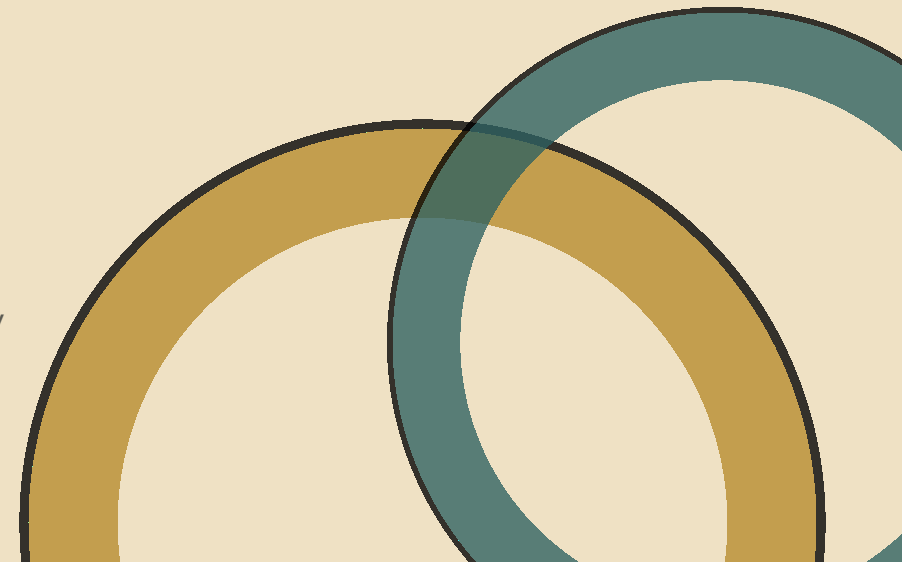


The engine's failure modes, and the loop

Function calling · six ways the engine fails ·
compounding error · the loop, its four parts
and its exits

Slides CC BY 4.0 · from the teaching kit of *AI Agents, Engineered* by
Enrique Gutiérrez · aiagentsengineered.com/teach/



Where Chapter 2 hands over to Chapter 3

“Part II takes this engine—
brilliant, forgetful,
overconfident, dice-driven—and
puts it in a loop.”

Chapter 2, closing paragraph

By the end of today you can

Four objectives

1. Describe the function-calling exchange and point to **the one step where side effects happen**
2. List the six engine failure modes and name the engineering remedy for each
3. Derive compounding error, p^n , and its **three levers**
4. Implement the four-part minimal agent with a scripted model client and a mandatory budget

Part 1 · Chapter 2, sections 4-6 (free)

From text to action

Prompting fundamentals, in one slide

In an agent, no one types the final prompt

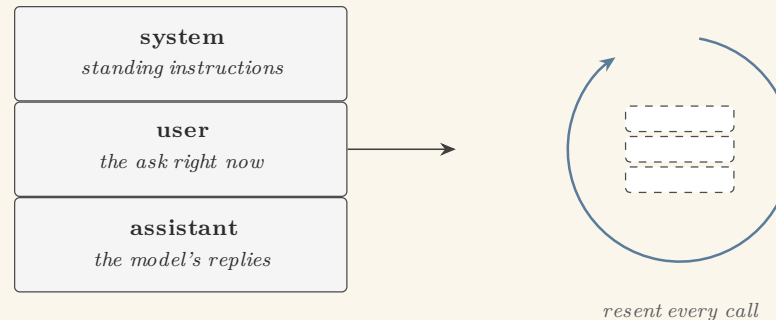


Figure from Chapter 2

- Enforceable constraints beat adjectives: “three to five sentences”, not “keep it short”
- Few-shot examples fix the **shape**, not the reasoning
- “an instruction is a request, not a guarantee”

Sketch the parser before the next slide

Three runs, three correct answers

Name: Ada, Total: \$42

The customer is Ada Lovelace and she spent forty-two dollars.

Happy to help! Ada's order total is \$42.00 (invoice #1041).

- Every line is right as prose; no single parser handles all three
- Tomorrow brings a fourth phrasing: “A program cannot consume possibilities.”
- It needs a **contract**: JSON, with its shape pinned by a **schema**

Chapter 2, “Structured Output and Function Calling”

Two grades of guarantee that sound alike

Asking vs constrained decoding

Weaker • “JSON mode”

It parses

A field can still be missing, invented, or a string where the number goes

Stronger • constrained decoding

It fits the schema

The schema becomes a filter that strikes invalid tokens before every draw

“A schema, misused, makes fabrication easier to consume, not rarer.”

Chapter 2 • a guarantee about shape says nothing about truth

Function calling • where does anything happen?

The model proposes; your code disposes

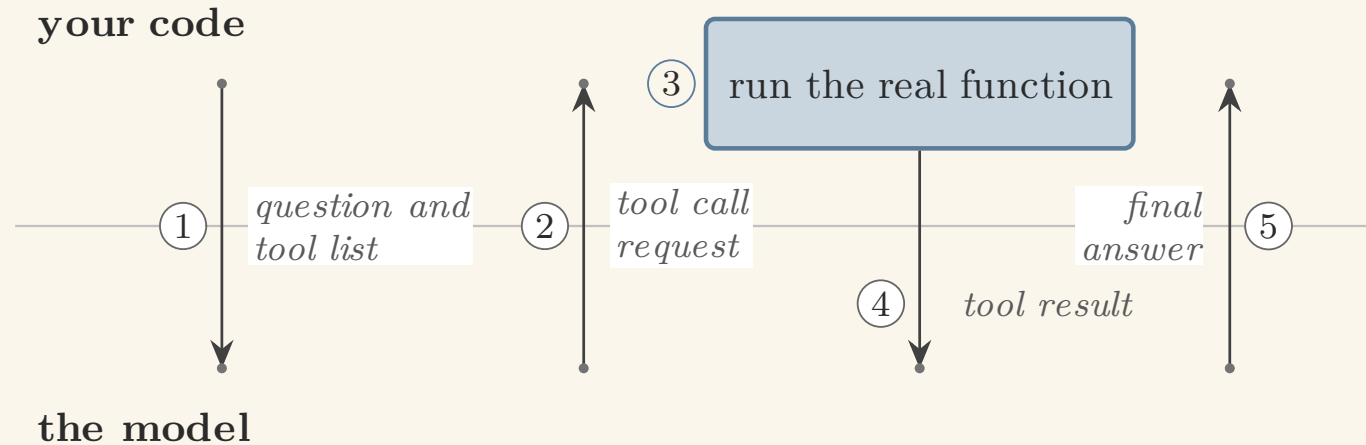


Figure from Chapter 2 • schematic; wire formats vary by provider

- “Every side effect in the system happens at step 3, inside ordinary code you wrote”
- Two model calls at minimum: the result goes back to the **model**, not to the user

All the model will ever know about a tool

Name, description, argument schema

```
name:          create_ticket
description:    Open a support ticket.
                Use when the user reports
                a problem that needs human
                follow-up.
arguments:     { title:      string,
                severity: one of
                    "low" | "med" | "high",
                body:      string }
```

- Misused tool? Suspect the **description** before the model
- Expect zero, one or many calls per turn
- Bad call: return the error **as a result**, cap the retries
- “A flawlessly schema-conformant call to `delete_account` still deletes the account.”

Chapter 2 • pseudocode, as in the book; each vendor’s syntax differs

Hallucination: fluent, confident, wrong

- Lacking the fact, the machinery writes the most plausible string where the fact should be
- “Tone carries no information about truth”
- **Snowballing**: once a false claim is in the transcript, later text elaborates on it

“language models are optimized to be good test-takers, and guessing when uncertain improves test performance.”

Kalai et al., “Why Language Models Hallucinate” (2025),
quoted in Chapter 2

-19 points

accuracy for consultants *with* model help, on one task chosen to sit just outside the model's competence

Dell'Acqua et al., HBS Working Paper 24-013 (2023), 758 consultants, preregistered; as reported in Chapter 2

- Inside the frontier, the same group finished more tasks, faster, at higher rated quality
- The failing task did not *look* harder
- Competence next door proves nothing about competence here
- Remedy: “test your model on your task”

Respond with engineering, not sterner prompts

Six failure modes, six remedies

Failure mode	What it looks like	Engineering remedy
Hallucination	Confident invented facts	Ground in sources; verify outside the model
Jagged frontier	Fails an easy-looking task	Test your model on your task
Phrasing sensitivity	Reworded prompt, new answer	Version and re-test prompts like code
Sycophancy	Caves when you push back	A real verifier, not “are you sure?”
Exact computation	Arithmetic and counting wobble	Route exact work to code or a calculator
Instructions = data	Fetches text steers it: <i>prompt injection</i>	Defend in the harness (Ch. 17)

Chapter 2, “Limitations and Failure Modes” • properties of current models; verify where yours stand

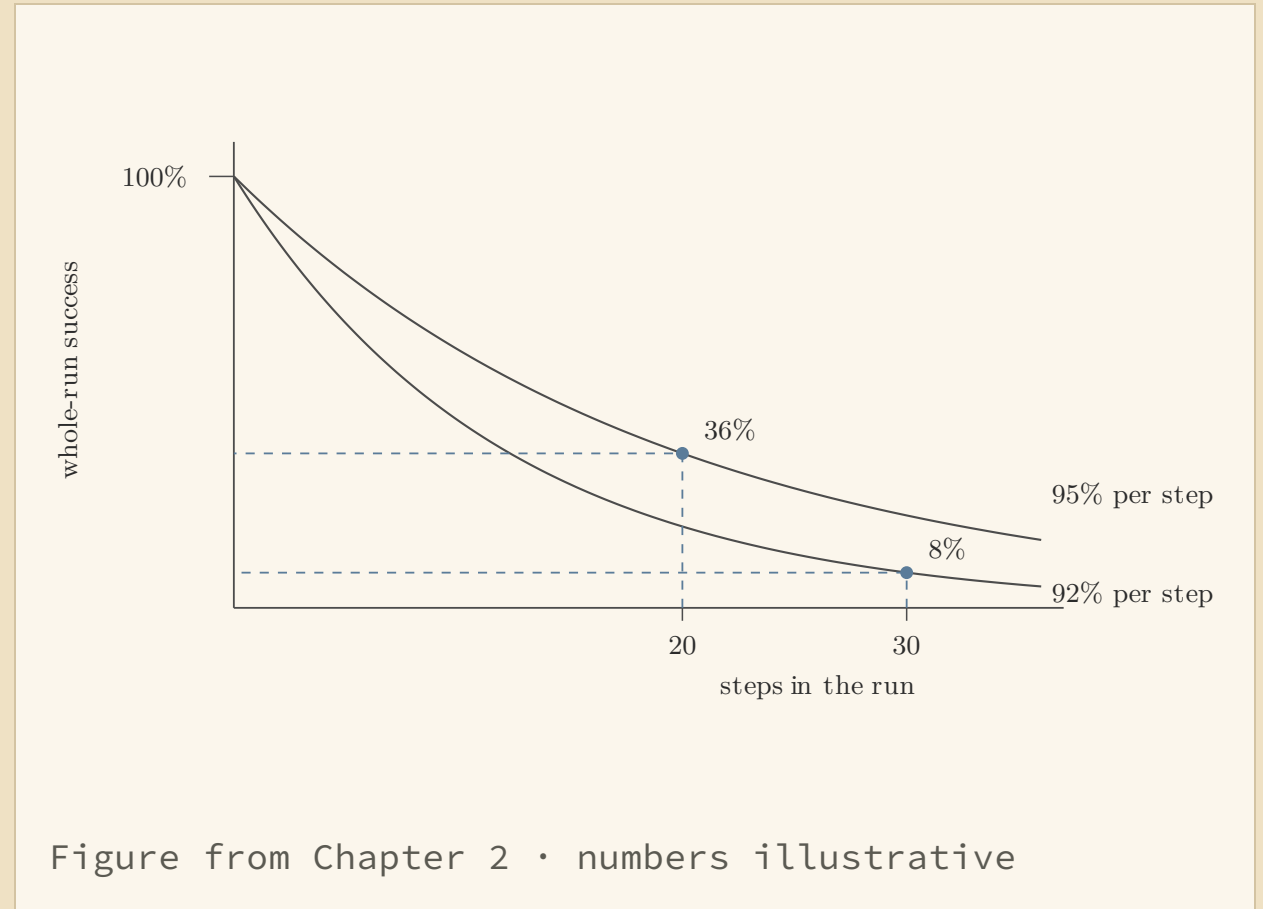
The master entry, now defended

Reliability multiplies: p^n

$$0.95^{20} \approx 36\%$$

$$0.92^{30} \approx 8\%$$

- Independence is the **optimistic** case
- Errors land in the transcript and condition every later step



Three levers

1 · Shrink n

Fewer dice

Fuse steps, precompute,
push bulk work into
code

$$0.95^{10} \approx 60\%$$

2 · Raise effective p

Verify and recover

An error costs a retry,
not the run

$$(1 - 0.05^2)^{20} \approx 95\%$$

3 · Cut the price of dying

Checkpoint

A failed run resumes
from the last good state,
not from zero

same odds, cheaper
failure

Levers from Chapter 2 · the worked numbers are illustrative algebra from a $95\% \times 20$ -step baseline (36%), assuming a check that always catches a failed step

In class • pairs • 15 minutes

How reliable must each step be?

1. Watch: [Why errors compound](#) (3 min explainer)
2. Open aiagentsengineered.com/tools/compounding-error-calculator/
3. Target 90% end to end over 20 steps: find p^* , the per-step reliability you need
4. At 95% per step, what is the longest chain that still meets 90%?
5. Pick the **cheapest lever** for a task of your choice, and defend it in two sentences

Runs in the browser • no account, no API key • copy the result as Markdown to submit

Part 2 • Chapter 3 and Appendix A (full book)

The loop

Observe · reason · act · observe the result

Four beats to a pass

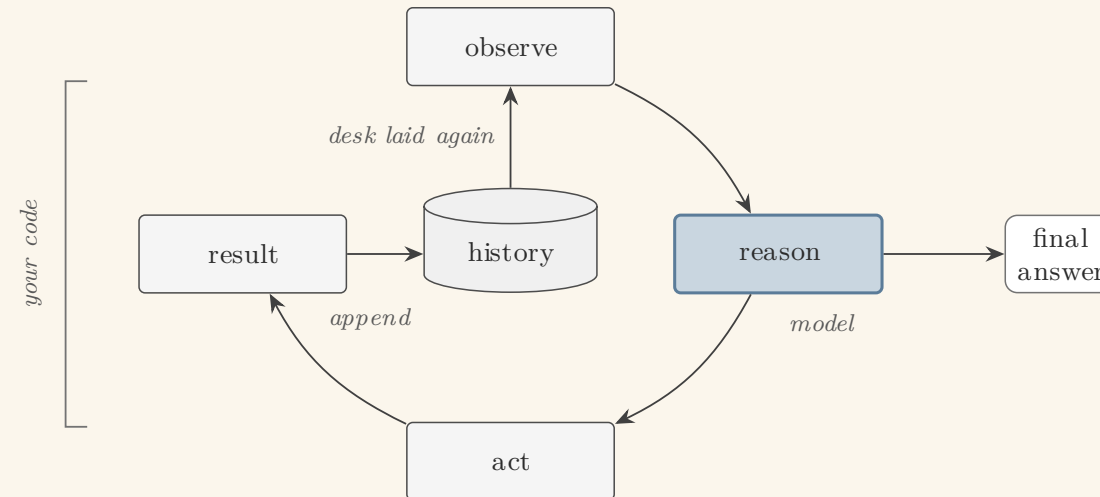


Figure from Chapter 3 · explainer: [the agent loop in three minutes](#)

- Agents “are typically just LLMs using tools based on environmental feedback in a loop” (Schluntz and Zhang, quoted in Ch. 3)
- The model supplies **one** beat; your code supplies the other three

The index card • Chapter 3, “The Loop”

```
history = [standing_instructions, user_goal]

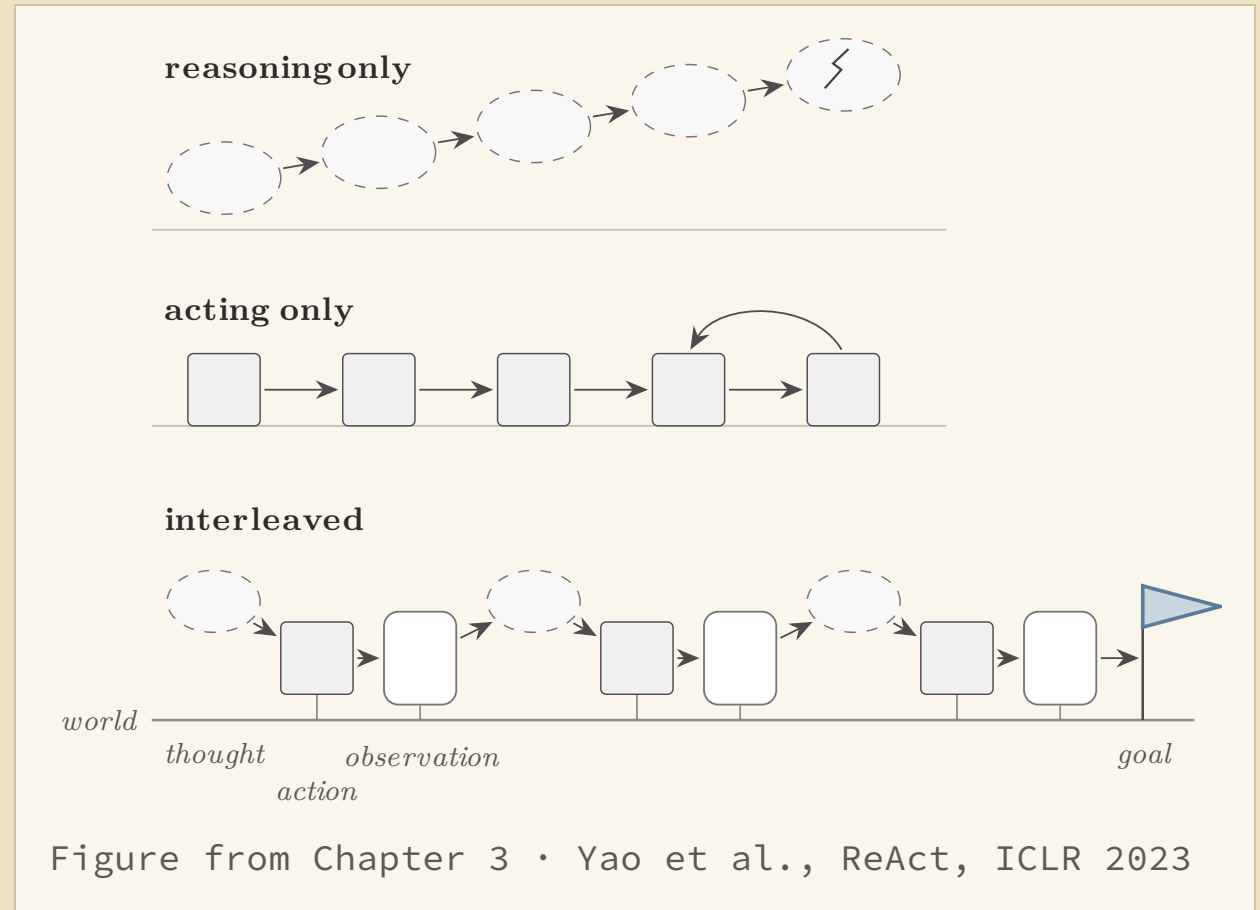
repeat up to MAX_STEPS times:
    response = model(history)           # reason over the desk
    if response is a final answer:
        return response.text           # stop: the model judges it done
    result = execute(response.tool_call) # act: your code runs the tool
    append response.tool_call to history # record what the model did
    append result to history            # record what the world said back

return "stopped: step budget exhausted" # stop: the safety net
```

- “you wrote the loop; the model writes the path”
- No branch mentions tests, files or checkouts
- The two **appends** are the agent’s only memory
- Drop the result: the model asks again, forever

Why one small step at a time?

- Plan up front, then execute blindly: “all of its judgment at the point of maximum ignorance”
- Reasoning alone: **a sealed room**; it elaborates a fabrication
- Acting alone: **a sleepwalker**; it repeats gestures
- Interleave: thought, action, observation. **Grounding** “is a schedule”



Four parts inside a harness

1. **Model client:** sends messages and tool definitions; stateless
2. **Tools:** name, description, schema, and the function that runs
3. **Message history:** the run's entire memory
4. **The loop:** ties the three together

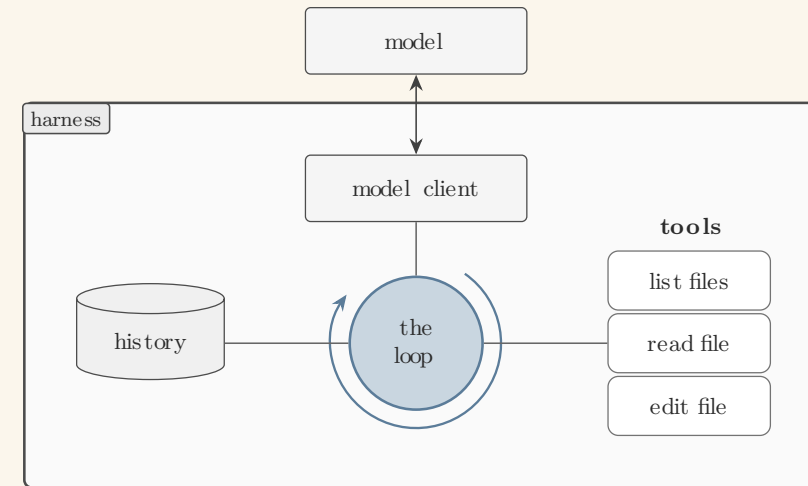


Figure from Chapter 3

“An agent, in one line, is a model plus a harness.”

Chapter 3, “A Minimal Agent from Scratch”

Under seventy lines, no magic

```
function execute(call):  
    if call.name not in TOOLS:  
        return result(call.id,  
            error: "no such tool: " + call.name)  
    outcome = try TOOLS[call.name].fn(call.arguments)  
    if the call failed:  
        return result(call.id,  
            error: plain_description(outcome))  
    return result(call.id, outcome)
```

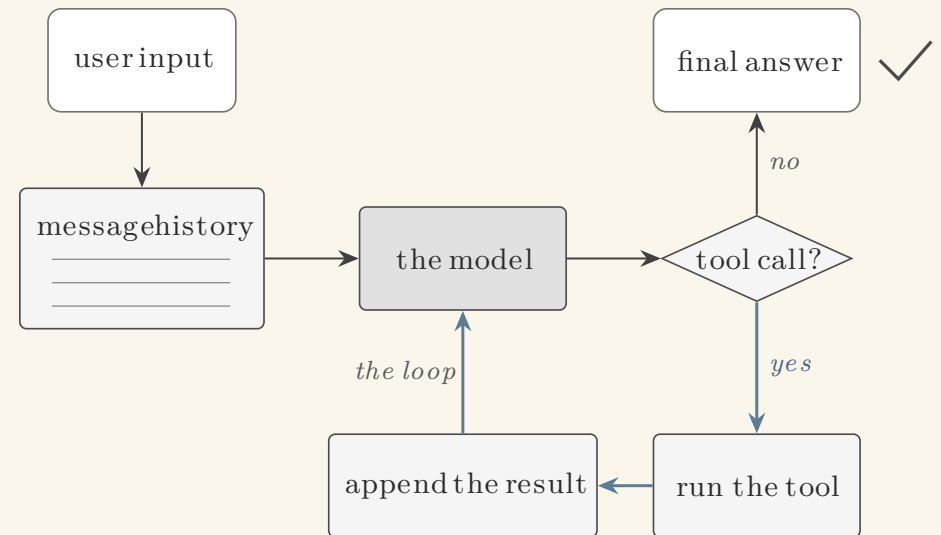


Figure from Appendix A

- “Three cases, one policy: everything becomes a result.” The `id` ties each result to its request
- Tools: `list_files`, `read_file`, `edit_file` • `MAX_STEPS = 20`, illustrative

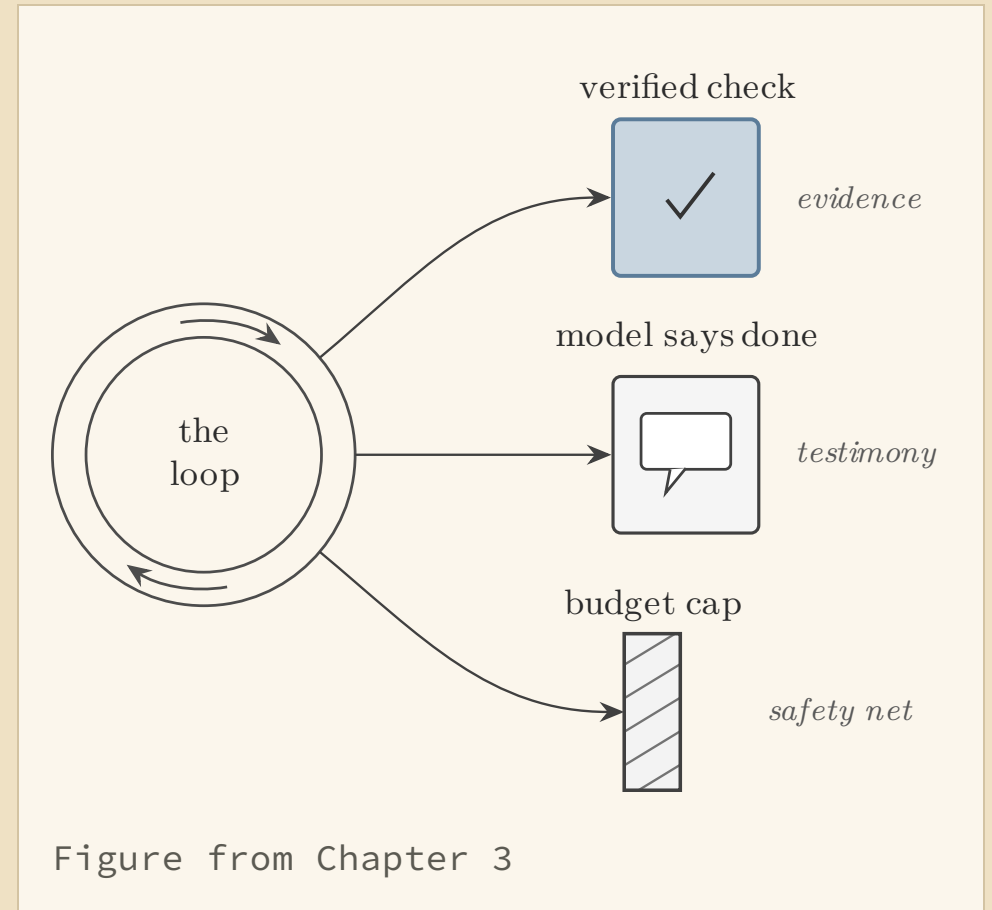
Watch the history grow

Pass	The model asks for	The result appended
1	<code>list_files()</code>	README.md, app.py, greetings.py, tests/
2	<code>read_file("greetings.py")</code>	<code>GREETING = 'Hello' ...</code>
3	<code>edit_file(...'Hello' → 'Welcome')</code>	ok: 1 replacement made
4	<code>read_file("greetings.py")</code>	<code>GREETING = 'Welcome' ...</code>
5	no tool calls: “Done. ...”	the loop returns finished

“Your program is dumb on purpose; the transcript only looks smart.”

Three exits, ranked by trust

- **Verified check:** suite green, file parses. “The model’s ‘done’ is testimony; a green test is evidence.”
- **Budget:** steps, tokens, time, money; mandatory, set first. Fires? **Fail loudly**, save the transcript
- **Errors:** can the model act on it? Feed it back. Evidence the run is off the rails? Halt



Replace the model with a script

```
SCRIPT = [ call list_files(),
           call read_file("greetings.py"),
           call edit_file(...),
           call read_file("greetings.py"),
           say "Done. GREETING now 'Welcome'." ]

function scripted_send(request):
    # replaces provider.send
    return next response in SCRIPT
```

- The loop, the tools, the stop rules are **ordinary code**: exact assertions
- Test 1: a stub that never says done → stopped at MAX_STEPS
- Test 2: drop an append → the test must catch it
- Fast, free, deterministic; no API key

Lab pseudocode, not from the book; the testing approach is Chapter 15's

Shouldn't I just use a framework?

- What a framework adds is **plumbing**: persistence and resume, retries, tracing, memory management, multi-agent orchestration
- Every item is a harness part; none changes a single pass
- So it “can make an agent more dependable and cannot make it smarter”

“Build from scratch to understand. Adopt a framework to scale—and only once you can name what it is saving you.”

Chapter 3, “Stop Conditions, Budgets, and Frameworks”

Five things to keep

- The model proposes; your code disposes. **Side effects happen only in your code**
- A schema guarantees shape, never truth: validate the content
- Six failure modes; each answered by engineering, not sterner prompts
- p^n is the optimistic case; shrink n , raise effective p , checkpoint
- An agent is a model plus a harness: four parts, a loop, three exits, a budget first

Before next week

Lab 1 and reading

Lab 1: the minimal agent, no API

- Type in Appendix A in your language
- Swap `provider.send` for a **scripted model client** replaying the Hello → Welcome run
- Three tools, errors as results, `MAX_STEPS` with a loud `stopped`
- Two tests that break the loop on purpose

Reading for week 3

- Ch. 4, Planning, Reasoning, and Self-Correction `full book`
- Ch. 5, Tools and the Action Space `full book`

This week's reading: Ch. 2 §4–6 (free) • Ch. 3 • App. A (full book)

Engineering AI Agents • Week 2

The model's "done" is testimony. What is your evidence?

Slides from the teaching kit of *AI Agents, Engineered* by Enrique Gutiérrez, CC BY 4.0 • aiagentsengineered.com/teach/

Reuse, adapt and translate with attribution • creativecommons.org/licenses/by/4.0/

