PARTICIPANT TEXT · PART 2 OF 3

Agentic AI for Teaching & Academic Work

Part 2 — Building

Chapter 5 The Actor model

Chapter 6 Structured output

Chapter 7 Agent architecture

Chapter 8 Tools

Faculty Development Workshop

Symbiosis Institute of Technology, Pune

Facilitator: Mr. Dinesh Wadhwani

10–11 September 2026

Part 2 — Building

Part 1 was about understanding. This part is about construction.

The order is deliberate. Chapter 5 covers the platform mechanics, because you cannot build anything without knowing where code goes and what it costs. Chapter 6 covers getting machine-readable output, because everything after it depends on being able to consume what the model produced. Chapter 7 covers agent architecture, and Chapter 8 covers tools, which is where the failure from Chapter 2 finally gets solved properly.

By the end of Part 2 you will have written an agent. Not configured one, written one, and understood every line.

Chapter 5 — The Actor model

5.1 Why a platform at all

You can do a great deal from a chat window, and for exploratory work you should. But three things become impossible.

It cannot run without you. A chat window requires a human to type. Anything that should happen on a schedule, or in response to an event, or over sixty records, needs somewhere to execute.

It cannot be given to a colleague. Your prompt lives in your history. A colleague cannot use it, improve it, or inherit it when you change departments.

It cannot use tools. The chat interface may have some built in, but you cannot add a function that computes attainment from your marks sheet. That requires code, and code requires a runtime.

A platform solves all three. This book uses Apify, and Chapter 5 explains its model, but the concepts transfer to anything similar. What matters is the shape of the thing, not the vendor.

5.2 Three sections, always

On Apify, a program is called an **Actor**. It takes structured input, does something, and writes structured output.

Every Actor in this book has this shape:

```
import { Actor } from 'apify';
await Actor.init();

// 1. READ THE INPUT
const input = await Actor.getInput() ?? {};

// 2. DO THE WORK
//    ...ordinary JavaScript...

// 3. WRITE THE OUTPUT
await Actor.pushData(results);

await Actor.exit();
```

Only the middle section ever changes. A grade calculator, an agent that loops for five turns, a three-stage grading pipeline, a scheduled attendance system: all of them are input, work, output.

This is worth stating plainly because it removes most of the intimidation of a new platform. There is no framework to learn, no lifecycle to memorise, no dependency injection. There is a function that reads input, ordinary code, and a function that writes output.

5.3 The four files

The Node.js template gives you a file tree. Four files matter, and you will only ever edit two.

File	What it does	You edit it?
`src/main.js`	Your code	**Constantly**
`.actor/input_schema.json`	Defines the input form	**Yes**
`.actor/actor.json`	Actor metadata	No
`package.json`	Dependencies	Rarely

The one exception on `package.json` is adding a library. When you need to send email, you add `nodemailer` to the dependencies and the platform installs it at build time. Otherwise leave it alone.

5.4 The input schema is the interface, and this matters more than it looks

`input_schema.json` declares fields. Each field appears as a form control on the Actor's Input tab, which someone fills in before running.

That sounds like a convenience. It is the single most important design decision on the platform, for a reason that has nothing to do with usability.

THE FAST PATH AND THE SLOW PATH

What you change	Rebuild needed?	Elapsed time
Anything in the input schema	**No**	About ten seconds
Anything in `main.js`	Yes	One to two minutes

Now consider what you actually spend your time doing. From Chapter 4: prompt iteration. Trying a phrasing, reading the output, tightening a constraint, trying again. That is most of the work, and it is work you do dozens of times.

If the prompt is hardcoded in `main.js`, every iteration costs two minutes. Twelve iterations is twenty-four minutes of waiting. The work becomes unpleasant and you stop doing enough of it, which means your prompts stay bad.

If the prompt is an input field, twelve iterations costs two minutes total.

So put your prompts in the input schema. Also the model name, the temperature, the thresholds, the delays, anything you will tune. The question to ask of every value in your code is:

> Will I want to change this without changing the logic?

If yes, it belongs in the schema.

There is a second benefit that emerges later. An Actor whose prompts and thresholds are inputs can be handed to a colleague who uses it for their own course by filling in a form. They never open the code. That is the difference between a script you wrote and a tool your department can use.

What a field declaration looks like

```
{
  "title": "Attendance Alert System",
  "type": "object",
  "schemaVersion": 1,
  "properties": {
    "thresholdPercent": {
      "title": "Attendance threshold (%)",
      "type": "integer",
      "editor": "number",
      "description": "Students below this are flagged.",
      "default": 75
    },
    "promptTemplate": {
      "title": "Message prompt",
      "type": "string",
      "editor": "textarea",
      "prefill": "You are a college professor writing to..."
    },
    "sendMode": {
      "title": "What to do with the messages",
      "type": "string",
      "editor": "select",
      "enum": ["review_sheet", "send"],
      "enumTitles": ["Write for review (safe)", "Actually send"],
      "default": "review_sheet"
    }
  },
  "required": ["thresholdPercent"]
}
```

Note `enum` on the last field. It restricts the value to a fixed set, which means an invalid mode cannot reach your code at all. Constraining input at the boundary is cheaper than validating it inside.

Note also `default` versus `prefill`. A default applies silently when the field is absent. A prefill puts a value in the form for the user to see and edit. For prompts you want prefill, because the user should see what they are changing.

5.5 Builds

Editing code requires a build, which takes one to two minutes and produces a container image.

Two things about builds cause avoidable frustration.

A build failure is almost always a syntax error in the JSON. A missing comma in `input_schema.json` fails the build with a message that does not obviously say so. Check the JSON first, every time.

Environment variables are applied at build time. This catches roughly a third of every group. You add your API key, run the Actor, and it reports the key as missing. The key is there. The container was built before you added it.

Add the variable, then build. Change the variable, build again.

5.6 Secrets

Your API key goes in **Settings → Environment variables**, with the **Secret** box ticked. Your code reads it as `process.env.GEMINI_API_KEY`.

It does not go in an input field. The reason is specific: input values are stored with the run record and visible in the run history. A secret environment variable is encrypted and never displayed.

This is worth doing correctly now, on a workshop key that does not matter, so that it is automatic later when the key is attached to something that bills.

The same applies to anything credential-shaped. An email app password, a database connection string, a webhook URL with a token in it. If leaking it would be bad, it is a secret environment variable.

5.7 Memory, and what you are actually paying for

Apify bills in **Compute Units**. One CU is one gigabyte of RAM running for one hour.

The memory allocation is set on the Input tab before you run, not in code. It defaults higher than you need.

At 512 MB for thirty seconds, a run costs a fraction of a cent. At 4 GB it costs eight times as much for identical work, because you are billed for allocated memory whether or not you use it.

Set the lowest memory that works. On a free tier with five dollars of monthly credit, this is the difference between a few hundred runs and a few thousand.

The general lesson generalises beyond this platform: in any usage-billed environment, find out what the meter measures before you build, not after the first invoice.

5.8 Logging is your debugger

Every `console.log` appears in the Log tab while the run is in progress. There is no step debugger, and you will not miss one.

Log at the boundaries. What came in, what each step produced, what went out. Log the decisions, not just the results.

```
console.log(`Read ${students.length} student records.`);
console.log(`${flagged.length} below ${threshold}%. ${skipped.length} skipped.`);
for (const s of skipped) console.log(`  SKIPPED ${s.roll_no}: ${s.reason}`);
```

That third line is the useful one. A count of skipped records tells you something is wrong; a list of which records and why tells you what.

> **THE DEBUGGING RULE**
>
> When an Actor misbehaves, do not re-read your logic first. Read the log top to bottom and find **the first line that is not what you expected**.
>
> That line is your bug. Everything after it is a symptom.
>
> This is the same discipline as reading a stack trace from the bottom, and it saves the same amount of time.

5.9 Output and retention

`Actor.pushData()` writes records to a **dataset**, which you can view as a table and export as JSON, CSV or Excel.

On the free tier, datasets are retained for **seven days**.

This catches people out in a specific and annoying way. You build something useful during a workshop, intend to use it the following month, and find the results gone. The Actor is still there; the data it produced is not.

Export at the end of every run that matters. It is one click and it is the difference between having your work and not.

5.10 Scheduling, and the difference between platforms

An Actor can be scheduled from the **Schedules** tab with a cron expression.

When	Cron
Fridays at 5 pm	`0 17 * * 5`
Weekdays at 8 am	`0 8 * * 1-5`
First of the month at 9 am	`0 9 1 * *`

The schedule runs on the platform's servers. Whether your laptop is on is irrelevant.

This is the sharpest practical difference between Apify and the local alternative. n8n, running on your machine, has a Schedule Trigger that fires **only while n8n is running**. On a desktop that means only while the terminal window is open. Friday five o'clock arrives, the machine is off, nothing happens, and nobody discovers this until someone asks why they were not notified.

If something must run unattended, it needs a host that is always on. Apify's own scheduling, a college server, or a hosted service. A desktop tool is a manual tool, whatever its trigger says.

5.11 The other platform, and what the comparison teaches

You will build the same thing twice during the workshop: once as an Actor and once as an n8n workflow. The comparison is not busywork.

	Apify Actor	n8n
What it is	Cloud platform, you write code	Local tool, you connect nodes
The agent loop	**You write it**	Hidden inside a node
Install needed	None, browser only	Node.js on the machine
Iteration speed	Slower, needs a build	Instant per node
Data visibility	Log output	Input and output panel per node
Where your work lives	Your own account	That machine
Runs when the machine is off	Yes	**No**
Connecting to Gmail, Sheets, Slack	You write it	Built in

Two things are worth extracting.

n8n hides the loop; the Actor shows it. This recurs at every level. In n8n, data flows as an array of items and each node runs once per item, so the iteration is implicit and you never write it. In your Actor you wrote `for (const student of students)` yourself. Same result, opposite philosophy.

The same is true one level up. n8n's Agent node contains the reasoning loop and you cannot open it. In Chapter 7 you write that loop, and you can therefore change the stopping condition, insert a validation step, or log whatever you want.

Neither is better. They solve different problems. n8n is genuinely superior for anything that needs to touch a dozen external services, because those integrations already exist and writing them yourself is a waste of your life. Apify is superior for anything you will maintain, anything that must run unattended, and anything where you need to understand exactly why it did what it did.

The most common sensible arrangement in real use is n8n handling the schedule and the connections, calling an Actor that does the work.

5.12 Which to reach for

Situation	Reach for
Something you will maintain for a semester	Actor
Something a colleague must be able to run	Actor
Something that must run when your PC is off	Actor
Something that reads Gmail and writes to Sheets	n8n
Something where you must see why it decided	Actor
Getting something working in twenty minutes	n8n
Teaching someone what an agent is	Actor, because the loop is visible

5.13 Summary

An Actor reads structured input, does work, writes structured output. Only the middle changes.

The input schema is not a convenience. Putting prompts and tunable values there converts a two-minute iteration cycle into a ten-second one, which determines whether you do enough iteration to get good results.

Environment variables apply at build time. Add the key, then build.

Set memory low. You are billed for allocation, not use.

Free-tier data lasts seven days. Export.

Scheduled Actors run on the platform. Scheduled workflows on your desktop run only while your desktop is running, which is a different thing entirely.

Chapter 6 — Structured output

6.1 The problem with prose

Your first AI-powered Actor produces text. A paragraph of feedback per student, say. It works, and it is useful, and then you try to do something with it.

You want to flag the students who need a meeting. The information is in there, expressed in sentences, differently each time. There is no reliable way to extract it. You cannot search for a phrase, because the phrasing varies. You cannot parse a position, because there is no structure.

Prose is a terminal format. It is for humans, at the end of a pipeline. If anything downstream needs to consume the output, the output must be structured.

6.2 Asking for JSON

You specify the structure in the prompt, literally, by showing it.

```
Return ONLY valid JSON in exactly this structure. No markdown
code fences. No commentary before or after.

{
  "student_name": "string",
  "percentage": 0,
  "performance_band": "one of: Excellent, Good, Satisfactory, Needs Improvement, At Risk",
  "strengths": ["string", "string"],
  "priority_action": "string - one specific thing to do next",
  "feedback_text": "string - 3 to 4 sentences",
  "needs_meeting": true
}

Set needs_meeting to true ONLY if the percentage is strictly
below 40. Compute the percentage first, then apply this rule.
```

Three things about this specification are doing work.

The structure is shown, not described. From Chapter 4: an example is a direct sample from the region of output space you want. Describing the fields in prose is far less effective than showing them.

The types are indicated by the placeholder values. `0` signals a number, `true` signals a boolean, `"string"` signals a string. This is not a formal schema and the model may still return `"42"` instead of `42`, but it substantially improves the hit rate.

Constrained fields list their options inline. `"one of: Excellent, Good, ..."` narrows the distribution for that field considerably. Without it you will get synonyms, and synonyms break any code that switches on the value.

6.3 Format drift, and handling it rather than fighting it

You asked for no code fences. You will get code fences.

From Chapter 3: in the training corpus, JSON appears inside triple backticks overwhelmingly often. The fence is the probable continuation, and your instruction competes against that prior without reliably winning.

You could escalate the instruction. It helps a little. The better response is to stop fighting and handle it:

```
function extractJson(raw) {
    // Models wrap JSON in markdown fences habitually, despite
    // being told not to. Strip them first.
    const cleaned = raw
        .replace(/```json/gi, '')
        .replace(/```/g, '')
        .trim();

    // Then locate the outermost brace pair, which survives any
    // preamble like "Here is the JSON you requested:".
    const start = cleaned.indexOf('{');
    const end   = cleaned.lastIndexOf('}');

    if (start === -1 || end === -1 || end < start) {
        throw new Error(`No JSON object found. Reply began: ${raw.slice(0, 200)}`);
    }

    return JSON.parse(cleaned.slice(start, end + 1));
}
```

Three lines of defence: strip the fences, find the braces, and fail with the raw text visible so you can see what actually came back.

That last point is not decoration. When this throws, the error tells you whether the model returned prose, returned an apology, returned nothing, or returned malformed JSON. Those are four different problems with four different fixes, and an error message reading "unexpected token" distinguishes none of them.

6.4 One failure must not destroy the batch

Put the parse inside a try/catch, **inside the loop**, and record the failure rather than throwing.

```
for (const student of students) {
    try {
        const raw    = await callGemini(buildPrompt(student));
        const parsed = extractJson(raw);
        results.push({ ...parsed, parse_error: false });

    } catch (err) {
        console.log(`  FAILED for ${student.name}: ${err.message}`);
        results.push({
            student_name:  student.name,
            parse_error:   true,
            error_message: err.message
        });
    }
}
```

Without this, one malformed response on student seventeen ends the run and you lose the sixteen that worked and the thirteen that had not run yet.

With it, student seventeen is flagged with the raw output preserved, the other twenty-nine complete normally, and you go back and look at just the one.

This is not defensive over-engineering. Malformed JSON arrives often enough that you will meet it on your first substantial run.

6.5 Syntactic validity is not correctness

Here is the mistake that separates people who have built these systems from people who have read about them.

You now have parseable JSON with the right field names and plausible types. It does not follow that the values are right, and the specific way they go wrong is instructive.

Return to the specification in 6.2. The rule was:

```
Set needs_meeting to true ONLY if the percentage is strictly
below 40.
```

Now consider a student who scored 21 out of 50. That is 42 per cent. The rule says `needs_meeting` should be **false**.

A substantial fraction of runs will return **true**.

Nothing is broken. The model read a rule and a situation, and produced the more probable continuation, which is that a student at 42 per cent is a student to worry about. That is a defensible judgement. It is not the rule you gave it.

THE FAILURE CLASS THIS REPRESENTS

This is the class of error you will spend the rest of your time with this technology learning to catch.

It is not a crash. It is not malformed. It is not obviously wrong. It is a plausible answer that quietly overrode your instruction with its own judgement, and it will pass any inspection short of checking the rule.

The design response is to validate semantics in code. You gave the rule; the code can check it.

```
// The model was told to apply this rule. Verify that it did.
const expected = (parsed.percentage < 40);

if (parsed.needs_meeting !== expected) {
    console.log(`  RULE VIOLATION for ${parsed.student_name}: `
              + `percentage ${parsed.percentage}, `
              + `model said needs_meeting=${parsed.needs_meeting}, `
              + `rule says ${expected}. Overriding.`);
    parsed.needs_meeting   = expected;
    parsed.rule_overridden = true;
}
```

Note what this does. It does not just detect the violation; it corrects it, and it records that a correction happened. If `rule_overridden` starts appearing frequently, your prompt needs work. If it appears once in fifty, the code caught it and nothing downstream was affected.

The general principle: **if a rule can be checked in code, check it in code.** Asking the model to follow a rule is a request. Verifying the rule is a guarantee.

6.6 Where to constrain, and where not to

There is a temptation, once you have seen the above, to move everything into code. That is also a mistake, in the other direction.

Kind of output	Where it should come from
A number that must be exact	Code. Always.
A boolean derived from a stated rule	Model may propose, code must verify
A category from a fixed list	Model, constrained by an enum, verified in code
A judgement across a fixed rubric	Model, reviewed by a second model or by you
Prose for a human to read	Model. Code cannot do this.

The line runs between things with a determinate answer and things requiring judgement. Determinate answers belong in code. Judgements belong to the model, with verification proportional to the stakes.

Writing warm, specific, useful feedback for a student is genuinely something a model does better than any code you would write, and better than a template. That is a real capability and you should use it. Computing whether 42 is less than 40 is not.

6.7 Type coercion, and the small annoyances

Three practical points that will otherwise cost you twenty minutes each.

Numbers arrive as strings. The model may return `"percentage": "42"`. Coerce explicitly with `Number()` before comparing, because `"42" < 40` is false but `"9" < 40` is also false in some comparisons and true in others depending on what is on the other side.

Optional fields go missing. If the model has nothing to say for `strengths`, it may omit the key entirely rather than returning an empty array. Use `parsed.strengths ?? []` rather than assuming presence.

Enums drift towards synonyms. You asked for one of five bands and got "Very Good", which is not on the list. Validate against the list and either coerce or flag:

```
const BANDS = ['Excellent', 'Good', 'Satisfactory', 'Needs Improvement', 'At Risk'];
if (!BANDS.includes(parsed.performance_band)) {
    parsed.band_invalid   = parsed.performance_band;
    parsed.performance_band = null;
}
```

Preserving the invalid value in a separate field is better than discarding it, because it tells you what the model wanted to say and therefore how to fix the prompt.

6.8 When not to ask for JSON

A brief counterweight. JSON has a cost: it constrains the output, and constraint sometimes reduces quality.

If you want a paragraph of feedback and nothing else, ask for a paragraph. Wrapping it in `{"feedback_text": "..."}` adds parsing risk for no benefit, and the model spends attention on producing valid syntax rather than good prose.

Ask for structure when something downstream will consume it. Ask for prose when a human is the next reader.

The middle case, where you want both a structured judgement and prose, is well served by asking for JSON with a prose field inside it. That works well and is what the specification in 6.2 does.

6.9 Summary

Prose is a terminal format. Anything a later step must consume has to be structured.

Specify the structure by showing it, indicate types with placeholder values, and list the options for constrained fields inline.

Expect markdown code fences and strip them. Locate the outermost brace pair. Fail with the raw text visible so you can diagnose what actually happened.

Put the parse in a try/catch inside the loop, so one bad response does not destroy the batch.

Syntactic validity is not correctness. A model will return well-formed JSON that quietly substitutes its own judgement for your stated rule. If a rule can be checked in code, check it in code, correct the violation, and record that you did.

Determinate answers belong in code. Judgements belong to the model, with verification proportional to the stakes.

Chapter 7 — Agent architecture

7.1 Where a fixed sequence stops

You now have an Actor that calls a model once per record and parses the result. That covers a great deal of academic work.

Consider something it cannot do.

A student writes: "*I did not understand the third normal form example from the class on 14 August. Can you explain it differently?*"

To answer this properly, something must:

1. Work out which topic and which class this refers to
2. Retrieve your actual material for it
3. Notice the request is for a *different* explanation, and avoid repeating the original
4. Produce an explanation at the right level
5. If the student remains confused, try another approach

Steps 1 and 5 are the problem. In a fixed sequence you decide in advance what happens at every step. Here, what happens depends entirely on what the student said and how they respond to what you say back.

Crossing that boundary requires three additions.

7.2 Tools, memory, and the loop

Tools

A tool is a function the model can call. A calculator. A lookup over your notes. A web fetch. A date calculation.

The critical property is this: **you provide several tools and do not tell the model which to use.** It reads the request and chooses. That delegation of choice is the first thing that makes a system an agent rather than a sequence, and it is where the leverage comes from.

Memory

A sequence forgets everything between steps. An agent must remember what it has already tried, or it will call the same tool repeatedly, get the same answer, and make no progress.

The loop

The essential addition:

```
think → act → observe → think → act → observe → ... → answer
```

The model reasons about what to do, does it, looks at what came back, and reasons again. It stops when it judges the task complete.

> **THE DEFINITION**
>
> A sequence is a path you drew.
>
> An agent is a model choosing its own path, one step at a time, using tools you gave it.

7.3 The five components

Every agent, in every framework, in every language, is these five things. Frameworks differ in naming and syntax and in very little else.

Component	Role	Get it wrong and
Model	The reasoning engine	Weak models loop; expensive ones bankrupt you
Instructions	Who it is, what it must and must not do	It improvises, confidently
Tools	What it can actually do	It guesses instead of checking
Memory	What it retains across steps	It repeats itself and makes no progress
Loop control	When to stop	It runs forever, or gives up too early

Learn these once and every agent library you meet afterwards is a matter of looking up method names.

7.4 The loop, in five lines of structure

Before the code, the shape:

```
1. Build a prompt: instructions + tool list + everything tried so far + the question
2. Ask the model what to do next
3. If it names a tool, run it and push the result into history
4. If it gives an answer, stop
5. Repeat until answered, or the iteration cap is reached
```

That is the entire idea. Everything in agent engineering is refinement on those five lines.

The pattern has a name in the literature, **ReAct**, for reason-and-act. The name is more imposing than the mechanism.

7.5 A worked trace

Here is what actually happens, for the question "*What is the CO3 attainment for this internal assessment?*", with three tools available.

```
--- Iteration 1 ---
THOUGHT:  I need the marks data before I can compute anything.
          There is a tool called read_marks_data.
ACTION:   read_marks_data()
OBSERVE:  20 records, each with roll_no and q1 through q5.

--- Iteration 2 ---
THOUGHT:  I now need to know which questions assess CO3, and
          what threshold applies. get_co_mapping will tell me.
ACTION:   get_co_mapping()
OBSERVE:  CO3 is assessed by q2 and q4. Threshold 0.6.
          Max 10 marks per question.

--- Iteration 3 ---
THOUGHT:  I have the data and the mapping. I must not estimate
          this. I will use the calculator.
ACTION:   calculate_attainment({questions:["q2","q4"],
                                threshold:0.6,
                                max_per_question:10})
OBSERVE:  11 of 20 students met the threshold. 55.00 per cent.

--- Iteration 4 ---
THOUGHT:  55 per cent falls in the 50 to 59 band, which is
          Level 1 on the institute scale. I have everything.
ANSWER:   Using questions q2 and q4, threshold 60 per cent,
          across 20 student records: CO3 attainment is 55.00
          per cent (11 of 20), which is Attainment Level 1...
```

Four things in this trace deserve attention.

The agent decided the sequence. Nothing told it to read the marks before the mapping, or to fetch both before calculating. It worked that out from the tool descriptions.

It used a calculator for the arithmetic. Deliberately, because the instructions forbade doing it itself. Chapter 8 develops why this is the whole game.

It stopped when it had enough. That judgement is fallible in both directions, which is why the cap exists.

Each thought is conditioned on the previous observations. From Chapter 2, this is why the loop works at all. The reasoning is not a report of a plan formed earlier. Each thought is generated fresh, given everything that has happened, and the accumulated history is what makes each successive thought better informed.

7.6 Memory is an array

Here is the part that surprises people, and it is worth stating baldly because the literature makes it sound more elaborate than it is.

Memory, in the agent you will write, is a JavaScript array.

Each iteration appends an object containing the thought, the tool called, the arguments, and the result. Then the entire array is formatted as text and pasted into the next prompt.

```
const trace = [];

// ... inside the loop, after running a tool ...
trace.push({
    thought: step.thought,
    tool:    step.tool,
    args:    step.args ?? {},
    result
});
```

And the prompt builder:

```
function buildPrompt(trace) {
    const history = trace.length === 0
        ? '(nothing yet - this is your first step)'
        : trace.map((step, i) =>
            `Step ${i + 1}:\n` +
            `  thought: ${step.thought}\n` +
            `  called : ${step.tool}(${JSON.stringify(step.args)})\n` +
            `  result : ${JSON.stringify(step.result).slice(0, 1500)}`
          ).join('\n\n');

    return `${systemPrompt}

TOOLS YOU CAN CALL:
${toolCatalogue()}

WHAT HAS HAPPENED SO FAR:
${history}

USER QUESTION:
${question}

Reply with ONE JSON object and nothing else. No markdown fences.

To call a tool:
{"thought": "why you are doing this", "tool": "tool_name", "args": {}}

To give your final answer:
{"thought": "why you can answer now", "answer": "your answer"}`;
}
```

This follows directly from Chapter 2's observation that the model is stateless. There is nowhere for memory to live except the context, so memory *is* the context, assembled fresh on every call.

The truncation, and why it is there

Notice `.slice(0, 1500)` on the result.

Every iteration includes the whole history. A tool that returns something large, a full marks table, a fetched web page, makes each subsequent prompt bigger. By iteration four you may exceed the context window, or simply pay for a very long prompt to no benefit.

Truncating stored results caps the growth. If your agent starts failing on the fourth iteration with a token error, this is why, and this is the fix.

The three kinds of memory

For completeness, though you will rarely need the third.

Type	Holds	Use for
Full trace	Everything tried this run	Almost everything. Start here.
Windowed	The last N steps only	Long conversations that outgrow the context
Long-term	Facts retrievable across sessions	Rarely needed at departmental scale

A doubt-solving agent holding a conversation wants a window of perhaps ten exchanges. It should remember the current conversation and forget last week's, which is both what you want and what avoids the privacy problem of a shared machine retaining one student's queries into the next student's session.

7.7 Writing agent instructions

Agent instructions are prompts, so RCTFG from Chapter 4 still applies. Three things are added.

Describe the tools, and when to use them

Do not rely on tool names. The model chooses by reading descriptions, and a vague description produces bad choices.

```
TOOLS:
- read_marks_data: Returns all student marks. Call this first for
  any question about scores or attainment.

- get_co_mapping: Returns which questions assess which CO, and the
  institute's attainment thresholds. Call before any calculation.

- calculate_attainment: Computes attainment percentage. You MUST
  use this for every calculation. NEVER compute percentages,
  counts or averages yourself.
```

That last instruction is the most important line in the prompt, and Chapter 8 explains why it must be absolute with no carve-outs.

Say when to stop

```
Once you have what you need, answer immediately. Do not call
additional tools to double-check a result you already have.
```

Without this, agents fail in the opposite direction from runaway looping. They become over-cautious, re-reading files, recalculating, spending iterations on reassurance. This is more common in practice than the runaway case.

Say what to do when stuck

```
If a tool returns an error or empty data, report exactly what you
tried and what happened. Do NOT estimate from partial data. Do
not retry the same call more than twice.
```

THE FAILURE THIS PREVENTS

Without this instruction, an agent whose tool fails will frequently fall back on estimation and present the estimate as a result, with nothing in the output indicating that anything went wrong.

For an attainment figure headed into an accreditation document, that is not a minor bug. It is a fabricated number in a formal submission, produced by a system you told your department to trust.

Write this into every agent you build.

7.8 The iteration cap

A hard limit, in code, not a request in the prompt.

```
for (let i = 1; i <= maxIterations; i++) {
```

Five is right for everything in this book.

The failure it prevents is an agent that calls a tool, dislikes the result, calls it again, and continues indefinitely. On a free tier this exhausts your daily quota in minutes. On a billed account it produces an alarming invoice.

The reason it must be in code rather than in the prompt follows from Chapter 3's instruction decay. An instruction saying "do not use more than five tool calls" is a token sequence competing for influence. A `for` loop with an upper bound is a guarantee.

Set it before your first run, not after your first incident.

7.9 Fewer tools than you think

> **THE DESIGN RULE THAT MATTERS MOST**
>
> An agent with three well-described tools substantially outperforms one with ten.
>
> With too many, it picks wrong, calls several unnecessarily, burns iterations deciding, and produces a trace you cannot follow.
>
> Start with two. Add a third only when you have watched it genuinely fail for want of it.

The instinct is to give the agent every capability available, on the reasoning that more options mean more can be achieved. The opposite holds, and the mechanism is clear from Chapter 2.

Tool selection is a prediction over the tool descriptions in the context. Ten descriptions means a wider distribution over ten options, several of which will be superficially plausible for any given request. The model does not evaluate tools; it predicts which one is a probable continuation given the description text and the question. More candidates, worse discrimination.

Every tool you add makes every decision harder.

7.10 Read the trace

The reasoning trace is the primary debugging surface for an agent, and reading it is a habit worth forming immediately.

An agent that produced a wrong answer almost always shows you where it went astray. A tool returned something unexpected. The instructions were ambiguous at one specific decision. It chose a tool whose description was too broad. All of this is visible.

By contrast, debugging an agent by re-reading its final output is guesswork. The output tells you that something is wrong, not what.

WHAT TO LOOK FOR

In the trace	Means
A tool called that should not have been	Its description is too broad
A needed tool never called	Its description is too narrow, or the instruction is missing
The same tool called twice with the same arguments	It has no stopping instruction, or the result was unhelpful
A thought that does not follow from the observation	The result was formatted confusingly
Reaching the cap without answering	Missing stopping instruction
Arithmetic in a thought rather than a tool call	The prohibition needs strengthening

This is one of the substantial advantages of writing the loop yourself. You control what gets logged. In a framework you see whatever it chose to expose, which is often the final answer and a summary.

7.11 Why write the loop rather than use a framework

CrewAI, LangGraph, Mastra, n8n's Agent node. All of them implement what Chapter 7 describes, and all of them hide it.

Configure one and it works, and you have learned an import statement. Write one and you understand the mechanism, after which every framework is a matter of looking up method names, because they are all doing this.

There are also three practical benefits.

You can change the stopping condition. Perhaps you want the loop to end when a validation tool returns success, rather than when the model says it is done. That is three lines in your loop and impossible inside a node.

You can insert steps. A check between iterations. A log of the exact prompt sent. A hard rule that a particular tool must be called at least once before answering.

You can see everything. No abstraction is deciding what you get to know.

There is a cost, and it is real: you write and maintain sixty lines that someone else would have maintained for you. For a production system serving thousands of users, use the framework. For understanding, and for the scale of thing you will build for your own courses, write it.

7.12 Summary

An agent is a fixed sequence plus three things: tools it chooses between, memory of what it has tried, and a loop.

Five components: model, instructions, tools, memory, loop control.

The loop is five lines of structure. Build a prompt from instructions plus tools plus history plus question, ask what to do, run the tool, append the result, repeat until answered or capped.

Memory is an array, because the model is stateless and the context is the only channel. Truncate stored results or the context grows until it breaks.

Instructions must describe the tools, say when to stop, and say what to do when a tool fails. The third is the one that prevents fabricated figures.

The iteration cap goes in code, not in the prompt.

Fewer tools beat more, because tool selection is a prediction and more candidates mean worse discrimination.

Read the trace. It is the only thing that tells you why.

Chapter 8 — Tools

8.1 The demonstration

This is best experienced rather than read, and the workshop runs it live, but the shape is worth having in writing.

Take a marks table for twenty students, five questions each. Paste the whole thing into a prompt and ask: CO3 is assessed by questions two and four, each out of ten, and a student attains CO3 by scoring at least sixty per cent of the combined marks. How many students attained CO3, and what is the attainment percentage?

You will get a clean, specific, confident answer.

Now count the rows by hand. The true figure is eleven of twenty, which is fifty-five per cent, Level 1 on a typical institute scale.

Most runs return something between sixty and seventy-five per cent.

Now run it twice more without changing anything. The number will usually change.

8.2 What that demonstration establishes

Three things, and the third is the one that matters.

The model cannot do this. From Chapter 2: no arithmetic unit, tokenisation that destroys place value, sampled output. This is architectural.

The error is not random noise. It clusters plausibly near the truth. It is never four hundred per cent, never negative, never zero. It is always a number a reasonable person might believe.

A calculation that changes between runs was never a calculation. This is the five-second test you should apply to any number a model gives you, and it is diagnostic.

Consider what a sixty-five per cent figure would have meant. Fifty-five is Level 1. Sixty-five is Level 2. The error moved the attainment level by a full band, in an upward direction, on a number destined for an accreditation file.

8.3 The principle

THE MODEL DECIDES. THE TOOL COMPUTES.

The agent chooses **which** calculation to perform and **how to interpret** the result. Both of those are things it is genuinely good at.

The arithmetic is performed by arithmetic.

Notice what does not change when you apply this. Same model. Same data. Same prompt structure. The only change is the division of labour, and the answer goes from plausibly wrong to exactly right.

This single principle separates a system you could put in front of your department from a demonstration.

8.4 The anatomy of a tool

A tool is two things: a description the model reads, and a function that runs.

```
const tools = {
    calculate_attainment: {
        description:
            'Computes CO attainment exactly. ' +
            'args: { "questions": ["q2","q4"], "threshold": 0.6, ' +
            '"max_per_question": 10 }. ' +
            'Returns exact student counts and the attainment percentage. ' +
            'You MUST use this for every attainment calculation. ' +
            'NEVER compute percentages, counts or averages yourself.',

        run: async (args) => {
            const questions = args.questions;
            const threshold = Number(args.threshold);
            const maxPerQ   = Number(args.max_per_question);

            if (!Array.isArray(questions) || questions.length === 0) {
                return { error: 'questions must be a non-empty array, e.g. ["q2","q4"]' };
            }
            if (!threshold || !maxPerQ) {
                return { error: 'threshold and max_per_question are both required
numbers.' };
            }

            const valid = ['q1', 'q2', 'q3', 'q4', 'q5'];
            for (const q of questions) {
                if (!valid.includes(q)) {
                    return { error: `Unknown question column "${q}". Valid: ${valid.join(',
')}` };
                }
            }

            const maxTotal = questions.length * maxPerQ;
            let met = 0;

            for (const row of MARKS_DATA) {
                let scored = 0;
                for (const q of questions) scored += Number(row[q]) || 0;
                if (scored / maxTotal >= threshold) met++;
            }

            return {
                students_assessed:          MARKS_DATA.length,
                students_meeting_threshold: met,
                attainment_percentage:      Number((met / MARKS_DATA.length *
100).toFixed(2)),
                questions_used:             questions,
                threshold_applied:          threshold,
                max_total_marks:            maxTotal
            };
        }
    }
};
```

8.5 The description is a prompt

This is the part people underestimate. The model selects tools by reading their descriptions, and a description is therefore a prompt subject to everything in Chapter 4.

Compare:

```
Name: calc
Description: Does calculations
```

against the description in 8.4.

The first tells the model almost nothing. It will be called for things it cannot do, and not called for things it can.

Three rules for a tool description:

Say what it does. One clear sentence.

Say exactly what it needs. Show the argument shape literally, with realistic values. `args: { "questions": ["q2","q4"], "threshold": 0.6 }` is far more effective than "takes a list of questions and a threshold", for the same reason that showing a JSON structure beats describing it.

Say when to use it, and where it matters, when not to. "Call this first for any question about scores." Or, for a web fetch: "Use ONLY when you need current information from a specific page. Do not use it for general knowledge you already have."

8.6 Errors are how the agent recovers

Look again at the error returns in 8.4. They are complete sentences that name the problem and show the correct form.

This is deliberate, and it is a design decision people miss.

When a tool returns `{ error: 'questions must be a non-empty array, e.g. ["q2","q4]' }`, that error goes into the trace, becomes part of the next prompt, and the agent can correct its call and try again. Inside the loop. Before you see anything.

A tool that throws an exception kills the run. A tool that returns `null` gives the agent nothing to work with, and it will improvise, which is the failure from Chapter 3.

Write your error messages for the model to read. This is an unusual instruction and it is correct. The consumer of that string is a language model deciding what to do next.

8.7 Tool categories for academic work

Category	Examples	Why it exists
Calculation	Attainment, statistics, grade aggregation, date arithmetic	The model cannot be trusted with numbers
Retrieval	Course notes, syllabus, marks sheet, previous papers	The model does not know your material
External data	Web fetch, an institutional API	The model has a training cutoff
Action	Send email, write a sheet, create a file	Producing an effect, not just text
Validation	Check marks total, verify a CO exists, confirm a rule	Catching the agent's own errors

The last category is the most underused and one of the most valuable.

```
    validate_total: {
        description:
            'Checks that a set of marks adds up to the expected total. ' +
            'args: { "marks": [3,4,3], "expected_total": 10 }. ' +
            'Call this before presenting any question paper or mark sheet.',
        run: async (args) => {
            const marks = args.marks;
            if (!Array.isArray(marks)) return { error: 'marks must be an array of
numbers.' };
            const sum = marks.reduce((a, b) => a + Number(b), 0);
            const expected = Number(args.expected_total);
            return sum === expected
                ? { valid: true, total: sum }
                : { valid: false, total: sum, expected,
                    message: `Total is ${sum}, expected ${expected}. Adjust and try
again.` };
        }
    },
```

A tool that returns `{ valid: false, message: "Total is 22, expected 20. Adjust and try again." }` lets the agent detect and fix its own error, inside the loop, before the output reaches you. The mechanism is the same as 8.6: an informative failure is actionable.

8.8 The failure a tool does not prevent

Here is the second-order problem, and it catches people who have understood everything above.

A tool guarantees the calculation is correct. It does not guarantee the agent passed the right arguments.

An agent may correctly call `calculate_attainment` while passing questions two and five, when CO3 is actually assessed by two and four. The arithmetic is flawless. The answer is wrong.

Nothing in the tool can catch this. The tool was asked about q2 and q5, and it correctly computed the attainment for q2 and q5.

THE FIX, AND IT IS THE SAME TRICK AS CHAPTER 4

Require the agent to state its inputs before its result.

```
When reporting any calculated result, state the exact inputs you
used: which questions, which threshold, how many student records.
Do this before stating the result.
```

The output becomes:

> "*Using questions q2 and q4, threshold 60 per cent, across 20 student records: CO3 attainment is 55.00 per cent (11 of 20).*"

A wrong argument is invisible inside a percentage and obvious in a stated list. You will spot "q2 and q5" in half a second. You would never have spotted it inside a number.

This is the same mechanism as requiring CO justifications in Chapter 4. In both cases you are converting an expensive checking task into a cheap one by making the model expose its working. It is the most transferable single technique in this book.

8.9 Read tools and write tools are different animals

A distinction that matters as soon as your agent can affect the world, and one that is easy to overlook because the code looks the same.

A **read tool** fetches something. Calling it twice is harmless. If the agent is uncertain, retrying costs a little quota and nothing else.

A **write tool** changes something. Sends an email. Updates a sheet. Creates a record. Calling it twice sends two emails.

Agents retry. That is what the loop does. An agent that calls `send_email`, gets an ambiguous response, and calls it again has sent a parent two copies of the same message about their child.

Three responses, in order of preference.

Do not give the agent write tools. Have it decide and record decisions; have deterministic code act on those decisions afterwards. This is what the attendance system does: the agent produces a list of decisions, and separate code turns approved decisions into messages. The agent cannot send anything because it has no tool that sends.

Make write tools idempotent. If the tool records what it has already done and refuses duplicates, a retry is harmless.

```
const SENT = new Set();

    send_notification: {
        description: 'Sends one notification to one recipient. ' +
                     'Calling twice for the same recipient has no effect.',
        run: async (args) => {
            if (SENT.has(args.recipient_id)) {
                return { already_sent: true,
                         message: 'Already sent to this recipient in this run. No action
taken.' };
            }
            // ... actually send ...
            SENT.add(args.recipient_id);
            return { sent: true };
        }
    },
```

Put a human between the decision and the action. For anything that reaches a student or a parent, this is not optional, and Chapter 11 develops it.

8.10 Summary

A language model cannot perform arithmetic that matters. The failure is quiet, plausible, and varies between runs.

The model decides which calculation to perform and how to interpret it. Code performs the arithmetic.

A tool is a description plus a function. The description is a prompt: say what it does, show the argument shape literally, and say when to use it.

Write error messages for the model to read. An informative error lets the agent correct itself inside the loop; an exception or a null does not.

Validation tools are underused. A tool that reports a failed check turns the agent into something that catches its own mistakes.

A tool guarantees the calculation, not the arguments. Require the agent to state its inputs, which makes a wrong argument visible at a glance.

Read tools are safe to retry. Write tools are not. Prefer agents that decide and code that acts.
