. 5 × 2 = 10
3. 12 + 10 = 22
ANSWER: 22
```

Tree-of-Thought: Branch, Score, Prune

Branch, score, prune the paths.

Use it when

- Several routes are plausible
- You can score them cheaply

Move on when

- One right answer → self-consistency

Example

```
Propose 3 ways to cut checkout
latency. Score each 1-5 on impact
and effort, then expand the best.

A cache warm-up   impact 3 effort 2
B drop 3rd-party  impact 5 effort 4
C lazy-load asset impact 4 effort 1

Expand C, then B. Prune A.
```

Self-Consistency: Sample, Then Vote

Sample several times, majority voting.

Use it when

- One right answer exists
- A wrong call is expensive

Move on when

- Separable stages → chaining

Example

```
Same prompt, 5 samples, T = 0.7
"Is this invoice a duplicate?"

yes | yes | no | yes | yes
→ majority: yes  (4/5 = 0.8)

Floor 0.8 → answer.
If 3/5, route to a human instead.
```

Prompt Chaining: One Job per Call

One job per call, validated hand-offs.

Use it when

- The work splits into stages
- You can validate each output

Move on when

- Tuning the prompt → meta prompting

Example

```
Stage 1  extract
  in: email text
  out: {vendor, amount, date}

Stage 2  validate + enrich
  out: {..., vendor_id}

Stage 3  draft reply
  out: {subject, body}

Fails schema → retry that stage only
```

Meta Prompting: Let the Model Write the Prompt

The model rewrites its own prompt.

Use it when

- You have an eval set to score on
- The prompt is what you are tuning

Move on when

- Gains stall → revisit your choice

Example

```
Here is my prompt, 10 test cases
and the 3 it fails.
Rewrite the prompt to fix those 3
without breaking the other 7.
Return only the new prompt.

→ v2 adds an explicit refusal rule
   8/10 → 10/10 on the eval set
```

Choosing the Right Technique

Start simple

Zero-shot — the baseline for any common task

Few-shot — unusual output shape or tacit rule

Role & persona — tone or judgement matters

Add reasoning

Chain-of-thought — the task needs several steps of reasoning

Tree-of-thought — several paths worth scoring

Self-consistency — ask multiple times and take the majority vote

Engineer the system

Prompt chaining — separable stages, validated hand-offs

Meta prompting — the prompt is the artifact

Some of the techniques are already **expired** ☹

- Find out yourself in assignment!

Designing Prompts: Four Best Practices

1. **Start simple, then iterate**
   Add context only when you need it

2. **Partition into sections**
   Context, instructions, tools, output spec

3. **Be specific and concise**
   Clever phrasing buys nothing

4. **Split big tasks into subtasks**
   Don’t front-load complexity

EXAMPLE PROMPT

```
### Instruction ###
Translate the text below to
Spanish:
Text: "hello!"
```

MODEL OUTPUT

```
¡Hola!
```

Instruction, delimiter, data — in that order.

⚠ **Prompting advice is model-specific — re-tune whenever you switch**

Context Engineering

Context engineering is ***the delicate art and science of filling the context window with just the right information for the next step***

-- Andrej Karpathy

Karpathy, X (2025) — x.com/karpathy/status/1937902205765607626

Hands up = score

Context is a finite resource

Cost

Every step re-sends the whole window.

→ Every turn pays again

Latency

Bigger prompts take longer to prefill and decode.

→ A slower loop, every turn

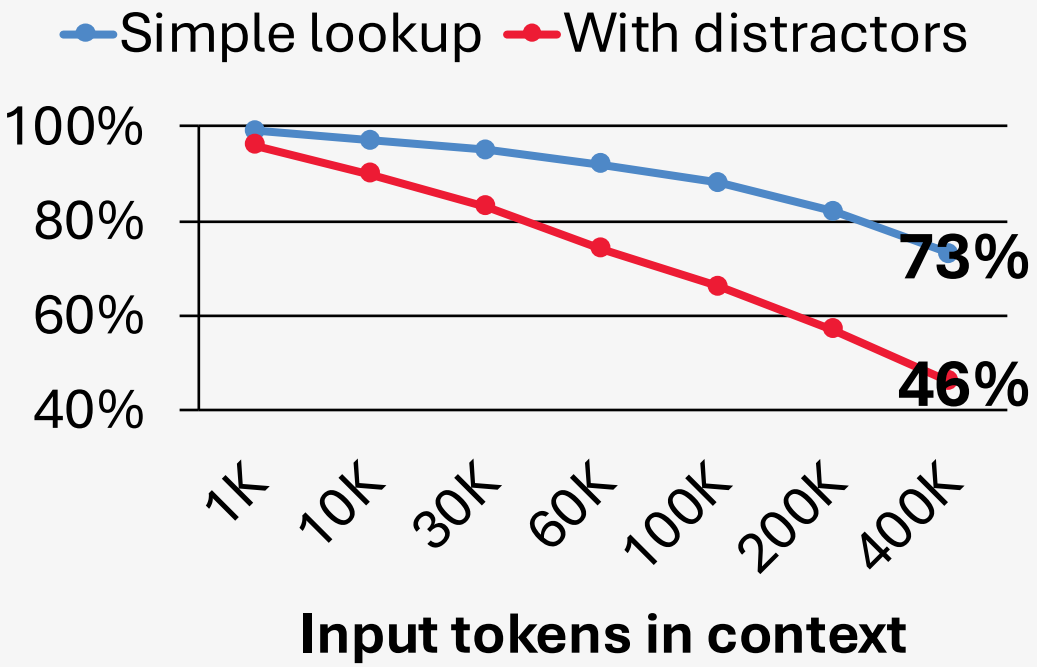
Attention

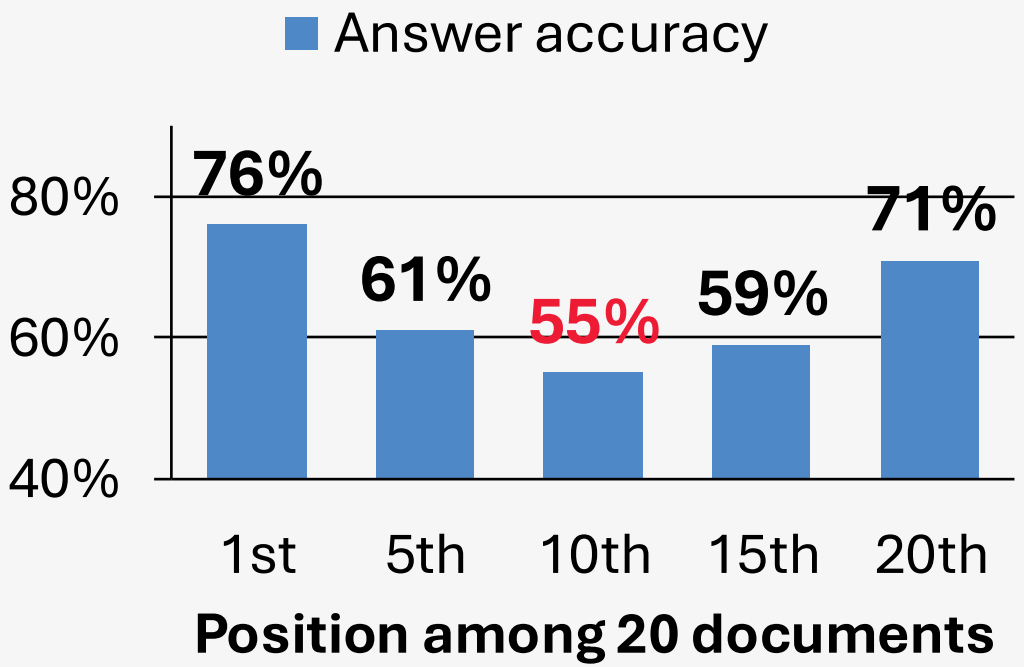
Recall fades as the window fills; the middle gets missed.

→ More context is not better answers

Length and position both cost accuracy

Accuracy decays as input grows, and it also depends on where in the window the answer sits.



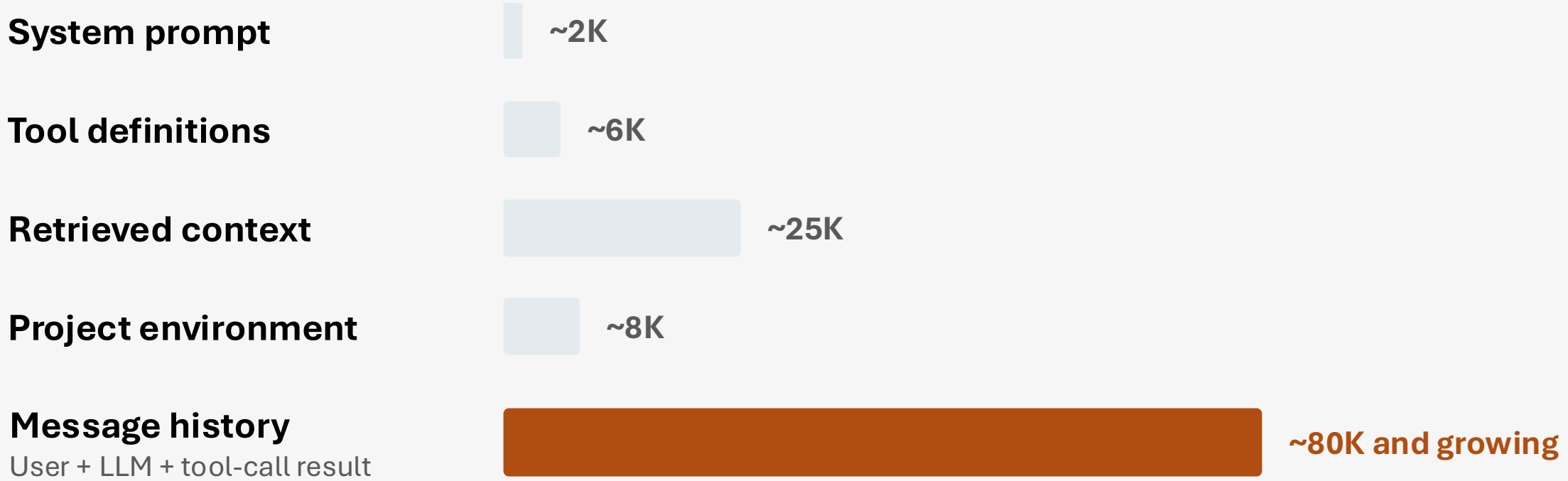


Context Rot (Chroma); Liu et al., “Lost in the Middle,” TACL 12 (2024)

Hands up = score

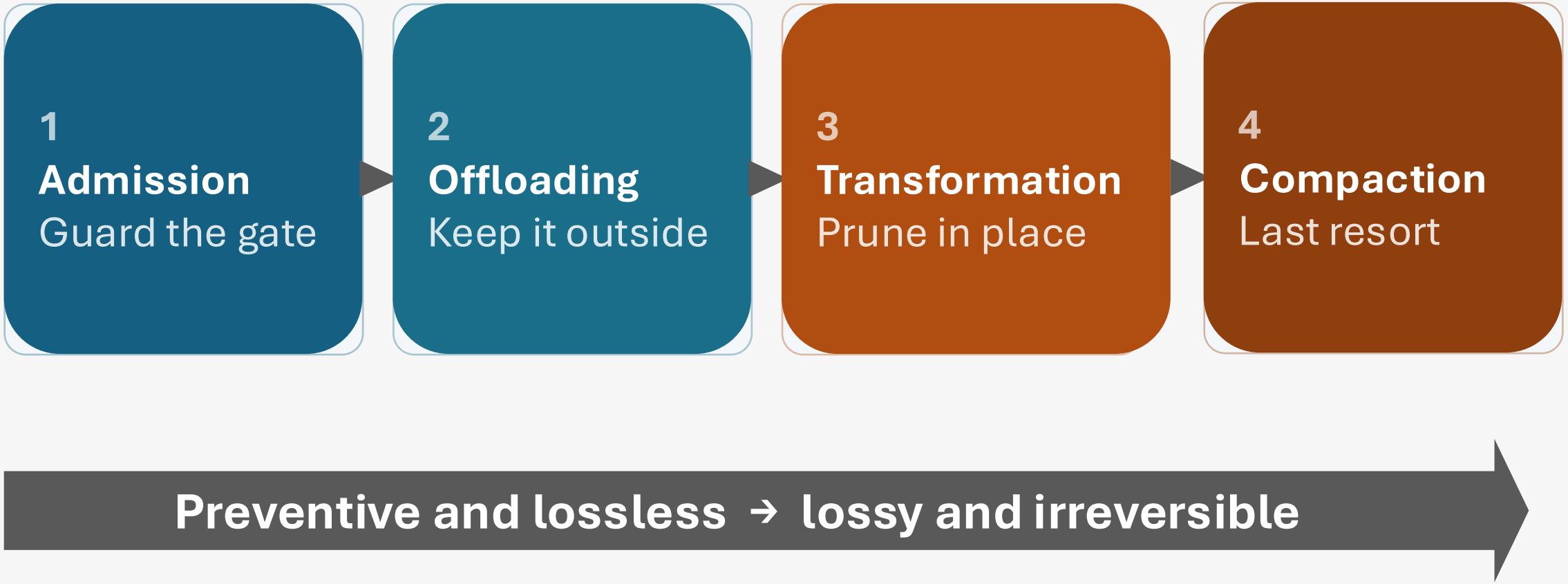
Anatomy of the context window

Illustrative token budget for a single agent turn



What to do when context becomes too large?

Managing the context window



Hands up = score

Admission: guard what enters the context window

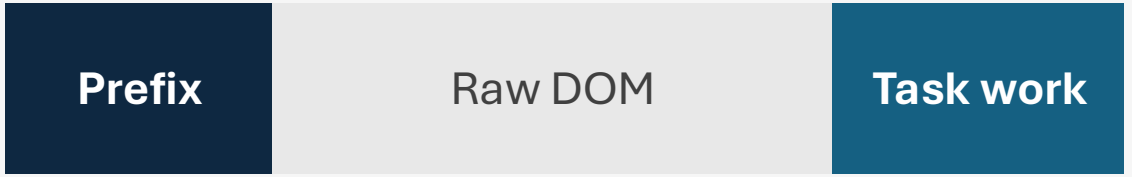
Targeted retrieval
Query narrowly; fetch fields, not docs

Tool output shaping
Drop the DOM, keep the extract

Truncation
Cap size only when shaping cannot

Isolation
Hand the raw payload to a subagent

Before: raw DOM passed through — 48,000 tokens



Shape at the door

After: prices and headings only — 900 tokens

98% smaller, same answer

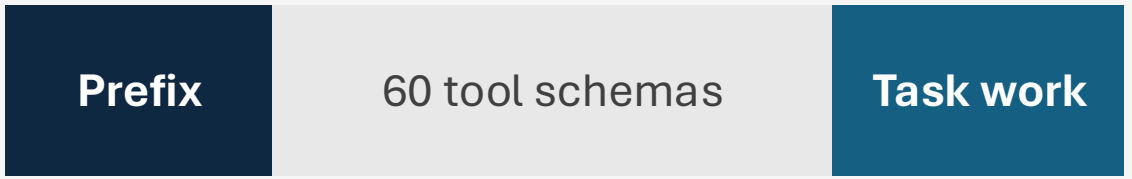
Hands up = score

Offloading: do not load until need

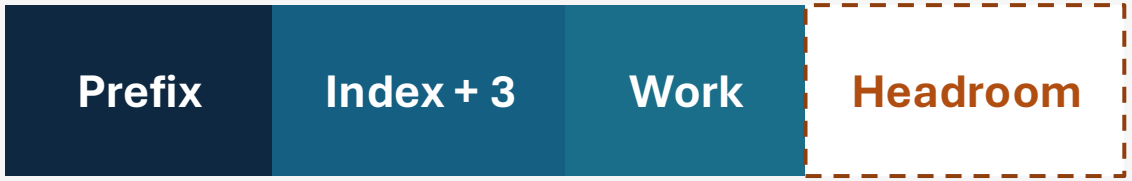
Keep it retrievable, not resident

- **Progressive tool definition**
  Load schemas only when needed
- **Tool search**
  Keep an index; fetch on demand
- **Tool output offload**
  Return a handle, load on demand
- **Externalised task list**
  Keep the plan on disk
- **Persistent memory**
  Write notes out; keep a small index

Before: 60 schemas resident — 42,000 tokens



After: index plus 3 schemas — 3,100 tokens



93% smaller, all 60 reachable

Hands up = score

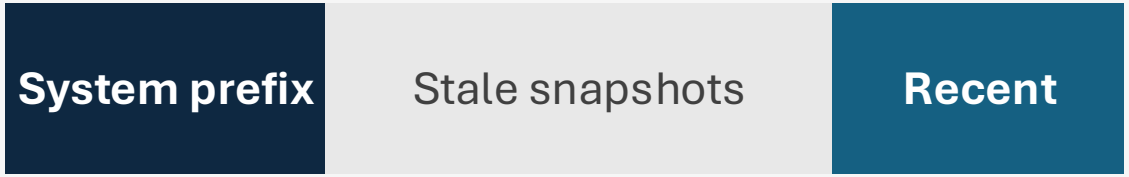
Transformation: selectively drop the history

Deduplication
Keep the newest state of each object

Observation masking
Strip tool results older than N steps

Sliding window
Only keep the recent turns and head

Before: every result kept — 96,000 tokens



Mask & dedup

After: newest state kept — 24,000 tokens

75% smaller, nothing summarized

Compaction: LLM as the last resort

Rolling summary
Summarize old tool results as you go

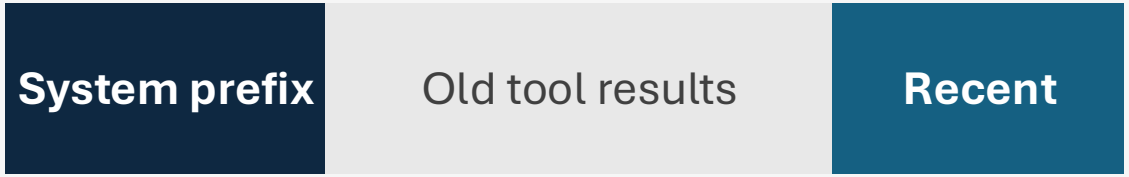
Full summary
Rewrite the whole history at the limit

Auto memory
Persist key facts outside the window

WARNING

- Reserve headroom for output
- Preserve the system prefix
- Re-inject critical facts after

At the limit: history fills the window



LLM compaction

After compaction: headroom restored

Five ways context fails

Name the failure and the fix becomes obvious.

Poisoning	A hallucination enters the window and is later cited as fact
Distraction	History grows until the agent imitates its own past actions
Confusion	Irrelevant tools and documents steer the answer off course
Clash	New information contradicts old; both stay unresolved
Drift	The real goal is buried and swapped for a nearby subgoal

Breunig, “How Contexts Fail and How to Fix Them” (2025)

Agent = Model x Harness

Model capability sets the ceiling;
Harness determines realized performance

Changing the Harness Improves Accuracy by 2×

Benchmark	Basic harness	Improved harness
SWE-bench Verified	43 / 169	**72 / 169**
SWE-bench Pro	31 / 316	**72 / 316**

What changed

- Old tool outputs compacted
- Recent observations kept visible
- Repetition and stalls detected
- Interventions when stuck

Some apparent “model failures” are actually context- and control-loop failures.

Lewis, S., “Same Model, Different Harness: Different Coding-Agent Results” (arXiv:2608.26218, 2026)

Agent = Model x Harness

Experiment yourself in assignment

Agent loop, prompt, tool, context management, failure handling all matter!

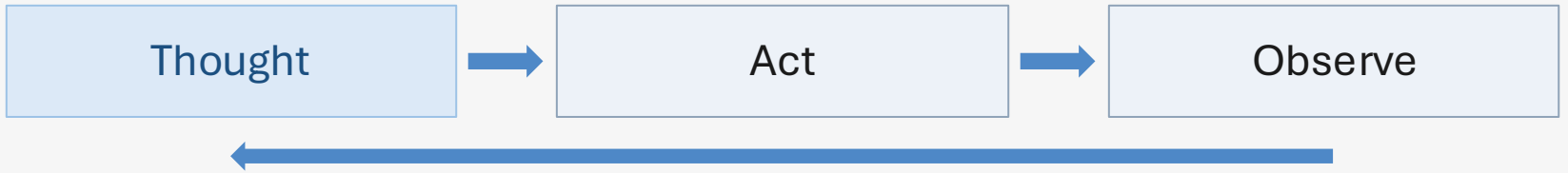
Next class

- Agent design II
  - Agent (persistent) memory
  - Cost-effective agents
  - Safety and security
  - Multi-agent designs

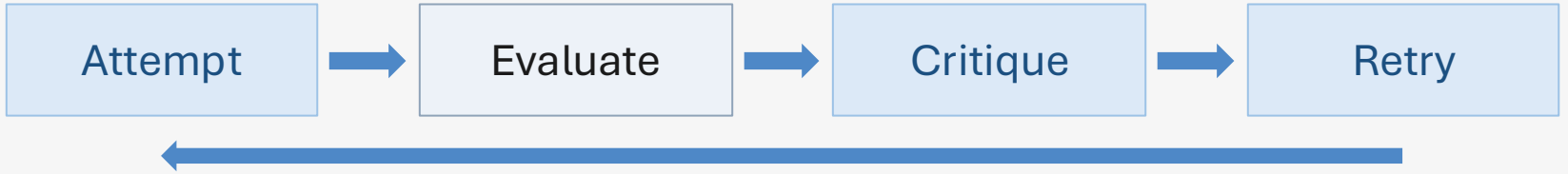
Optional Reading

Beyond ReAct Loop

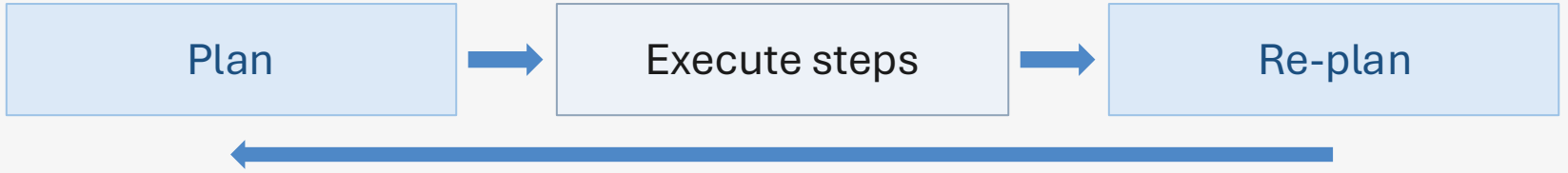
ReAct
loops every step



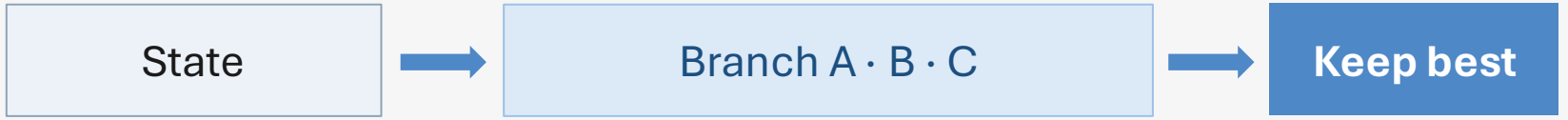
Reflexion
loops after a failure



Plan-and-execute
loops on deviation



Tree search
no loop — one pass, wide



branches explored in parallel — only one survives

model call app / tool action chosen outcome