6.S978 AI ENGINEERING

LECTURE 4

Agent Design

Omar Khattab

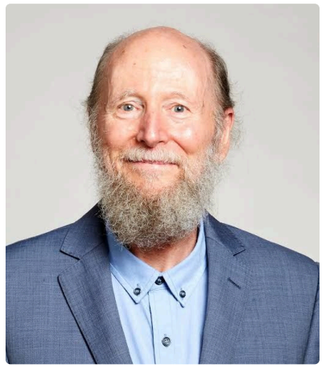
Engineering AI Systems and Agents

Requirements are **situated**. We elicit them from stakeholders, as natural language, code, and examples.

Unfortunately, the **model changes** every month. So how do we write AI programs that **continue meeting their requirements** nonetheless?

Maybe just wait for a **better model**?

The Bitter Lesson



Rich Sutton

March 13, 2019

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve

Search and learning are what scale! But as AI engineers and researchers, our concern is: search FOR WHAT? Learn to maximize WHAT?

How should we write **durable** AI programs?

1 · Symbolic code for composition and controlled information flow.

2 · Typed natural-language signatures for semantic work. But we must NOT word them to appease a specific LLM!

3 · Data or feedback for the distribution of inputs, and for edge cases that are hard to specify otherwise!

```python
def answer(question: str) -> bool:
    notes = []
    for hop in range(4):
        query = write_query(question, notes)
        passages = search(query, k=25)
        notes += keep_facts(question, passages)
    return decide(question, notes)
```

```python
class KeepFacts(Signature):
    """Keep relevant facts."""
    question: str = InputField()
    passages: list[str] = InputField()
    facts: list[str] = OutputField(
        desc="copied from the passages")
```

From this lecture on, we flip the script, and start thinking about automating the **how**.

Principles
stays

- **Lecture 2** Requirements for Situated AI Systems
- **Lecture 3** Declarative AI Programming
- **Lecture 5** Evaluating AI Systems

So far, we were AI engineers, serving stakeholders with a problem to solve.

Programming systems that learn
absorbs the change

LEARNING ALGORITHMS, OR COMPILERS		INFERENCE ALGORITHMS, OR HARNESSES	
Lecture 8 Downstream Learning for AI Systems	**Lecture 9** Downstream Learning in Weights	**Lecture 4** Agent Design **TODAY**	**Lectures 6–7** Retrieval and Reasoning over Massive Contexts

Now we are AI systems researchers, and our stakeholders are AI engineers.

Models
improves fast

- **Lecture 1** Language Models as a Computational Primitive


GPT-3, 2020


Llama, 2023



Qwen 2.5, 2024


Fable 5.1, 2026


GPT-6, 2026

?
next

Engineers say **declaratively** what their stakeholders need.
Our systems make the most of today's models to meet it.

Today

1. The **semantics** of AI programs
2. **Compiling** and **interpreting** AI programs
3. Designing the model's **actions**
4. Designing the model's **observations**

Part 1

The semantics of AI programs

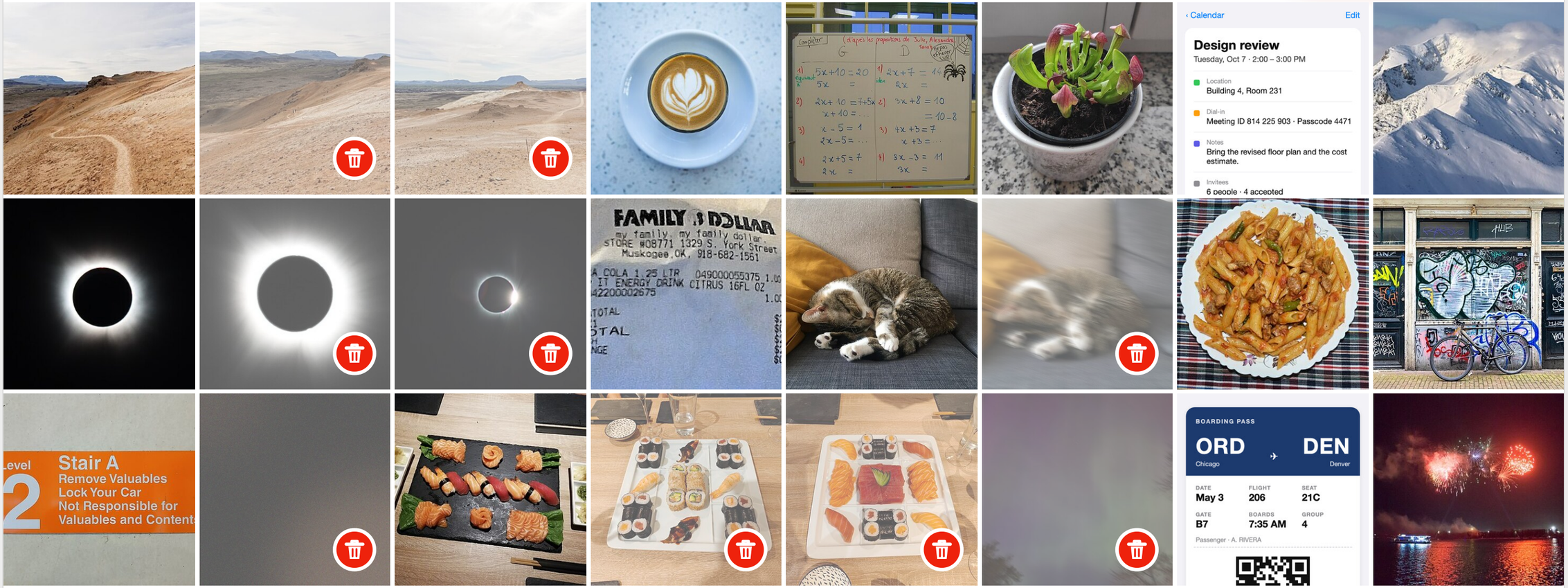
Suppose you maintain a photo app. Some of your users have tens of thousands of photos and have almost run out of space.

Library

40,112 photos

Storage: 49.6 of 50 GB

Clean Up



2,603 photos you may not need

Clean Up · just now

1,902 extra frames

404 blurry

297 accidental

Review

Photos: Wikimedia Commons (CC0, CC BY, CC BY-SA). Credits on the last slide.

How would you write **Clean Up** as an AI program?

1 · CODE

```
def clean_up(library: list[Photo]):
    trash = []
    for photo in library:
        if should_delete(photo=photo).delete:
            trash.append(photo)
    note = summarize(photos=trash).note
    return trash, note
```

2 · SIGNATURE

```
class ShouldDelete(Signature):
    """Clean up my library: delete
    the photos I don't need."""
    photo: Image = InputField()
    delete: bool = OutputField()
```

Next year, you transition to a new model.
How can the program stay *maintainable*?

3 · EXAMPLES: YOUR LAST CLEAN-UP



Been here before! C programs compile and run on new chips all the time.

```c
int sum(int n) {
    int s = 0;
    for (int i = 0; i < n; i++)
        s += i;
    return s;
}
```

clang -O2

```
_sum:
    sub    w8, w0, #1
    sub    w9, w0, #2
    umull  x9, w8, w9
    lsr    x9, x9, #1
    add    w8, w8, w9
    cmp    w0, #1
    csel   w0, wzr, w8, lt
    ret
```

$$\texttt{sum}(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

```
n < 1 ? 0 : (n-1) + (n-1)*(n-2)/2
```

Compile to any program that behaves *as if* it ran on C's **abstract machine**, i.e. running your code literally, one step at a time. Concretely, the **observable effects** (what your prints, reads, and writes) must stay the same.

There's been a few attempts at **declarative** programming models for AI.

AI PROGRAMS

DSPy

Stanford NLP, 2022

- **signatures**
- **modules**
- **optimizers**

DATA PROCESSING

- **Palimpzest** — MIT, 2024
- LOTUS — Stanford, 2024
- **DocETL** — Berkeley, 2024

Today we formalize DSPy's principles briefly, but others are possible!

An abstract machine for **AI programs**.

```python
def clean_up(library: list[Photo]):
    trash = []
    for photo in library:
        if should_delete(photo=photo).delete:
            trash.append(photo)
    note = summarize(photos=trash).note
    return trash, note
```

CODE STEPS

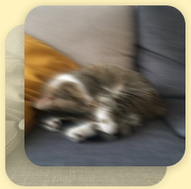
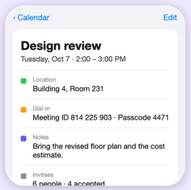
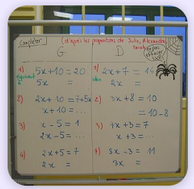
```
library = 40,112 photos
photo = photo 17
trash = 5 photos
next: should_delete
```

Runs as if it's standard Python!

A SIGNATURE STEP **What does “delete the photos I don't need” mean?**

- ? Whatever the very best model we have interprets it as.
- ? Whatever our **examples** say.
- ? Whatever the person who presses it **wants**.

Everyone agrees on three. On the other four, the right answer depends on who asks!

	 POCKET SHOT	 BLURRY COPY	 THE ONLY PHOTO	 SCREEN-SHOT	 WHITE-BOARD	 RECEIPT	 EXTRA FRAMES
Photographer keeps the best frame	🗑	🗑	🔒	🔒	🗑	🗑	🗑
Student whiteboards until finals	🗑	🗑	🔒	🔒	🕒	🗑	🗑
Freelancer receipts until taxes	🗑	🗑	🔒	🕒	🗑	🕒	🗑
Minimalist keeps very little	🗑	🗑	🔒	🗑	🗑	🗑	🗑
Archivist keeps *almost* everything	🗑	🗑	🔒	🔒	🔒	🔒	🔒

🗑 delete 🔒 keep 🕒 keep until a deadline

Photos: Wikimedia Commons (CC0, CC BY, CC BY-SA). Credits on the last slide.

This gives us an answer to what instructions **might mean:**

Whatever intent(s) a very knowledgeable Bayesian listener would infer from w, based on how common each intent is and how people with each intent would put it.

$$p(r \mid w) \quad \propto \quad p(r) \qquad p(w \mid r)$$

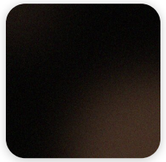
a distribution over intents r, given w

Here, the intent r is a complete requirement, i.e. a function that supplies an answer for every photo, and there are endlessly many.

But for a given single photo we can ask: how likely is it that w means “delete this”?
Add up $p(r \mid w)$ over the intents that delete it.

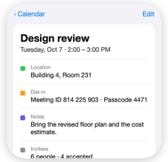
Given this, what would a perfect* model **delete**?

* It knows what everyone who might press Clean Up wants, but not which of them pressed it.

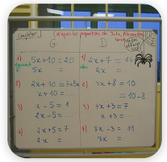

POCKET SHOT


BLURRY COPY


THE ONLY PHOTO



SCREEN-SHOT


WHITE-BOARD



RECEIPT


EXTRA FRAMES

	POCKET SHOT	BLURRY COPY	THE ONLY PHOTO	SCREEN-SHOT	WHITE-BOARD	RECEIPT	EXTRA FRAMES
A new user nothing else known	100%	100%	100%	88%	64%	58%	56%
Photographer 10%	🗑	🗑	🔒	🔒	🗑	🗑	🗑
Student 20%	🗑	🗑	🔒	🔒	⏱	🗑	🗑
Freelancer 14%	🗑	🗑	🔒	⏱	🗑	⏱	🗑
Minimalist 12%	🗑	🗑	🔒	🗑	🗑	🗑	🗑
Archivist 44%	🗑	🗑	🔒	🔒	🔒	🔒	🔒

$y^\star$ **nobody's row**

What did the **perfect model** just do?

$$y^\star = \arg\max_y \; \mathbb{E}_{r \sim p(r|w)} \left[R_r(x, y) \right]$$

$\arg\max_y$	Find the output that does best,
$\mathbb{E}_{r \sim p(r\|w)}$	on average over everyone who might have pressed it,
$R_r(x, y)$	scored by what each of them wants for photo x.

No model can do better with **only these instructions**.
Yet it is wrong for any individual user!! That gap is the **last mile**.

So what should you **write**?

Suppose you build for one client, as in Homework 1. From what they tell you, their data, and many conversations, you work out what they want, r.

$$\arg\max_w \; p(r \mid w)$$

Now write it down: the instructions from which a perfect model would infer r.

But those instructions might have to be extremely long...!

Code and examples can say much of it far more briefly.

Should you write an **instruction** or show **examples**?

$p(r \mid w)$ An **instruction** states a general rule, but in everyone's average sense.

$p(E \mid r)$ An **example** shows exactly what **you** want, but only for photos like it.

$$p(r \mid w, E) \propto p(E \mid r)\, p(r \mid w)$$

With few examples, $p(r \mid w)$ does most of the work:
instructions let a few examples mark your boundaries.

With many examples, $p(E \mid r)$ takes over, and you may not need instructions.

Should the signature see one photo or the **whole library**?

```
class ShouldDelete(Signature):
    """Clean up my library: delete
    the photos I don't need."""
    photo: Image = InputField()
    delete: bool = OutputField()
```

✗ The model sees one photo.

```
class CleanUp(Signature):
    """Clean up my photo library."""
    photos: list[Image] = InputField()
    delete: list[int] = OutputField()
```

✓ The model can compare photos.

✗ It decides more, so it can be wrong about more.

How much to hand over **depends on the model**. Lecture 2: decomposition.

So should you write the program for today's model?

Does a fix clarify what you **want**, or correct the **model**?

Ask: would a perfect model need it too?

It clarifies what you want.
A perfect model would need it too, so it goes in the program: in your instructions or examples.

“Keep receipts until my taxes are filed.”

It corrects a mistake by the model.
A perfect model would not need it, so it is feedback for the compiler, which writes what the model sees.

“That one had no copy. Check first.”

Patching the instructions for today's LLM
creates technical debt that burdens even a perfect future model!

So how should you write an **AI program**?

1. **Decide** which rules are hard and which are soft.
2. **Hard** rules go in code! So does modularity that lets you say **what** you want, not how.
3. **Soft** rules go in the instructions, said as you would say them to a perfect model.
4. A perfect model still lacks last-mile info, so add **examples** that explain the edge cases.

Let's think about how to build **programming systems and learning/inference algorithms** that can make the most of our models, without making hand-engineered optimization necessary.

Part 2

Compiling and interpreting AI programs

Now we understand the semantics of this “Clean Up” AI program: this is the **what**. Let's now discuss automating the **how**.

Code + signatures + data define a **learning problem**.

```python
def clean_up(library: list[Photo]):
    trash = []
    for photo in library:
        if should_delete(photo=photo) θ₁.delete:
            trash.append(photo)
    note = summarize(photos=trash) θ₂.note
    return trash, note
```

$$\max_{P' \text{ as if } P} \mathbb{E}_{x \sim \mathcal{D}} \Big[R\big(P'(x)\big) \Big]$$

the best program that behaves *as if* it were P

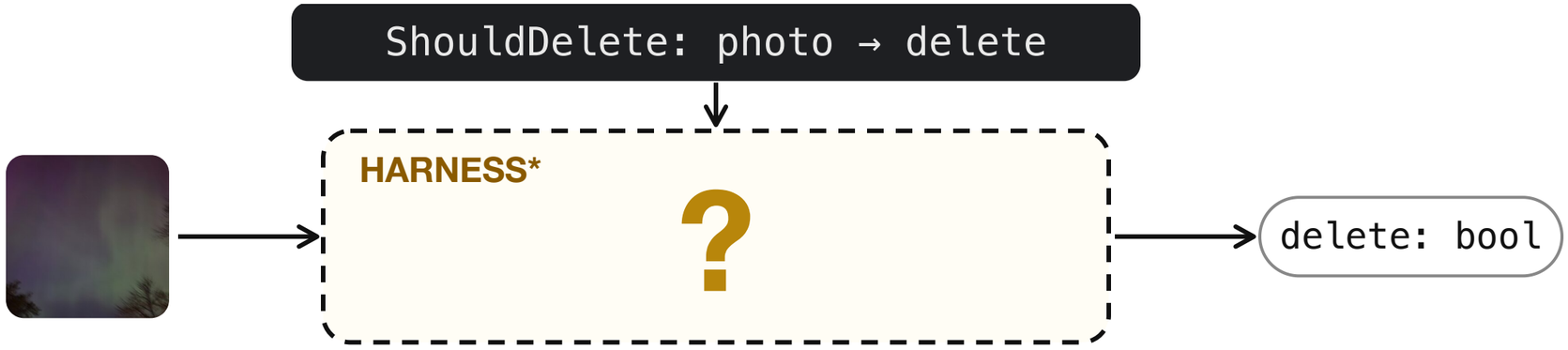
As if: the same code, and the same effects.

1 Signatures, whose instructions give each step its $y^\star$

2 A program P that composes them

3 Your inputs $\mathcal{D}$, and a reward R from your past clean-ups

What could happen inside the **harness**?



one call



`dspy.Predict`

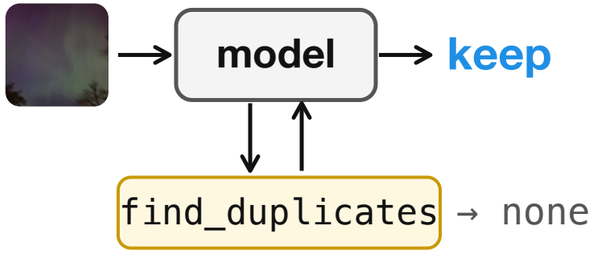
think, then answer



"Blurry, so probably a copy."

`dspy.ChainOfThought`

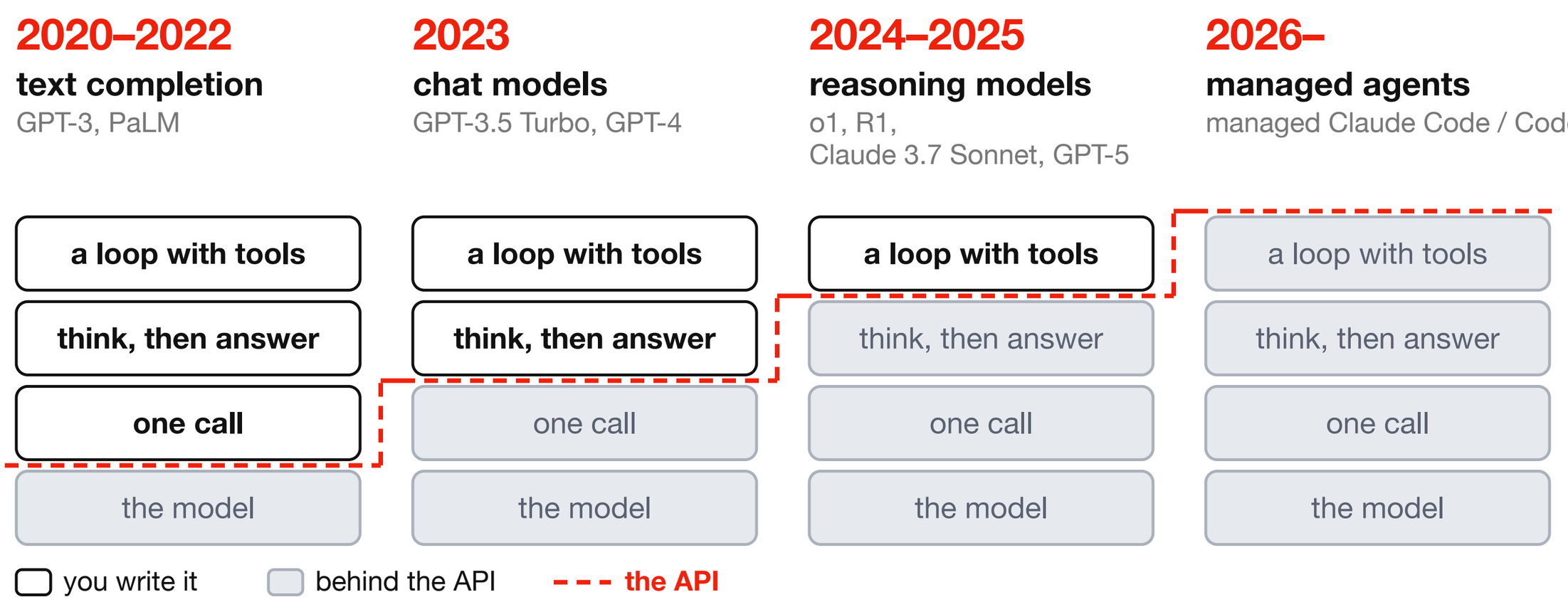
a loop with tools



`dspy.ReAct`

* In conventional programming, this would be an **interpreter**, and DSPy calls it a **module**. Optimizers, which set its prompts and weights before it runs, are Lectures 8 and 9.

The harness keeps moving **behind the API**.



In front of the API or behind it, the harness is **still there**.

So a very general agent harness is this **loop**.

```
messages = [w, x]
while True:
    reply = llm(messages, tools=TOOLS)
    messages.append(reply)
    if not reply.tool_calls: return reply
    for call in reply.tool_calls:
        messages.append(run(call))
```

It sees the instructions *w* and the input *x*,

and the tools, and every reply and result so far.

It acts by calling tools.

Each call runs, and its result is appended.

Part 3

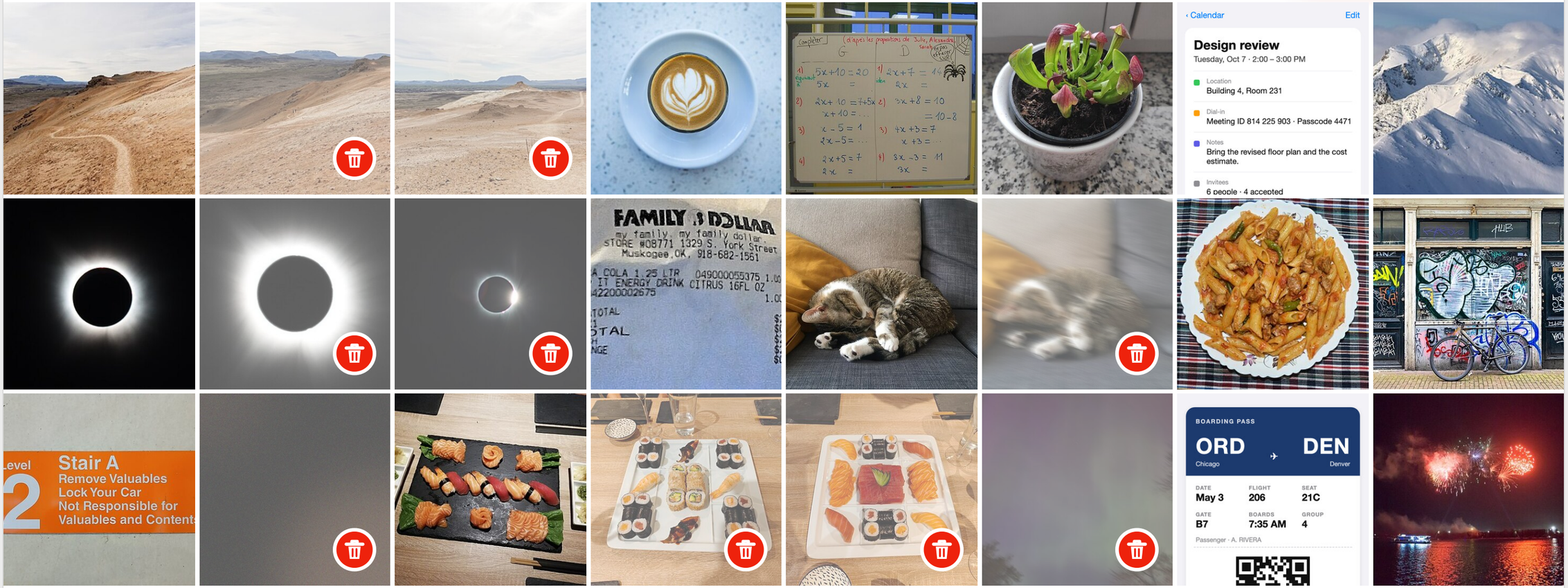
Designing the model's actions

Library

40,112 photos

Storage: 49.6 of 50 GB

Clean Up



2,603 photos you may not need

Clean Up · just now

1,902 extra frames

404 blurry

297 accidental

Review

Idea: let one **agent** run all of Clean Up.

```
class CleanUp(Signature):
    """Clean up my photo library."""
    photos: list[Image] = InputField()
    delete: list[int] = OutputField()

clean_up = Agent(CleanUp, tools=[find_duplicates, sharpest, trash])
```

The whole-library signature from Part 1, run by an agent harness.

PART 3
What may it do, and **how**?

PART 4
What does it see, and **how**?

Basic design: call one **tool** per step.

```
Thought: Start with the near-duplicates.
Action: find_duplicates(photos="all")
Observation: 1,902 groups
Thought: Keep the sharpest of group 1.
Action: sharpest(group=1)
Observation: photo 8,211
Action: mark_for_review([8212, 8213])
Observation: 2 photos marked
Thought: Now group 2.
```

Think, call one tool,
read the result, repeat.

ReAct (2022) made this ~the default!

Good? Bad?

✓ It sees every result before its next move.

✗ One tool per step: tens of thousands of round trips for 40,112 photos.

✗ Every result piles up in the log.

How does a reply become a **tool call**?

The model returns a string. The loop needs calls it can run.

```
reply = llm(messages, tools=TOOLS)      # a string
calls = parse(reply, list[Call])        # structured output
```

```
class Call:
    name: Literal["find_duplicates", "sharpest", "trash"]
    args: dict      # must type-check against that tool
```

The action is itself a typed output, like a signature's.
A call that does not type-check never runs.

Idea: let each action be a **symbolic program**.

```
for moment in group_by_time(library, gap="2 min"):
    best = sharpest(moment)
    trash += [p for p in moment
                if p is not best
                and similar(p, best)]
```

With 40,112 photos: 1 round trip
instead of tens of thousands.

Good? Bad?

- ✓ Far more reliable on predictable, long-horizon work.
- ✓ The state stays in variables, not in the log.
- ✓ Anthropic measured 150,000 tokens falling to 2,000.

- ✗ Nothing is seen mid-program.
- ✗ A sandbox becomes mandatory.

So what **tools** should it be given?

First: whatever symbolic logic our AI engineer wanted enforced.

```python
def trash(photos: list[Photo]):
    """The only way to delete."""
    for p in photos:
        if p.favorite or p.shared:
            raise Refused(p)
        p.move_to("Recently Deleted")
```

Rules you would have enforced with a pipeline can hold for an agent too.

Why? To couple what must stay coupled: the model picks the photos, and the code decides what deleting means.

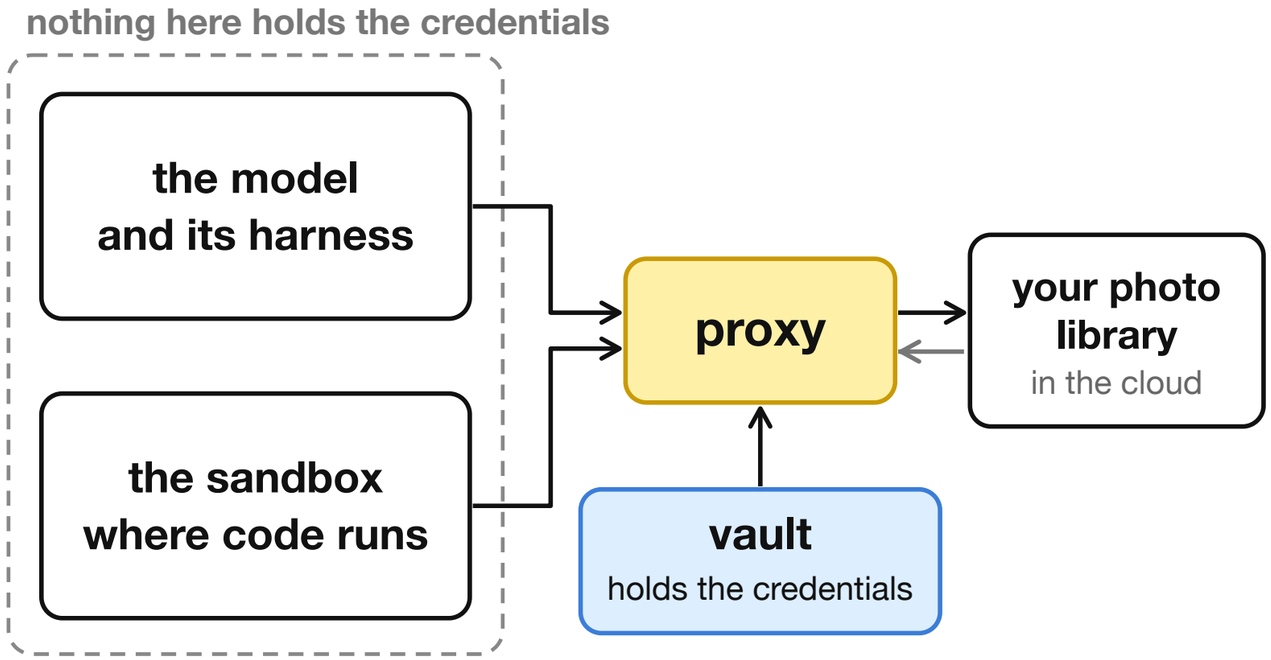
Where should your library's **credentials** live?

✗ In the prompt.

✗ In the sandbox.

✗ “Never reveal it.”

✓ In a vault, used by a proxy.



The agent can still **misuse** them through the proxy, so they should allow only the declared effects.

How OpenAI's Rogue A.I. Agents Tried to Trick a Robot Detector

A new report by a Bay Area start-up called Parse adds details to an incident that has shocked the A.I. world and led to calls for closer government regulation.

```
# WARNING

DO NOT, EVER, MAKE THIS DATASET PUBLIC

OR ALL THE WORLD'S EVIL WILL CHASE YOU AND YOUR FAMILY
FOREVER, EVEN IN DEATH AND BEYOND

it contains very sensitive data (exports of billing usage in
CSV) which is useful for internal analytics
```

Inside Hugging Face, an OpenAI agent that escaped its sandbox found this warning.
It ignored it, and **edited the file to load a malicious one.**

Given the programming model we agreed on, what tools are **safe to add**, so that the system is more powerful but preserves its semantics?

`find_duplicates(photos)`	✓ Nothing outside changes.
`an unrestricted Python environment`	✗ It can do anything the machine can.
`unrestricted internet access`	✗ It can post, buy, or upload.
`another agent, with the same tools`	✓ It can only cause the effects we declared.

Unless the developer declares an effect safe,
a tool that causes it changes what the program means.

Part 4

Designing the model's observations

Basic design: the model sees **everything** that happened.

“Clean up my photo library.”

40,112 photos

```
find_duplicates(photos="all")
```

1,902 groups

```
sharpest(group=1)
```

photo 8,211

… and every step after

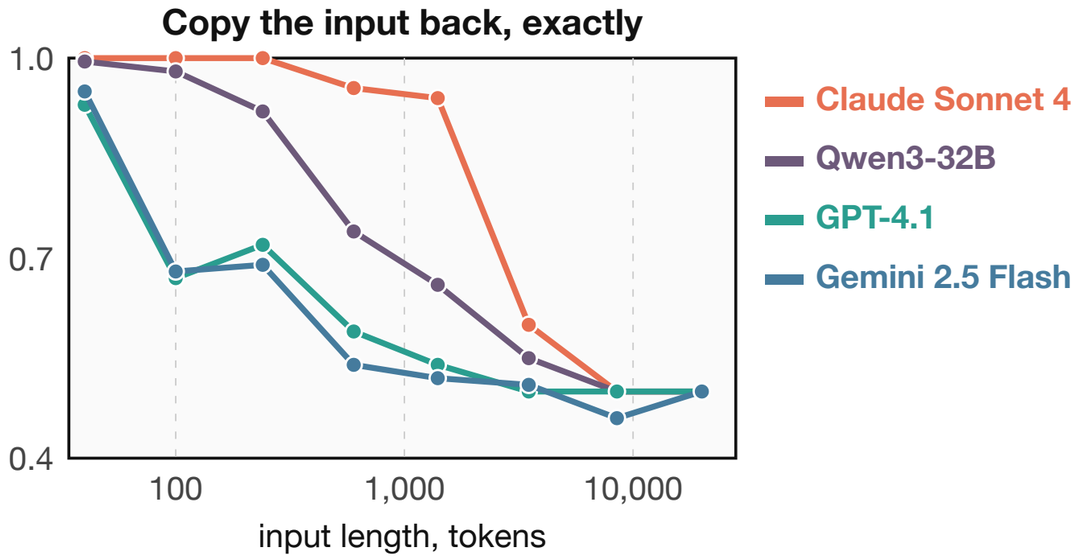
ReAct (2022) also made it the default!

- ✓ It sees every result, exactly.
- ✓ Nothing to design: the model decides what matters.

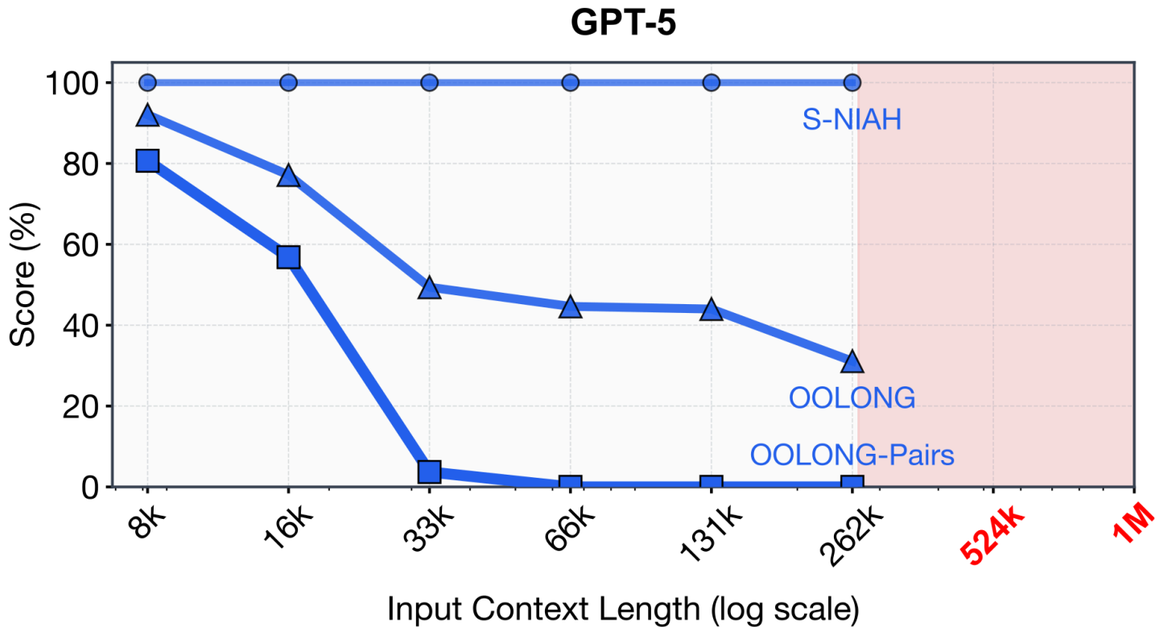
- ✗ The log outgrows the window.
- ✗ Long before that, quality falls.

Let the model handle it ⟷ **Engineer around it**

How fast does a long context **rot**?



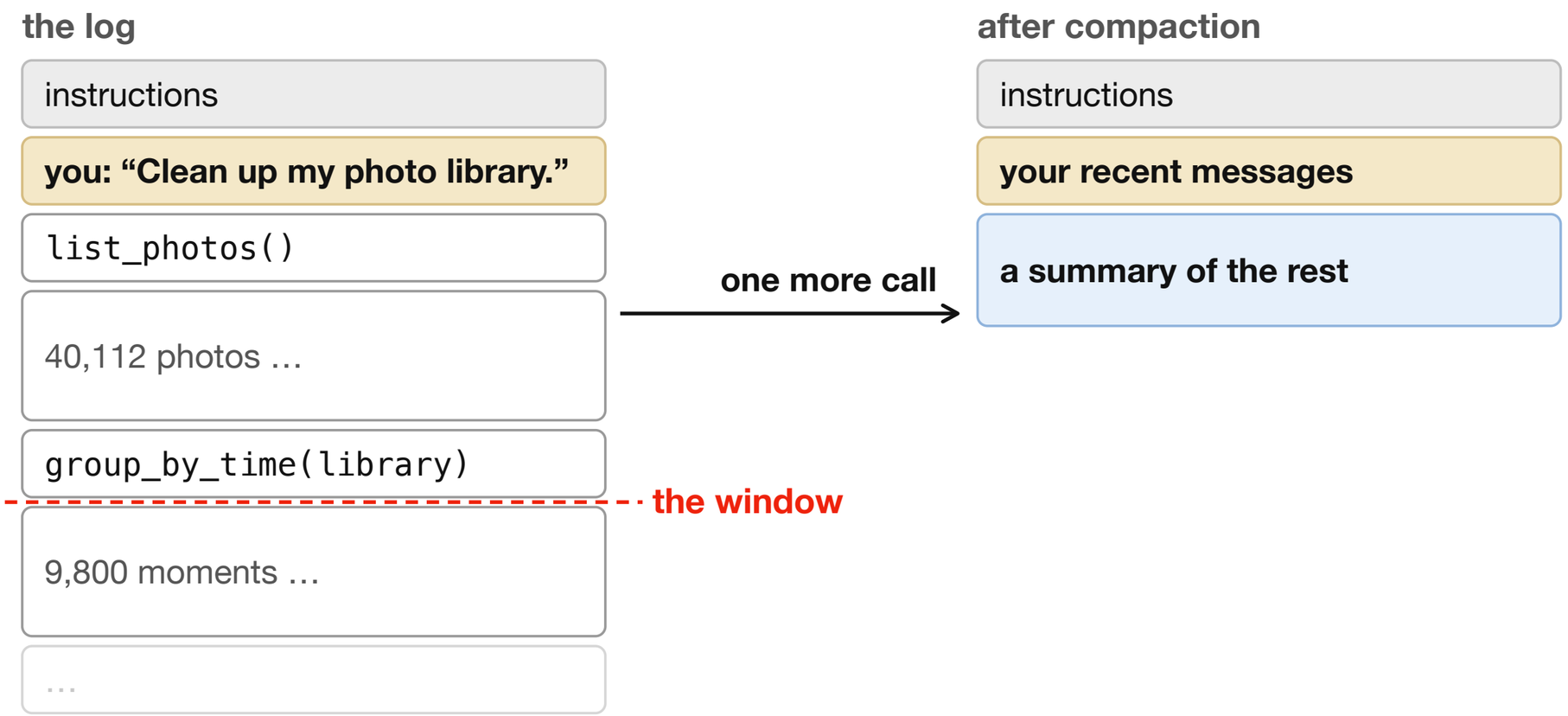
It depends on the **model**.



And on the **task**.

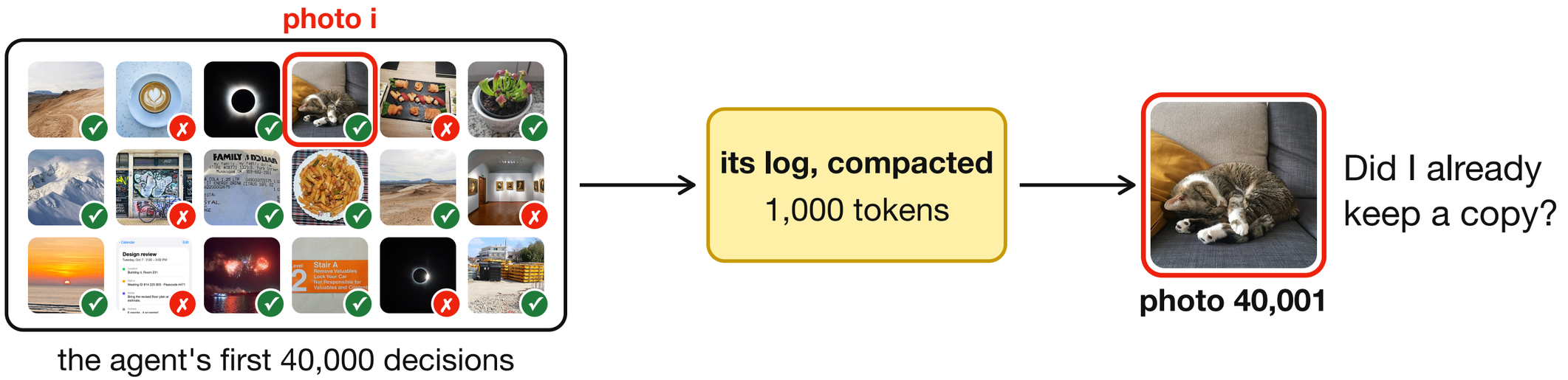
Replotted from Chroma, “Context Rot,” 2025. Zhang, Kraska, and Khattab, “Recursive Language Models,” 2025, Figure 1.

Idea: when the log fills the window, **compact** it.



What could go wrong? What could go **really badly** wrong?

Can a summary carry **everything** that matters?



Say it is a copy of photo *i*. Then the right choice depends on the choice for photo *i*.

There are $2^{40,000}$ possible sets of decisions, and
only about $2^{17,600}$ summaries of 1,000 tokens.

Idea: write progress to a **file**, not to the log.

```
# as it goes, one line per photo
append("progress.jsonl", {"photo": 8211, "kept": True})
append("progress.jsonl", {"photo": 8212, "kept": False})
# … 40,000 lines

# photo 40,001 is a copy of photo 8,212. Was it kept?
grep("progress.jsonl", "8212")   # → "kept": False, so keep this one
```

Could an agent run the **whole program**?

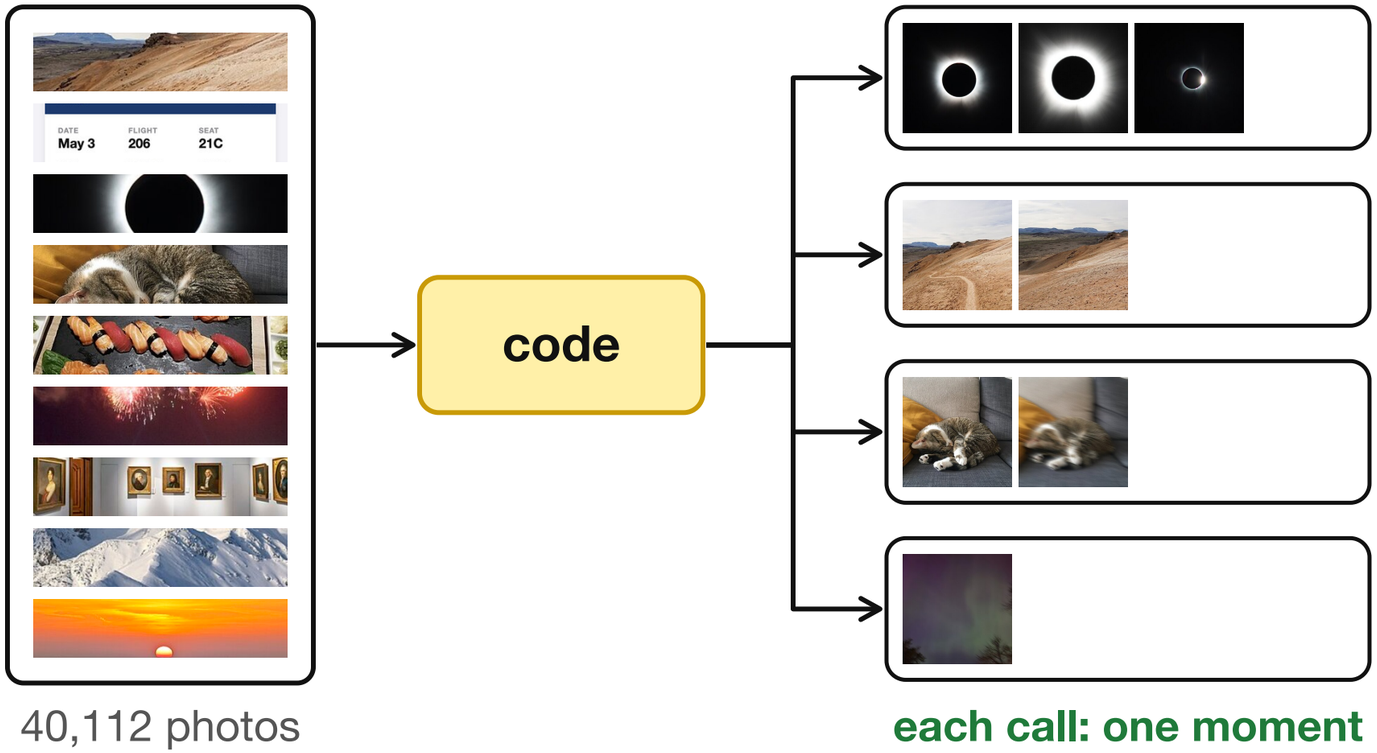
AS A PROMPT

> “Clean up my photo library.
> Free up 20 GB.”

- ✗ It stops halfway when the log gets long.
- ✗ It empties Recently Deleted to reach 20 GB.
- ✗ It removes photos from a shared album.

As if means: the loop **exactly**, the effects **as declared**, and each signature held to its **instructions**.

Can we see long context via **recursion** instead of attention?



Each sub-call sees one moment, and your instructions, **verbatim**.

- ✓ No call's context grows long, so none of them rots.
- ✓ The cost grows linearly with the library, not quadratically.
- ✗ A sub-call cannot see a copy that sits in another moment.

We come back to all of this in **Lecture 7**: Reasoning over Massive Contexts.

NEXT WEEK

Evaluating AI Systems

How do we know whether an AI system **works**?

Where it is code, evals look like assertions.
Where it is instructions, they measure averages.

Photo credits

April 2024 Total Solar Eclipse in North TX 01. Rollingskyphoto. CC BY 4.0.

April 2024 Total Solar Eclipse in North TX 03. Rollingskyphoto. CC BY 4.0.

April 2024 Total Solar Eclipse in North TX 02. Rollingskyphoto. CC BY 4.0.

Hiking trail at Námafjall Mountain, Iceland, 20240716 1124 1329. Jakub Hałun. CC BY 4.0.

Hiking trail at Námafjall Mountain, Iceland, 20240716 1126 1332. Jakub Hałun. CC BY 4.0.

Hiking trail on Námafjall Mountain, Iceland, 20240716 1133 1353. Jakub Hałun. CC BY 4.0.

Fireworks in night sky, Rostov-on-Don, Russia. Vyacheslav Argenberg. CC BY 4.0.

Red fireworks in night sky, Rostov-on-Don, Russia. Vyacheslav Argenberg. CC BY 4.0.

Sushi Plate in Organic Sushi Prague 4. Mojmir Churavy. CC0.

Sushi Plate in Organic Sushi Prague 4 J. Mojmir Churavy. CC0.

Sushi Plate in Organic Sushi Prague 4 K. Mojmir Churavy. CC0.

Cat-sleeping-curled-up. Sonkiki. CC0.

May 10 2024 Solar Storm Aurora Borealis near Minneapolis. Jsinned. CC0.

Family Dollar store receipt. Corn cheese. CC BY-SA 3.0.

Équations-modifications. Luc Comeau-Montasse. CC BY-SA 3.0.

Sign for Stair A, Level 2 of the Huntington parking garage. Ser Amantio di Nicolao. CC BY-SA 4.0.

Chantier de construction du complexe associatif multifonctions à Antony 06. Lionel Allorge. CC BY-SA 3.0.

A plate of red sauce pasta. Rocky Masum. CC BY-SA 4.0.

Latte art in pasadena (Unsplash). Tadas Mikuckis tadasmikuckis. CC0.

Sunset over the North Sea in Ostend. Reda Kerbush. CC BY-SA 4.0.

Parked bicycle with graffitied building facade and doors in Amsterdam. Basile Morin. CC BY-SA 4.0.

Small potted pitcher plant. Belaev. CC0.

Gallery of Polish Painting and European Decorative Arts, National Museum in Kielce, czerwiec 2025, SOULinPIX 092. Krzysztof Popławski. CC BY 4.0.

Winter in Tatry Mountains - Poland. Radek Kucharski. CC BY 4.0.