PARTICIPANT TEXT · PART 1 OF 3

Agentic AI for Teaching & Academic Work

Part 1 — Foundations

Chapter 1 What an agent is

Chapter 2 How a language model actually works

Chapter 3 The failures you will meet

Chapter 4 Prompting as engineering

Faculty Development Workshop

Symbiosis Institute of Technology, Pune

Facilitator: Mr. Dinesh Wadhwani

10–11 September 2026

Preface

Who this book is for

You teach engineering. You have probably used ChatGPT or Gemini to draft something, and it worked well enough that you kept using it, and badly enough that you stopped trusting it for anything important. That combination is the correct response to the technology as most people encounter it.

This book is about the next step. Not better prompting, though we will cover that properly. The step where the model stops being something you talk to and becomes a component in a system you build, with its capabilities used deliberately and its weaknesses designed around.

The examples are drawn from academic work because that is where you will use this. Question papers, attainment calculations, rubric grading, student doubts, accreditation documentation. Not because these are the most impressive applications, but because they are the ones sitting on your desk.

What this book assumes

That you can read code. Not that you write JavaScript daily, but that a for-loop and a function call do not slow you down.

That you know what an algorithm is, and are therefore capable of being properly suspicious when told that something has no algorithm inside it.

That you have taught long enough to have opinions about assessment.

Nothing else. No machine learning background is needed. Chapter 2 builds the necessary understanding of how these models work from first principles, and it does so at a level of rigour appropriate to people who teach computing rather than people who read popular science.

How this book is organised

Part 1, Foundations, covers what an agent is, how a language model actually works, the specific ways it fails, and prompting as an engineering discipline. Chapter 2 is the load-bearing chapter. Everything after it makes more sense.

Part 2, Building, covers the platform mechanics, structured output, agent architecture and tool use. This is where you write code.

Part 3, Judgement, covers multi-agent systems, grounding, guardrails, evaluation, ethics and institutional adoption. This is the part that determines whether anything you build survives contact with your department.

The companion volume, the *Lab Manual and Test Set*, contains the exercises. This book explains why; that book explains how. They are meant to be used together, but this one stands alone and is written to be read afterwards as well as during.

A note on tone

This book will tell you, repeatedly, that the technology is unreliable in specific ways. That is not scepticism for its own sake. Every failure described here is one you will meet, and the entire value of understanding the machinery is that you can then design around it rather than being surprised by it.

The most useful thing an engineer can know about a component is its failure mode. That is as true here as it is anywhere else.

Chapter 1 — What an agent is

1.1 A question about your Tuesday afternoon

Consider something you did last week.

You needed a set of questions for an internal assessment. You opened a chat window, described the topic, and asked for five questions. What came back was serviceable. You edited two of them, discarded one, wrote a replacement yourself, added the marks allocation, worked out which course outcome each question mapped to, and pasted the result into your department's template.

Total time saved, perhaps twenty minutes. Real, and worth having.

Now consider what happened to that work. The prompt is gone. The judgements you made while editing, which questions were too easy, which topic was out of syllabus, what your students find hard about normalisation, went nowhere. Next month you will do the whole thing again from the beginning.

This is the observation the entire field is built on, and it is worth stating precisely.

When you use a chatbot, you are the runtime.

You decide what to ask. You supply the material. You evaluate the output, decide whether it is good enough, and determine the next step. The intelligence sits in the model. The *process* sits in you, and it is not written down anywhere, so it cannot be repeated, inspected, improved or handed to a colleague.

Everything in this book is an answer to one question: how much of that process can move out of your head and into something that runs on its own?

1.2 Three levels, and why people confuse them

There are three distinct things people call AI tools, and conflating them causes an enormous amount of wasted effort. The distinction is not about how clever the system is. It is about who decides what happens next.

The chatbot

You are in the loop for every step. You ask, you read, you decide, you ask again. The system has no memory of last week and no ability to act. It produces text and stops.

This is not a criticism. For exploratory work, for thinking out loud, for one-off drafting, this is exactly right and anything more elaborate is overhead. A great deal of useful academic work will always happen this way.

The workflow

Here the steps are decided in advance, by you, at design time. Read the marks file. Compute the percentage for each student. Compare against the threshold. Generate a report. Send it.

The system runs without you. It does the same thing every time. Given the same input it produces the same output, which means it can be tested once and then trusted. When it fails, it fails loudly and in a specific place.

A workflow may contain an AI step. Drafting the report text is a reasonable job for a language model. But the *sequence* is fixed and you fixed it.

The agent

Here the system decides the sequence at runtime.

You give it a set of tools and a goal. It reasons about which tool to use, uses it, looks at what came back, and reasons again. It stops when it judges the goal achieved. You did not specify the order of operations because you could not, because the right order depends on what it finds along the way.

This is genuinely different in kind, and it is worth being precise about what the difference is.

> THE DEFINITION TO HOLD ON TO
>
> An **agent** is a system where a model chooses an action, takes it, observes the result, and decides what to do next, without you in the loop for each step.
>
> The distinguishing feature is not intelligence, autonomy, or the presence of a language model. It is **who chooses the next step**.

1.3 Two things that are often mistaken for agency

Because the word "agent" now carries commercial weight, a great deal of software is described as agentic when it is not. Two mistakes recur, and both are worth inoculating yourself against.

A schedule is not autonomy

Something that runs every Friday at five o'clock without being told is not thereby an agent. A cron job that runs a backup script at two in the morning is not an agent. Your operating system's update checker is not an agent.

When a process starts has nothing to do with **who decides what it does**. A scheduled workflow is a workflow that you are not present for. It executes the identical sequence every time. The only thing the scheduler contributes is the decision to begin.

This confusion is common and consequential, because a scheduled workflow is often exactly what you want, and building it as an agent instead makes it slower, more expensive and less predictable for no benefit.

A branch is not a decision

Consider a system that reads attendance and sends different messages depending on the percentage. Below seventy-five per cent, a gentle note. Below sixty, a firmer one. Below fifty, an urgent one.

It feels like the system is deciding. It is not. You decided, when you wrote those three thresholds. The system is evaluating an if-statement that you authored. The fact that a language model generates the prose inside each branch does not change the control flow, any more than a printf statement makes a C program intelligent.

The test is straightforward. Could you draw the flowchart in advance, completely, without knowing the data? If yes, it is a workflow. Every path was determined at design time.

1.4 The comparison that matters

	Chatbot	Workflow	Agent
Who decides the steps	You, live	You, at design time	The model, at runtime
Runs without you	No	Yes	Yes
Same input, same output	Approximately	Exactly	No
Handles the unforeseen	You handle it	It breaks	Sometimes
API calls per run	One per exchange	One per AI step	Three to ten, unpredictable
Cost to build	Nothing	Low	Moderate
Cost to trust	You check each answer	Test once, then rely on it	**Ongoing supervision**

The last row is the one that decides whether your project survives, and it is almost always omitted from discussions of this technology.

A workflow can be tested. You construct a set of inputs with known correct outputs, you run them, you confirm the results, and from then on you have a component you can rely on. The testing cost is paid once.

An agent cannot be tested in that way, because it does not do the same thing twice. You can test that it behaves sensibly across a range of cases, which is worth doing, but you cannot establish that it will behave sensibly on the next case. Its correctness is a statistical property, not a logical one. The supervision cost is recurring, and it is paid by you, every time, for as long as the system runs.

That is not a reason to avoid agents. It is a reason to be certain you need one.

1.5 The question to ask before building anything

> **Do the steps change depending on what the system finds?**

If no, build a workflow. It will be faster to build, cheaper to run, easier to debug, and you will be able to trust it.

If yes, an agent may be justified. Then ask a second question:

> **Am I prepared to check its output every single time?**

If no, do not deploy it. A workflow you trust is worth more than an agent you do not.

Applying this to academic work

Task	Steps change?	Build as
Compute CO attainment from a marks sheet	No. Fixed formula.	Workflow
Generate a question paper to a fixed blueprint	No. Fixed structure.	Workflow with an AI step
Send attendance alerts above a threshold	No. Fixed rule.	Scheduled workflow
Grade essays against a rubric	Barely. Depends on the answer.	Workflow with an AI step, plus review
Decide *which* students need parent contact this week	Yes. Depends on history, trajectory, prior contact.	Agent
Answer a student's doubt on your course	Yes. Depends entirely on the question.	Agent
Investigate why a cohort underperformed	Yes. Each finding suggests the next check.	Agent
Draft NBA documentation from scattered records	Yes. Depends what records exist.	Agent

Four of eight are workflows. In real departmental use the ratio is more lopsided still, because the recurring tasks that consume your time are recurring precisely because they are procedural.

The industry talks about agents because agents are interesting. Your Friday afternoon is saved by workflows.

1.6 The attendance example, developed

The distinction becomes concrete when you look at a single case carefully. This example recurs throughout the book, so it is worth setting up properly.

A colleague wants to email parents when a student's attendance falls below seventy-five per cent. The sheet is in Google Drive. It should run weekly.

The workflow version

Read the sheet. For each student, compute attendance as classes attended divided by classes held. If below seventy-five, bucket by severity. Draft a message. Write it out for review.

Five steps, in that order, every week. This is a workflow, and it is a perfectly good solution. It will run reliably for years.

Where it falls short

Now look at three students, all sitting at roughly the same number today.

Student A is at 72.5 per cent. A month ago she was at 91. She is falling sharply and something has changed. A parent should hear about this now.

Student B is at 70 per cent. A month ago he was at 55. He is recovering, and doing so substantially. A warning letter here would land on a student who is already fixing the problem, and it would be worse than sending nothing.

Student C is at 67.5 per cent, where she has been for a month. Her parents were contacted five days ago. A fourth email is not communication, it is noise. What is needed is a conversation with the class mentor.

The workflow treats all three identically, and it must, because attendance percentage is the only thing it looks at. You could extend it: add a rule about the previous month's figure, add a rule about days since last contact, add a rule about repeat cases. Each rule is a decision you make in advance about a situation you are imagining.

Eventually you have twenty rules and you no longer understand their interactions, which is the familiar fate of any sufficiently elaborated rule engine.

The agent version

Give the system three tools. One returns current attendance. One returns a student's history and the date a parent was last contacted. One computes the number of days between two dates.

Then give it a goal: decide, for each student below threshold, whether to contact the parent, escalate to the mentor, or do nothing, and record a reason for each.

Now something interesting happens. The system looks at the current figures, notices that several students are below threshold, decides it needs their history to distinguish between them, goes and gets it, notices that one was contacted recently, checks how recently, and reaches three different conclusions with three different reasons.

Nobody wrote the rule about recovering students. The instruction said that a student who is improving should probably not receive a warning, which is a principle rather than a threshold, and the system applied it to a case it had not seen before.

That is the difference. Not the schedule, not the branching, but the fact that deciding what to look up is itself part of the task.

1.7 What you give up

The agent version is better in the ways just described, and worse in ways that must be stated.

It costs five to ten times as much per run, because the reasoning loop makes multiple model calls where the workflow made one.

It is slower, taking tens of seconds where the workflow took two.

It does not produce identical output on identical input, which means that if you run it twice you may get two different decisions about the same student. This is unsettling and it is real.

It can loop indefinitely if misconfigured, calling a tool, dissatisfied with the result, calling it again.

And most importantly, it can be confidently wrong in ways a workflow cannot. A workflow that fails to read the sheet throws an error and stops. An agent that fails to read the sheet may proceed to reason about students it has invented, and present its conclusions in the same measured tone it would use for real ones.

Chapters 2 and 3 explain why that last failure is intrinsic to the technology rather than a defect that will be fixed in the next version. Understanding that properly is the difference between building something you can defend and building something that will embarrass you eighteen months from now.

1.8 Summary

An agent is a system in which a model chooses its own next action, using tools you provided, until it judges the task complete.

A schedule does not make something an agent. Neither does branching on a threshold. The distinguishing property is who decides the sequence, and when.

Most academic tasks are workflows. Build them as workflows. Reach for an agent when deciding what to look at is itself part of the problem, and only when you are prepared to supervise the result.

The cost of an agent is not the API bill. It is that its correctness is statistical rather than logical, so the supervision never ends.

Chapter 2 — How a language model actually works

2.1 Why this chapter exists

You could use everything in this book without reading this chapter. Many people do, and they build systems that work most of the time and fail in ways they find mysterious.

The mystery is unnecessary. The failures of language models are not arbitrary. They follow directly from the architecture, and once you understand the architecture, the failures become predictable, which means they become designable-around.

This chapter is written for people who teach computing. It is not a survey of deep learning, and it does not require any. It builds the specific understanding needed to answer four questions:

- Why can a model that writes flawless prose about statistics not reliably average sixty numbers?
- Why does it invent things, and why can it not tell you when it is doing so?
- Why does the same prompt produce different answers?
- Why does telling it to be accurate help, but not solve the problem?

2.2 The one-sentence description

A language model is a function that takes a sequence of text and returns a probability distribution over what the next small piece of text will be.

That is the whole thing. Everything else, the fluency, the apparent reasoning, the sense that something is thinking, emerges from applying that function repeatedly.

It is worth sitting with how strange this is. There is no representation of a fact anywhere in the system. There is no database. There is no arithmetic unit. There is no module that decides whether it knows something. There is a very large function, and a loop.

2.3 Tokens

The model does not operate on characters or on words. It operates on **tokens**, which are word fragments produced by a compression scheme applied to a large corpus of text.

Common words are usually a single token. Rarer words split into several. The sentence

```
The relation is not in third normal form.
```

might tokenise as

```
The | relation | is | not | in | third | normal | form | .
```

whereas an unusual technical term might split as

```
Boyce | -C | odd
```

and a number might split in ways that have nothing to do with its mathematical structure:

```
1247  ->  12 | 47
5891  ->  58 | 91
```

That last detail matters enormously and we will return to it.

A model has a vocabulary of tokens, typically in the range of tens of thousands to a few hundred thousand. Every input is converted to a sequence of token identifiers, and every output is produced one token at a time.

2.4 The prediction loop

Here is what happens when you send a prompt.

Your text is tokenised. The token sequence is fed through the network. The output is a vector with one number for every token in the vocabulary, converted into a probability distribution. One token is selected from that distribution. It is appended to the sequence. The whole extended sequence is fed through again. Another token is produced. And so on, until a stopping token appears or a length limit is hit.

In pseudocode:

```
sequence = tokenise(prompt)

repeat:
    distribution = model(sequence)          # one probability per vocabulary token
    next_token   = sample(distribution)     # pick one
    sequence     = sequence + next_token
until next_token == END or length exceeded

return detokenise(sequence minus prompt)
```

Three consequences follow immediately, and each explains a behaviour you have already observed.

It has no plan. When the model emits the first token of its answer it has not decided what the answer is. There is no outline held in reserve. Each token is chosen given everything so far, and the answer takes shape as it is produced. This is why asking a model to "think step by step" genuinely helps: the intermediate tokens become part of the

sequence, and subsequent tokens are conditioned on them. The reasoning is not a report of a decision made earlier. The reasoning is the mechanism.

It cannot revise. Once a token is emitted it is part of the input for everything that follows. There is no backspace. A model that begins a sentence badly must continue from where it is. This is one reason output quality degrades when a response starts down a wrong path.

It is a distribution, not a lookup. Asked for the capital of France, the token for "Paris" carries overwhelming probability, and you will get Paris every time. Asked for something less determined, several tokens carry meaningful probability, and which one you get depends on the sampling. There is no distinction inside the model between these two cases. Both are the same operation on the same function. Only the shape of the distribution differs.

2.5 Sampling and temperature

If you always took the highest-probability token, the model would be deterministic. Some systems do this, and the result is repetitive and often oddly flat, because natural text is not the most probable text at every position.

Instead a token is sampled from the distribution, and a parameter called **temperature** controls how much the distribution is flattened before sampling.

At temperature near zero, sampling collapses towards always taking the most probable token. Output becomes near-deterministic and conservative. At higher temperatures, lower-probability tokens get a realistic chance, and output becomes more varied and more surprising.

This is the direct explanation for a behaviour that frustrates people. You run the same prompt twice and get different answers. Nothing is broken. You sampled twice from the same distribution.

For academic work the practical guidance falls out cleanly.

Task	Temperature	Why
Generating question variants	Higher, 0.7 to 1.0	You want variety. Repetition is the failure.
Grading against a rubric	Low, 0.0 to 0.3	You want consistency. Variety is the failure.
Extracting structured data	Very low	There is one right answer.
Drafting feedback prose	Moderate, 0.4 to 0.6	Some warmth, but not unpredictable.

Notice that low temperature reduces variation but does not eliminate the underlying problem. A low-temperature model is consistently sampling from the same distribution,

and if that distribution places substantial probability on a wrong answer, you will get the wrong answer consistently. Consistency and correctness are different properties.

2.6 Context

Everything the model knows about your specific situation is in the token sequence you sent. There is no other channel.

The **context window** is the maximum sequence length the model can process, measured in tokens. Modern models have large windows, hundreds of thousands of tokens, which is enough for a substantial amount of course material.

Two properties of the context window are worth understanding.

It is stateless. Each call is independent. When you have a multi-turn conversation, the entire conversation is re-sent every time. The model is not remembering. The interface is resending. This is why, in Chapter 7, memory in an agent turns out to be nothing more elaborate than an array you append to and paste back in.

Position matters. Material at the beginning and end of a long context tends to be used more reliably than material buried in the middle. This is a measured effect and it has a practical implication: when you supply reference material, put your instructions after it, not before, so the instructions sit near the end where they carry most weight.

2.7 Where the knowledge lives

The model was trained by being shown an enormous quantity of text and adjusted, repeatedly, to predict the next token better. What it retains from that process is stored in its parameters as statistical structure.

This has a particular character that is easy to mis-describe.

It is not a database. There is no row containing the fact that BCNF is stricter than 3NF. Instead, the parameters encode that in contexts about normal forms, tokens describing BCNF as more restrictive are highly probable. When the model produces that claim it is not retrieving a fact. It is generating the most probable continuation, and the continuation happens to be true because the training text was largely true about this topic.

This distinction sounds academic and is not. It has three concrete consequences.

Well-represented material is reliable. Anything discussed extensively and consistently in the training data comes out correct nearly always, because the probability mass is concentrated on the correct continuation. Standard normalisation theory is in this category. So is basic circuit analysis, standard algorithms, and the content of every widely-used textbook.

Sparsely-represented material is unreliable in a specific way. For an obscure topic, the distribution is diffuse. Several continuations have similar probability, one of which may be correct. You will get a fluent answer either way, and the fluency is not evidence, because fluency is produced by the same mechanism regardless of whether the content is right.

Absent material produces confident invention. This is the important case. If the training data contained nothing about the specific example you used in class on the fourteenth of August, there is no correct continuation available. But the function still returns a distribution, and something still gets sampled. The model produces a plausible-sounding example about third normal form, because plausible-sounding examples about third normal form are what the surrounding context makes probable.

Nothing in the architecture distinguishes this case from the first one. There is no confidence signal, no flag, no internal state that means "I am making this up." The probabilities are just the probabilities.

> **THE CONSEQUENCE TO REMEMBER**
>
> The model has no mechanism for representing the difference between knowing and not knowing.
>
> It is not being dishonest when it invents your worked example. It has no faculty with which to be honest about this.

2.8 Why arithmetic fails

Now we can answer the question that matters most for academic use, and it deserves a full treatment because it is the failure that will cost you.

Ask a model to average sixty numbers and it will give you a number. That number will be wrong, plausibly wrong, and differently wrong each time you ask.

Three separate mechanisms conspire here.

There is no arithmetic

Nowhere in a transformer is there a unit that adds. Adding is not an operation the architecture performs. What the architecture performs is next-token prediction.

When a model correctly answers "what is 12 plus 7", it is not computing. It is producing the most probable continuation of a sequence that looks like an addition question, and because "12 + 7 = 19" and similar strings appear abundantly in training data, the probability mass sits firmly on "19". It has learned the *appearance* of arithmetic on small numbers, which happens to be indistinguishable from arithmetic on small numbers.

Scale up and the illusion breaks. There are far too many possible sums of sixty two-digit numbers for any of them to have been memorised. The model must generalise, and what it generalises is the *form* of an answer, not the value.

Tokenisation destroys numerical structure

Recall that numbers split into tokens by textual frequency, not by place value. A four-digit number may become two tokens split at an arbitrary point. Column-wise addition, the algorithm you were taught, requires access to individual digits aligned by position. The model does not reliably have that. It has fragments.

The output is sampled

Even where the distribution favours the correct answer, sampling means a nearby wrong answer can be drawn. This is why repeated queries give different numbers, and why the numbers cluster near the truth without landing on it.

The shape of the failure

This is the part to internalise, because it determines how you must design.

The failure is not loud. There is no error, no warning, no hedging. The model does not say it is unsure. What you get is a clean, specific, confident figure.

And it is usually *close*. Ask for an attainment percentage where the truth is 55 and you will get something in the sixties or low seventies. Not 400. Not zero. A number that looks entirely reasonable and passes any sanity check you would apply while skimming.

Consider what that means in an academic context specifically.

A wrong attainment figure does not look wrong. It enters a CO–PO matrix. The matrix goes into a self-assessment report. The report goes to an accreditation panel. Nobody recalculates it by hand, because avoiding that recalculation was the entire reason for automating it. By the time anyone questions it, it has been in three documents for eight months and the raw data may no longer be available.

> **THE RULE THIS PRODUCES**
>
> Never let a language model perform arithmetic whose result matters.
>
> This is not a limitation to be worked around with better prompting. There is no prompt that installs an arithmetic unit. The solution is structural: the model decides *which* calculation to perform, and code performs it. Chapter 8 develops this properly.

2.9 Four other failure mechanisms

Arithmetic is the clearest case but not the only one. Each of the following follows from the same architecture.

Fluency is independent of correctness

The mechanism that produces well-formed sentences is the mechanism that produces content. They are not separate stages. There is no grammar module and a separate truth module.

This means fluency carries no information about accuracy, and your instinct to treat confident prose as more reliable than hesitant prose is actively misleading here. In human writing, hedging often signals genuine uncertainty. In model output it signals that hedging language was probable in that context.

This has an uncomfortable corollary that Chapter 9 addresses in detail. A model grading student work rewards fluent writing over correct content, because fluency is what it is best at recognising.

It follows the shape of your question

If you ask "why is the third normal form the most important normal form", the model will produce reasons, because the question presupposes the claim and reasons are the probable continuation. It will not usually challenge the premise.

Ask instead "is the third normal form the most important normal form, and if not, what is the actual trade-off" and you get something better. The framing is part of the input, and the input determines the distribution.

This is not agreeableness in a psychological sense. It is that a question containing a premise makes continuations consistent with that premise more probable.

It drifts across a batch

Grade five submissions in one conversation and the standard shifts. The fifth is evaluated in a context containing the previous four, which changes the distribution.

This is not a bug to be prompted away. It is a direct consequence of conditioning on the full sequence. The design response is to grade each submission in a fresh context, and to have a second pass check consistency across the batch. Chapter 9 builds this.

It cannot audit itself

Asking "are you sure?" does not consult a confidence estimate, because there is no confidence estimate to consult. It samples a continuation of a sequence that now contains a challenge, and continuations of challenges are frequently either capitulation or reassurance depending on context.

A model that revises its answer when questioned has not detected an error. A model that maintains its answer has not verified anything. Neither response is evidence.

2.10 What actually improves reliability

Given the above, the levers that genuinely work are limited and specific. It is worth naming them now, since the rest of the book is largely about applying them.

Put the material in the context. The model does not know your syllabus. Supplying it converts a question about the world, where the distribution is diffuse, into a question about a document in front of it, where it is sharp. This is grounding, and it is the single most effective intervention available. Chapter 11.

Move computation into code. Anything that must be exactly right is computed by a function, not predicted by a model. The model's job is choosing which function and supplying the arguments. Chapter 8.

Constrain the output shape. Asking for a specific JSON structure narrows the distribution considerably and makes the result checkable by machine. Chapter 6.

State what is out of scope. Telling the model explicitly what is *not* in your course is more effective than hoping it stays in scope, because it makes refusal a probable continuation where otherwise invention would be. Chapter 4.

Use a second pass. A separate call, with instructions to find fault rather than to agree, catches a meaningful fraction of errors. Not because the second model is better, but because it is conditioned differently. Chapter 9.

Lower the temperature where consistency matters. Partial, but real.

Notice what is absent from this list. There is no instruction that makes the model reliable at arithmetic. There is no phrasing that makes it aware of its own uncertainty. There is no prompt that makes it know what you taught. Those are architectural limits, and pretending otherwise is how systems fail in production.

2.11 Summary

A language model computes a probability distribution over the next token, and text is generated by sampling from that distribution repeatedly.

It contains no facts, no database, no arithmetic unit, and no representation of its own uncertainty. Its apparent knowledge is statistical structure over training text, which makes it reliable on well-represented material, unreliable on sparse material, and confidently inventive on absent material, with nothing internally distinguishing the three.

Arithmetic fails because there is no arithmetic, because tokenisation destroys place value, and because output is sampled. It fails quietly, with plausible numbers, which is the most dangerous possible failure mode for accreditation work.

Fluency and correctness are produced by the same mechanism and are therefore uncorrelated.

The interventions that work are grounding, moving computation to code, constraining output, declaring what is out of scope, adding a review pass, and lowering temperature. Everything else is decoration.

THE SENTENCE TO CARRY FORWARD

It is fluent before it is correct.

Fluency and correctness are separate properties of the output, and this technology delivers the first far more reliably than the second. Your judgement is not being replaced by these systems. It is being relocated, from producing material to checking it.

That is still a substantial saving. It is a different job, and designing as though it were the same job is how institutions get burned.

Chapter 3 — The failures you will meet

3.1 Why failures deserve a chapter

Chapter 2 explained the machinery. This chapter catalogues what goes wrong in practice, and it does so in a particular way: each failure is named, its mechanism traced back to the architecture, its appearance described, and the design response given.

The reason for this format is that these failures are not rare edge cases. Every one of them will happen to you within a month of using this technology seriously. Recognising a failure by name, and knowing immediately why it happened, is the difference between a fifteen-minute fix and an afternoon of confusion.

There is also a professional reason. When a colleague brings you a failure, and they will, the useful response is not "AI is unreliable." It is "that is confident invention on absent material, here is why it happened, and here is the instruction that prevents it."

3.2 Confident invention

What it looks like. You ask about something the model does not know. You receive a complete, fluent, specific answer that is wholly fabricated. A worked example you never used. A paper that does not exist. A student who is not in your class.

The mechanism. From 2.7. The model has no representation of not knowing. Given a context about your course, the probable continuation is content about your course, and something gets sampled whether or not any of it corresponds to anything real.

Why it is the most dangerous failure. Because the invention is contextually appropriate. It will use your terminology, match your notation, sit at the right difficulty level, and be indistinguishable from correct output by any means short of checking it.

The academic case that should worry you. A student asks the doubt agent about a topic outside your syllabus. The agent explains it, in your voice, grounded in nothing. The student now believes something you did not teach and cannot tell that you did not teach it, because the tool is on your course page with your name on it.

Design response. Three things, in order of effectiveness.

Supply the material and instruct the model to answer only from it. This is grounding, and it converts the diffuse case into the sharp case.

State explicitly what is out of scope. A prompt that lists the topics not covered makes refusal a probable continuation. This single line does more work than any other in a course-facing prompt.

Give an exact refusal string. Not "say you don't know" but the specific sentence to output, because a specified string is far more reliably produced than a described behaviour.

```
If the answer is not in the material provided, respond with
exactly: "That is not covered in our course material. Please
check with your instructor."
Do not answer from general knowledge.
```

3.3 Plausible arithmetic

What it looks like. A specific number, confidently stated, close to correct, different on each run.

The mechanism. From 2.8. No arithmetic unit, tokenisation that destroys place value, sampled output.

How to detect it. Run the same query three times. If the number changes, it was never a calculation. This is a five-second test and you should apply it to any number a model gives you.

Design response. Compute in code. The model chooses which calculation and supplies the arguments; a function does the arithmetic and returns the result. Chapter 8.

A second-order failure that catches people. Moving arithmetic to a tool guarantees the calculation is right. It does not guarantee the model passed the correct arguments. An agent may correctly call `calculate_attainment` while passing the wrong question columns. The maths is flawless and the answer is wrong.

The response is to require the model to state its inputs before its result:

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

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

3.4 Instruction decay

What it looks like. Your prompt contains eight rules. The first three are followed reliably, the last five intermittently. Or the model follows all eight for the first three items in a batch and drifts thereafter.

The mechanism. Instructions are tokens in the sequence, competing with every other token for influence on the distribution. They are not enforced by any mechanism. A long list of constraints dilutes each one. And material in the middle of a long context is used less reliably than material near the ends.

Design response. Fewer rules, more prominently placed. If a rule matters, it goes near the end of the prompt, not buried in a list of twelve. If you have twelve rules, you

probably have three rules and nine preferences, and the preferences are diluting the rules.

Where a constraint is critical, enforce it in code rather than asking. A validation function that returns `{ valid: false, message: "Total is 22, expected 20" }` is a guarantee. An instruction saying "marks must total 20" is a request.

3.5 Format drift

What it looks like. You ask for JSON. You get JSON wrapped in markdown code fences. Or JSON preceded by "Here is the JSON you requested:". Or JSON with an extra field you did not ask for, or a field renamed to something more natural.

The mechanism. In the training corpus, JSON appears inside code fences overwhelmingly often. The fence is the probable continuation. Telling the model not to produce it competes against that prior and does not always win.

Design response. Stop fighting it and handle it. Strip the fences before parsing, locate the outermost brace pair, and parse that.

```
function extractJson(raw) {
    const cleaned = raw.replace(/```json/gi, '').replace(/```/g, '').trim();
    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, and it eliminates the most common cause of a failed run. Wrap it in a try/catch inside your loop, so one malformed response flags itself rather than destroying the whole batch.

3.6 Fluency bias

What it looks like. In grading, a confidently written wrong answer scores higher than a correct answer written awkwardly.

The mechanism. From 2.9. The model's core competence is recognising well-formed text. Assessing technical correctness is a much harder discrimination, and where the two signals conflict, the stronger signal wins.

Why this one is not merely a technical problem. In an Indian engineering classroom, a large proportion of students are working in their second or third language. Fluency correlates with prior educational advantage. Automating a fluency bias means applying an existing inequity uniformly, at scale, with the appearance of objectivity, which is considerably worse than applying it inconsistently by hand.

Design response. Three things, and none is sufficient alone.

Separate content from expression in the rubric, and score them independently. If expression carries marks, it should carry them explicitly and in a stated proportion.

Add a review pass whose instructions specifically name this failure:

```
Check whether fluent or confident writing has been rewarded over
correct content, and whether correct content has been penalised
for poor expression, grammar, or non-native phrasing.
```

Hand-check the bottom decile. Every time. The students most affected by this failure are the least able to contest it.

3.7 Batch drift

What it looks like. The tenth item in a batch is judged against a different standard than the first.

The mechanism. Each item is processed in a context containing the previous items. The distribution shifts as the sequence grows.

Design response. Process each item in a fresh context. Then add a consistency pass over the collected results, which is a different question and a legitimate one for a model: "here are ten gradings, is the standard consistent, and if not which are outliers?"

3.8 Premise adoption

What it looks like. You ask a leading question and get a confirming answer.

The mechanism. The premise is part of the input and shapes the distribution.

Design response. Ask neutrally. Compare:

> "*Why is my second question too difficult?*"
>
> "*Assess the difficulty of each question against the stated Bloom's level. If a question is mismatched, say which and why. If all are appropriate, say so.*"

The second gives the model somewhere to go other than agreement. This matters more than it appears when you are using a model to review your own work, which is one of its better uses.

3.9 The self-verification illusion

What it looks like. You ask "are you sure?" and it either revises or reassures. Either feels like verification.

The mechanism. There is no confidence state to consult. The challenge becomes part of the sequence and a continuation is sampled.

Design response. Do not ask a model to verify itself in the same context. Verification requires either a fresh context with adversarial instructions, or code, or you.

The distinction between these is worth being clear about. A second model call with instructions to find fault is genuinely useful, because it is conditioned differently and does not have the first answer's reasoning in its context. Asking the same conversation "are you certain?" is not, because the answer being questioned is right there in the sequence, making consistency with it probable.

3.10 Runaway iteration

What it looks like. An agent calls a tool, is unsatisfied, calls it again, and again. Your daily quota vanishes in minutes.

The mechanism. The loop terminates when the model produces an answer rather than a tool call. If the instructions do not make stopping attractive, and the tool results are ambiguous, the probable continuation remains a tool call indefinitely.

Design response. Two things.

A hard iteration cap in code. Not a request in the prompt, a `for` loop with an upper bound. Five is right for everything in this book.

An explicit stopping instruction, because agents also fail in the opposite direction, burning iterations on reassurance:

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

3.11 Silent partial failure

What it looks like. A tool returns an error or empty data. The agent proceeds anyway, reasoning from nothing, and presents conclusions in its ordinary tone.

The mechanism. A tool error is just text in the context. Nothing forces the model to treat it as terminal.

Why this is the failure that produces fabricated accreditation figures. The tool fails, the agent estimates, the estimate looks reasonable, and there is no indication anywhere in the output that the underlying data was never read.

Design response. Instruct explicitly, and make error messages informative.

```
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.
```

And write your tools' errors for the model to read. A tool that returns `{ error: "questions must be a non-empty array, e.g. [\"q2\",\"q4\"]" }` gives the agent something to correct. A tool that throws, or returns `null`, gives it nothing, and it will improvise.

3.12 A diagnostic table

Keep this. It converts a symptom into a cause and a fix.

Symptom	Cause	Fix
Specific fact you cannot verify	Confident invention (3.2)	Ground it; state out-of-scope; exact refusal string
Number changes between runs	Plausible arithmetic (3.3)	Compute in code
Number stable but wrong	Wrong arguments to a correct tool	Require it to state its inputs
Later rules ignored	Instruction decay (3.4)	Fewer rules; move critical ones to the end; validate in code
`JSON.parse` throws	Format drift (3.5)	Fence-stripping parser
Well-written wrong answer scores high	Fluency bias (3.6)	Separate content from expression; review pass; hand-check the bottom
Standard shifts across a batch	Batch drift (3.7)	Fresh context per item; consistency pass
Agrees with your framing	Premise adoption (3.8)	Ask neutrally
"Are you sure?" changes nothing useful	Self-verification illusion (3.9)	Fresh adversarial context, or code, or you
Quota gone in minutes	Runaway iteration (3.10)	Hard cap of 5; explicit stopping instruction
Confident output from failed tools	Silent partial failure (3.11)	Instruct to report failures; informative tool errors

3.13 Summary

Every failure in this chapter follows from the architecture in Chapter 2. None is a defect awaiting a patch.

The recurring shape is that failures are quiet. The model does not throw exceptions. It produces well-formed output that is wrong, in the same register it uses when it is right.

Therefore your design must assume the failures rather than hoping to prevent them. Ground the model, compute in code, constrain the output, cap the loop, instruct it to report failures honestly, and add a review pass.

Chapter 4 — Prompting as engineering

4.1 The claim of this chapter

Prompting is often presented as a collection of tricks. It is better understood as specification writing.

Chapter 2 established that the model's output is a distribution conditioned on the input. It follows that the input is the only thing you control, and that changing it changes the distribution. A prompt is therefore not a request. It is the specification of a region of output space, and a vague prompt specifies a large region.

That reframing has a practical consequence. When output is unsatisfactory, the instinct is to edit the output. The correct move is to edit the prompt and regenerate. Editing output helps once. Editing the prompt helps every time you use it for the rest of the semester.

4.2 A realistic starting point

Here is a prompt close to what most faculty actually type.

```
Generate 5 questions on Database Normalization for my exam.
```

You will get five questions. They will look plausible. They will also, typically:

- Sit at the same cognitive level, usually recall
- Carry no marks allocation
- Map to no course outcome
- Overlap substantially with one another
- Include at least one topic you do not teach
- Be pitched at whatever difficulty the model inferred from nothing

None of this is a failure of the model. The prompt specified almost nothing, so the distribution was wide, and what came out was near its centre.

THE DIAGNOSIS

Vague prompts do not produce wrong output. They produce **average** output.

For teaching, average is the harder problem, because wrong output gets caught and average output gets used.

4.3 The five components

A working prompt specifies five things. If a prompt is underperforming, one of them is missing, and in practice it is almost always the fourth or fifth.

	Component	The question it answers
R	Role	Who is the model being?
C	Context	What material must it work from?
T	Task	What exactly must it produce?
F	Format	What shape must the output take?
G	Guardrails	What must it never do?

We will build the question-paper prompt through all five, and the point of doing it in stages is that you can see what each contributes.

R — Role

Role sets vocabulary, assumed audience, and depth. It works because it conditions the distribution towards text produced in that setting.

```
You are an experienced professor of Computer Engineering setting
internal assessment papers for a second-year B.Tech course at an
Indian engineering institute affiliated to a state university.
```

Specificity is doing the work here. "You are a teacher" changes little. The version above pins down difficulty, terminology, assessment conventions and regional context simultaneously, because text written in that setting differs systematically from text written for a different one.

C — Context

Context is the material the model must work from, and it is what separates a generic question generator from *your* question generator.

```
COURSE: Database Management Systems (CS204), Semester 4

SYLLABUS UNIT 3 — Normalization:
Functional dependencies; Armstrong's axioms; closure of attribute
sets; 1NF, 2NF, 3NF, BCNF; lossless-join and dependency-preserving
decomposition; multivalued dependencies and 4NF.

COURSE OUTCOMES:
CO3: Apply normalization techniques to eliminate redundancy and
     anomalies in relational schema design.
CO4: Analyse a given relational schema and decompose it while
     preserving dependencies.

NOT TAUGHT IN THIS COURSE: 5NF, domain-key normal form,
temporal databases.
```

That final line deserves attention out of proportion to its length.

The model was trained on the entirety of the internet's database material, which covers a great deal more than your syllabus. Without constraint, output drifts towards the centre of that larger distribution, and you get a question on fifth normal form on a paper for students who have never heard of it.

Stating what is out of scope makes refusal a probable continuation where otherwise invention would be. It is more effective than any amount of instruction to "stay within the syllabus", because it converts a general instruction into a specific list.

Every prompt you write for a course should carry this line.

T — Task

The task should be specific enough that two different people reading it would produce comparable output.

```
TASK:
Generate a 20-mark internal assessment covering CO3 and CO4.
- 2 questions at 3 marks (Understand level)
- 2 questions at 4 marks (Apply level)
- 1 question at 6 marks (Analyse level)

At least one Apply-level question and the Analyse-level question
must give a concrete relation with sample functional dependencies
for the student to work on. Do not ask students to merely define
or list.
```

Note that the marks structure is stated rather than implied, and that the requirement for a concrete relation is stated rather than hoped for. Anything you leave implicit, the model fills in from the training distribution.

F — Format

Under-specified format is the commonest reason output is technically fine and practically unusable. You want to paste it into a template, not spend twenty minutes reformatting it.

```
OUTPUT FORMAT:
A Markdown table with exactly these columns:
Q.No | Question | Marks | CO | Bloom's Level | Expected Answer Outline

After the table, add a section headed "Marking Scheme" with a
mark-by-mark breakdown for each question.

Do not include any preamble, commentary, or explanation of your
approach. Output the table and the marking scheme only.
```

That last sentence saves roughly ninety seconds on every single run, forever.

G — Guardrails

Guardrails specify what the model must refuse. They matter because without them the model always produces *something*, and "something" is exactly the failure you cannot detect by skimming.

```
CONSTRAINTS:
- Use only topics listed in the syllabus section above. If asked
  for a topic not listed, state clearly that it is out of syllabus
  and generate nothing for it.
- Tag every question with the CO it assesses. If a question does
  not clearly map to a listed CO, discard it and generate another.
- Bloom's level tags must match the verb actually used in the
  question. Do not label a "define" question as Analyse.
- Marks must total exactly 20.
```

Demonstrating that guardrails work

This is worth doing yourself, because the result is more convincing than the argument.

Take the prompt with Role, Context, Task and Format but no Constraints. Then ask:

```
Also add a question on 5NF.
```

It will almost certainly produce one, despite 5NF appearing in your NOT TAUGHT list, because nothing instructed it to refuse.

Now add the Constraints block and ask again. It refuses.

The difference is one paragraph, and that paragraph is the difference between a tool that quietly puts out-of-syllabus questions on your paper and one that does not.

4.4 Four techniques that earn their keep

Provide one worked exemplar

A single example of the output you want shapes results more powerfully than three paragraphs describing it. This is because an example is a direct sample from the region of output space you are specifying, whereas a description is an indirect gesture at it.

```
EXAMPLE OF THE STYLE REQUIRED:

Q. Consider the relation R(A, B, C, D, E) with functional
dependencies: A -> BC, CD -> E, B -> D, E -> A.
Determine whether R is in BCNF. If not, decompose it into BCNF
and state whether your decomposition is dependency-preserving.
[6 marks] [CO4] [Analyse]

Generate questions of comparable specificity and rigour.
```

This is the fastest available fix for output that is technically correct but pitched wrongly.

Require justification

Add:

```
For each question, add one sentence explaining why you assigned
that CO and that Bloom's level.
```

Two effects, and the second is the important one.

It marginally improves the tagging, because the justification tokens become part of the sequence and condition what follows, which is the same mechanism that makes step-by-step reasoning work.

More usefully, **it makes errors visible**. A wrong CO tag inside a table is nearly invisible; you would have to check each one against the question. A wrong justification is obvious in one second, because it describes recall while the tag says Analyse.

You have converted a checking task from expensive to cheap. This is the single most valuable habit in this chapter, and it recurs in Chapter 8 as a safety mechanism for agents.

Anchor Bloom's levels to verbs

Models inflate Bloom's levels. Everything becomes Analyse, because Analyse appears in contexts describing sophisticated assessment and sophistication is a probable continuation of a request for good questions.

The specific error is treating topic difficulty as cognitive level. Asking a student to *define* BCNF is Remember, however advanced BCNF is as a topic.

Anchor it explicitly:

```
BLOOM'S LEVEL DEFINITIONS FOR TAGGING:
Remember   - define, list, state, name
Understand - explain, describe, illustrate, classify
Apply      - solve, compute, demonstrate, normalize a given relation
Analyse    - compare, differentiate, decompose, examine trade-offs
Evaluate   - justify, critique, assess, defend a design choice
Create     - design, construct, formulate a new schema

Tag based on the verb the question actually requires the student
to perform, not on the topic's difficulty.
```

The final sentence is the one that corrects the error.

Iterate on the prompt, not the output

Stated already, repeated because it is the habit that separates people who use this technology from people who build with it.

When output is wrong, resist editing it. Change the prompt and regenerate. You are debugging a specification, and a specification that produces good output once will produce good output for the rest of the semester.

4.5 Where prompts should live

A practical point that has more effect on your working life than any technique above.

A prompt written into a chat window is gone when you close the tab. A prompt saved as a reusable asset with your course material attached becomes an instrument you use in one line.

Two mechanisms are available.

A saved assistant. Gemini's Gems, and equivalents elsewhere, hold standing instructions plus attached knowledge files. Your syllabus, CO list and previous papers live in the assistant. Generating a paper becomes a single sentence: "Generate a 20-mark internal assessment for Unit 3 covering CO3 and CO4." No context needed, because the context is permanent.

An input field in code. When you build the Actors in Part 2, the prompt goes in the input schema rather than being hardcoded. This matters for a mundane and important reason: changing an input takes ten seconds, changing code and rebuilding takes two minutes. Since prompt iteration is most of what you will do, that ratio determines whether the work is pleasant or unbearable.

The general principle: **anything you will tune often should be data, not code.**

4.6 What prompting cannot do

Necessary honesty, since Part 2 builds on this foundation.

It cannot make arithmetic reliable. No prompt installs an arithmetic unit. Chapter 8.

It cannot make the model know your material. No role-setting makes it aware of the example you used on the fourteenth of August. Chapter 11.

It cannot make output consistent. Same prompt, three runs, three variants. For question generation this is desirable. For grading it is a problem, and Chapter 9 addresses it structurally rather than through prompting.

It cannot make refusal certain. Guardrails substantially reduce confident invention. They do not eliminate it. Verification remains your job.

Recognising these four limits is what separates a prompt from a system. The rest of this book is about the systems.

4.7 Summary

A prompt is a specification, and it specifies a region of output space. Vague prompts produce average output, which is a worse problem for teaching than wrong output.

Five components: Role, Context, Task, Format, Guardrails. Format and Guardrails are the ones usually missing.

The line stating what is *not* in your course does more work than any other single line in an academic prompt.

Requiring justification converts error-checking from expensive to cheap, and is the highest-value habit in this chapter.

Prompts should live somewhere reusable, as a saved assistant or as data in an input field, never only in a chat window.

Prompting cannot fix arithmetic, material knowledge, consistency or guaranteed refusal. Those need structure, which is Part 2.
