Lesson Review and Index: Product Management Lessons 1 and 2

This index covers the two reviewed lessons, the two capstone case studies, and the recommended learning order. It links each topic to the guide that teaches it in depth. These lessons did not cover RAG, Replit or Nginx in depth, so those topics have their own guides rather than being forced in here.

1. Lesson 1: AI product costing and an n8n onboarding workflow

- **Define the job** of the AI feature: tone, length, audience. Send useful variables with each request.
- **Unit cost:** calculate cost from input and output tokens, then multiply by expected volume. Compare conservative, realistic and aggressive scenarios before deciding to charge, give a free allowance, or absorb the cost. *Guide: Tokens, context and cost.*
- **Define input, output, owner and cost** before using an LLM in any feature.
- **Employee onboarding:** an HR form captures hire ID, candidate name, email, department, role, manager and start date. Reference data sits in an access registry and a scheduling registry.
- **Three agents:** a Coordinator routes and tracks work; an Access agent decides and records access; a Scheduling agent handles meetings. *Guide: n8n multi-agent onboarding.*
- **Webhooks:** a doorbell. The workflow waits for an event instead of checking repeatedly. Use the test URL while building, then the production URL once active. Include a unique `hire_id` as the primary key.
- **Credentials and tools:** never connect a personal account to a practice automation. Consider permissions before connecting any tool. Build and test one small step at a time.

2. Lesson 2: Make, MCP and case-study thinking

- **Scenarios** are repeatable automations. Variables and settings let one flow serve many inputs. Check credit use, notifications and failure handling before you enable one. *Guide: Make.com.*
- **MCP** (Model Context Protocol) is a standard way for AI applications to connect to outside systems. A **tool** is a capability an agent can call. *Guide: Skills, subagents and agents.*
- **Repeated assignment expectations:** understand the user and the real problem first; define the goal and a success metric; split features into v1, v2 and cut; define the core loop, AI

strategy and feedback; create journey maps, a PRD and three key wireframes; measure events on a dashboard.

3. Capstone: Fireflies, meeting commitments

Part	Content
Problem	Commitments made in meetings show up in summaries but are not reliably assigned, owned, tracked or closed. The class figure: only 28 percent of extracted commitments reached a resolved state within 14 days.
User	Meeting attendees who owe or receive a follow-up.
Goal	Make every commitment visible, owned and resolvable, without becoming a full project-management tool.
Resolution states	A commitment is resolved only when it is **done**, **reassigned**, or **deliberately dropped**. Each state is explicit.
Success metric	Share of commitments resolved within 14 days (the baseline is 28 percent).
v1	Detect commitments from a transcript; show owner and due date; three resolution states.
v2	Reminders; reassignment suggestions; owner summaries.
Cut	Full project boards, task dependencies, time tracking.
Core loop	Meeting → commitment extracted → owner confirms → reminder → resolved state recorded.
AI strategy	Extraction with a confidence score; the owner always confirms before a commitment is saved.
Feedback design	One-tap confirm, edit or reject on each extracted commitment, so the model learns what matters to each team.
Boundaries	No full task management; no changes to the meeting platform itself.
Events	commitment_detected, owner_confirmed, reminder_sent, resolved_done, resolved_reassigned, resolved_dropped
Wireframe 1	Meeting summary with a list of detected commitments, each with owner and due date, and confirm, edit or reject buttons.
Wireframe 2	My commitments: list grouped by due date, with the three resolution states as clear buttons.
Wireframe 3	Weekly summary: open, resolved and dropped counts, plus a link to any item that is overdue.

4. Capstone: ixigo, the confirmed-seat problem

Part	Content
Problem	A traveller knows the origin and destination but has no confirmed seat on the preferred train or class.
User segments	Last-minute travellers; price-sensitive planners; travellers who must arrive by a fixed time.
Goal	Help the user discover, compare and choose viable alternatives, and understand each option honestly.
Decision factors	Price, travel time, arrival time, station convenience, confirmation probability and cancellation flexibility.
Uncertainty messaging	Show likelihood as a range, never a guarantee. Say what is unknown and why.
v1	Compare alternative trains and classes for the same route and date; show the six decision factors; show uncertainty.
v2	Nearby stations and dates; bus and flight options where data is available.
Cut	Redesigning rail inventory, timetables, quota rules, fares, payments, refunds, or support staffing.
Success metric	Share of users who choose an option and complete the journey without leaving to search again.
Core loop	Search → see viable options → compare on the six factors → choose → track the decision.
AI strategy	Rank options and explain trade-offs in plain language; never invent availability.
Feedback design	After each choice, ask whether the option was right, and log the reason.
Events	search_started, option_viewed, option_chosen, uncertainty_shown, option_changed_later
Wireframe 1	Results list with six factor columns and a clear confidence range per option.
Wireframe 2	Option detail: trade-offs in plain language, and cancellation terms.
Wireframe 3	Confirmation: choice saved, with a reminder of what remains uncertain.

5. The nine-step case-study framework

1. Problem framing
2. User research

3. Goal
4. Success metric
5. Scope (v1, v2, cut)
6. User flow and journey map
7. PRD
8. Wireframes
9. Metrics and event taxonomy (plus an optional experiment plan)

6. Recommended learning order

1. Tokens, context and cost (LLM basics and cost).
2. RAG and retrieval.
3. Skills, subagents and agents.
4. Webhooks, JSON and test versus production.
5. One n8n workflow with fake data.
6. Credentials, permissions and test accounts.
7. Make scenarios and MCP.
8. App building with Replit or Lovable.
9. One capstone, using the nine-step framework.

7. Trusted next steps

- Build one safe n8n webhook demo with fake data and a practice account.
- Learn MCP from its introduction, then the specification when you build.
- For Make, learn scenarios first, then MCP with the smallest permissions.
- Review any connector's permissions before you connect it.

Official sources: check current documentation for n8n, Make, Lovable, Replit and MCP before relying on a feature. Prices change; check the live pricing page on the day you decide.

Tokens, Context Windows and Cost

This guide is self-contained. You do not need the class video. Every term is explained the first time it appears, every number is worked out step by step, and you can check each calculation with a calculator. It answers four questions: what a token is, how much a request costs, how to keep the context window from filling up, and how to choose a model and settings that are fast, cheap and good enough.

> **Running example.** We use one food-seller app throughout. A seller in Mumbai types a dish name, category and price, and the app writes a short, appetising description. Each button click is one API call.

1. What a token is

An LLM does not read letters or words. It reads **tokens**: chunks of text that the model's tokenizer has learned to split out. A token can be a whole short word (`dish`), part of a long word (`Separ` + `ating`), a space plus a word, or punctuation. Rule of thumb for English: **one token is roughly four characters, or about three-quarters of a word.**

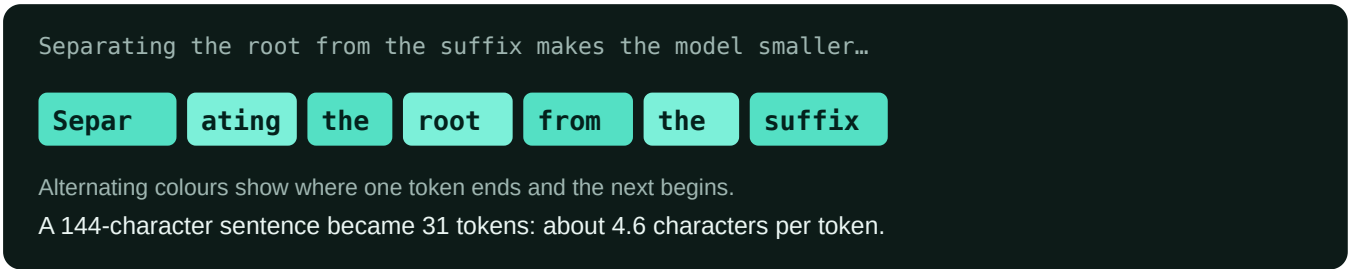


Figure 1. Tokens are not words. Common pieces are single tokens; rarer words split into several pieces. The example sentence (144 characters) is 31 tokens in one class tokenizer.

Why rare words cost more

The tokenizer was trained on a large text corpus. Frequent words such as *the* or *product* are single tokens. Rare words, names, code and non-English text break into more pieces, so they cost more tokens per idea. This is why an Indian-language or technical prompt can cost more than the same meaning in plain English. Always measure, never guess.

2. Input tokens and output tokens

Every API call has two sides, and each side is billed separately:

- **Input tokens** are everything you send: the system prompt (standing instructions), the user message, any examples, and any retrieved documents.
- **Output tokens** are everything the model writes back. Output tokens are usually priced higher than input tokens, and they are generated one by one, so they also take longer.

Total tokens per call = input tokens + output tokens. That total is what the usage report shows, and what the cost is based on.

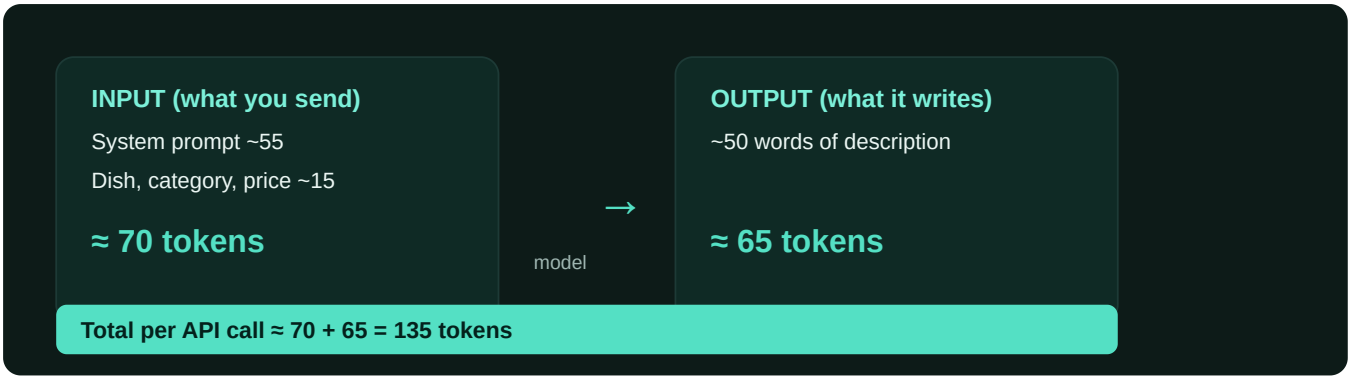


Figure 2. The seller-app call. Input is about 70 tokens, output about 65, total about 135. The system prompt is the part that repeats on every click.

3. Counting tokens before you ship

Measure first. Never estimate. Three ways to count:

1. **Count the prompt in the provider's console or token-counting endpoint.** This gives the exact input count before you pay for a call.
2. **Read the usage block in each response.** Every API response reports input and output tokens. Log them.
3. **In Claude Code or Claude Desktop, check the context usage panel** (see section 5). It shows how much of the window your session is using.

> **Common mistake:** counting only the words you typed. The system prompt, earlier turns in the conversation, examples and tool definitions are all input, and they are resent on every call.

4. Pricing, with a worked cost per call

Prices change, so the numbers below use the class example: **Claude Haiku at $0.80 per million input tokens and $4 per million output tokens.** Before you publish any cost, check the current rate on Anthropic's pricing page: anthropic.com/pricing. Use the formula, and replace the rates with today's rates.

Formula: cost per call = (input tokens ÷ 1,000,000 × input rate) + (output tokens ÷ 1,000,000 × output rate)

Worked example: one seller-app call

Step	Calculation	Result
Input cost	70 ÷ 1,000,000 × $0.80	$0.000056
Output cost	65 ÷ 1,000,000 × $4.00	$0.000260
Total per call	$0.000056 + $0.000260	**$0.000316**

So one generation costs about **$0.0003**, which is roughly 3 hundredths of a cent. The cost looks trivial, but it multiplies. Output is about 82% of the cost here, even though it is under half the tokens. That is why output length is the first thing to control.

From one call to a monthly bill

Multiply the cost per call by the number of calls. Use three scenarios so the business survives the bad case, not only the average one:

Scenario	Generations per dish	Example: 200 sellers × 15 dishes	Monthly cost (USD)
Conservative	1.2	3,000 dishes × 1.2 = 3,600 calls	3,600 × $0.000316 ≈ $1.14
Realistic	1.8	3,000 × 1.8 = 5,400 calls	5,400 × $0.000316 ≈ $1.71
Aggressive	2.8 (cap 3)	3,000 × 2.8 = 8,400 calls	8,400 × $0.000316 ≈ $2.65

The table shows the arithmetic, not a forecast. Use your own seller count and rates. Add 2 new dishes per seller per month to the dish count each month. The rule to take away: **a cost is only meaningful once it is multiplied by volume, and volume grows over time.** Decide whether to charge for generations, give a free allowance, or absorb the cost, using the conservative case as your floor.

Worked example: a multi-call feature ("Find my matches")

One click on a job-matching product can run 25 model calls. Price each call type separately, because they differ in model and size. The token counts below are **assumptions for teaching**; measure your own before you publish.

Call type	Model	Calls	Input tokens / call	Output tokens / call	Why this model
Query generation	Haiku	1	160	60	Short, structured task
Web search (external API)	Search API	4	0 tokens	0	Billed per search, not per token
Listing extraction	Sonnet	20	1,200	150	Reading messy web pages needs accuracy
Deduplication	Haiku	1	900	80	Simple comparison
Scoring and ranking	Sonnet	1	1,500	200	Judgement across the candidate's profile
Final synthesis	Sonnet	1	1,800	400	Writes the explanations users read

Cost per click, using Haiku at $0.80 / $4 and Sonnet at an *example* $3 / $15 per million (check current rates):

Call type	Input cost	Output cost	Subtotal per call × count
Query generation (Haiku, 1)	160 × $0.80/M = $0.000128	60 × $4/M = $0.000240	$0.000368
Listing extraction (Sonnet, 20)	1,200 × $3/M = $0.003600	150 × $15/M = $0.002250	$0.005850 × 20 = $0.117000
Deduplication (Haiku, 1)	900 × $0.80/M = $0.000720	80 × $4/M = $0.000320	$0.001040
Scoring (Sonnet, 1)	1,500 × $3/M = $0.004500	200 × $15/M = $0.003000	$0.007500
Synthesis (Sonnet, 1)	1,800 × $3/M = $0.005400	400 × $15/M = $0.006000	$0.011400
Total per click			**≈ $0.137 (plus search API fees)**

What this teaches: 20 extraction calls on Sonnet are about 85% of the token cost. The product lever is obvious: move extraction to a cheaper model, or trim the listing text before it reaches the model (see section 7). Notice that the search API is billed separately, so cost per click is tokens *plus* external fees.

5. The context window: your working memory

The **context window** is the total number of tokens the model can hold in one session at once: your messages, its replies, attached files, tool results and the standing instructions. Think of it as the model's desk. When the desk is full, older items must be removed or summarised.

Sizes vary by model and plan. Check the current figure in the model documentation for the model you use. Your own Claude Code usage screen in the class showed a window of about **967k tokens** for that model, so do not assume one fixed number.

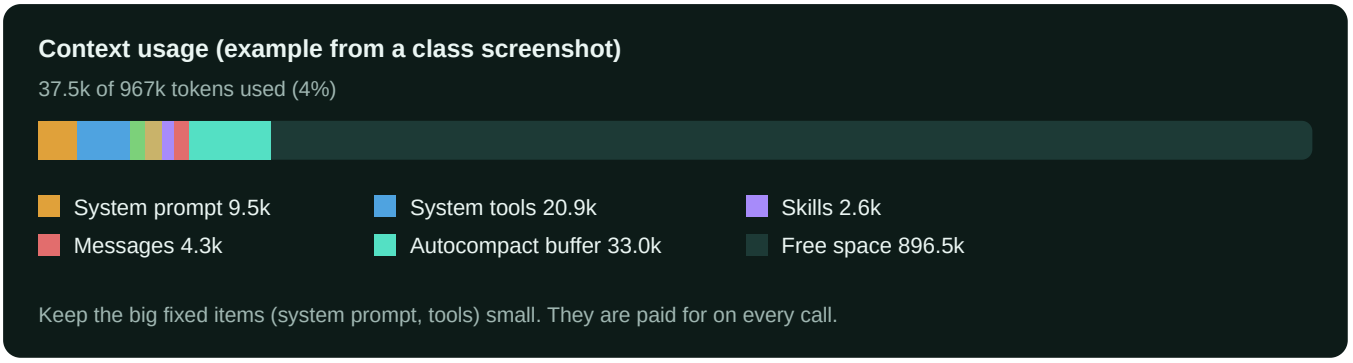


Figure 3. What fills a context window. The autocompact buffer (teal) is space kept back so the tool can summarise older messages before the window overflows.

How to check usage

- **In Claude Code:** type `/context` to see the breakdown: system prompt, tools, memory files, skills, messages and free space.
- **In an API response:** read the usage fields for input and output tokens on every call.
- **In a long chat:** start a fresh session when the topic changes. A new session starts empty, so it costs less per message.

Why replies cost more than prompts. Each reply is generated one token at a time, and it becomes part of the history that is resent with the next message. Output tokens are priced higher, so a long answer costs more than a long question. Ask for the shape you need ("five bullets, under 80 words") to keep replies short.

6. How the model chooses each word: attention and sampling

Attention: not every token gets equal weight

When the model processes the word *some* in "Riya poured tea into the cup and added some...", it does not treat every earlier word equally. It compares the current token with every earlier token, gives each a relevance score, turns the scores into weights that sum to 1, and mixes the earlier words according to those weights. In that sentence, *tea* and *cup* carry strong weight. The output

is a blend of the most relevant earlier tokens, so the same word *some* ends up with a different internal meaning depending on its context.

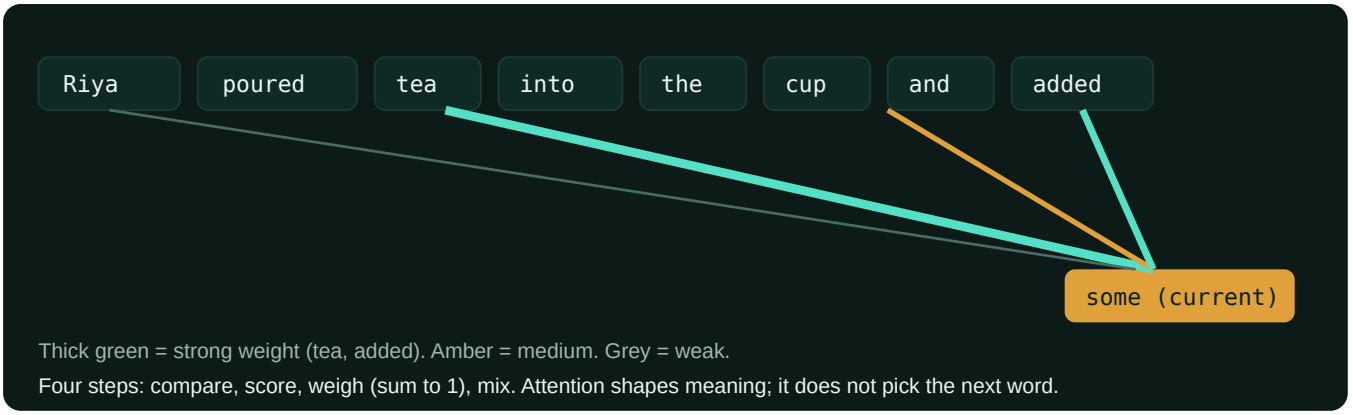


Figure 4. Attention as a weighted mix over earlier tokens. The weights are learned, not hand-written.

Sampling: how the next token is picked

After the model scores every possible next token, a sampling setting decides how the choice is made. These are the four controls you will see in most APIs:

Control	What it does	Use it when	Cost effect
Temperature	Lower = more predictable; higher = more varied. 0 is near-deterministic.	Structured output such as JSON: 0 to 0.3. Creative copy: 0.7 to 1.0.	None directly, but very high values can make output longer and less useful.
Top-k	Only the k most likely next tokens are candidates.	Stop rare, odd words from slipping in.	None directly.
Top-p (nucleus)	Keep the smallest set of tokens whose probabilities add up to p (for example 0.9).	A moderate way to balance variety and reliability.	None directly.
Max tokens	A hard limit on how many output tokens are generated.	Always set it. It caps output cost.	**Direct:** sets the largest possible output bill per call.

> **Rule of thumb for the seller app:** temperature 0.7 for the description, max tokens about 120 (enough for 50 words), and temperature 0 for any JSON the app must parse. Tune on real examples, then lock the values.

7. Using fewer tokens and getting more from each one

1. **Shorten the standing prompt, but keep the structure.** Say what the job is, the audience, the length and the format in one short block. Remove filler such as "you are a very helpful

assistant".

2. **Ask for structured output.** A JSON array of 4 search queries (about 60 output tokens) is cheaper and easier for code to use than three paragraphs of explanation.
3. **Trim large inputs before they reach the model.** Send the fields that matter, not the whole web page. Extract the job title, skills and location first, then pass only those.
4. **Use XML or clear tags to separate parts.** For example `<role>`, `<instructions>`, `<input>`. Tags help the model find each section, which reduces retries, and each retry costs tokens.
5. **Route each task to the smallest model that passes.** Use Haiku for short, structured steps and Sonnet only where judgement matters.
6. **Cap output.** Set max tokens and ask for a length ("under 50 words").
7. **Reuse, do not repeat.** Put standing instructions in a skill or project instructions, so you are not retyping them and the tool loads them efficiently (see the skills guide in this series).
8. **Cache what does not change.** Repeated prompts that share the same long system prompt can be cheaper when the provider supports prompt caching. Check your provider's current caching rules before you rely on it.
9. **Start a fresh session** when the topic changes, instead of carrying a long history into an unrelated question.

8. Choosing a model: fast, cheap or creative

Model family	Best for	Speed	Cost	Quality on hard reasoning
Haiku (small)	Classification, short copy, extraction of simple fields, query generation	Fastest	Lowest	Good for simple, well-defined steps
Sonnet (medium)	Reading messy inputs, scoring, most product features	Medium	Medium	Strong general quality
Opus (large)	Hard analysis, complex planning, high-stakes reasoning	Slowest	Highest	Strongest

Check the current list and prices in the provider documentation, since names and rates change. The decision rule: **start with the smallest model, measure quality on 20 real examples, and move up only where the smaller model fails.** Do not pay for a large model to do a simple job.

9. Limits and guardrails

- **Context limit:** the request fails or gets truncated when the window is full. Keep inputs under the limit with room for the answer.
- **Rate limit:** providers cap requests per minute. Retry with a short wait; do not retry in a tight loop, because each retry is billed.

- **Budget limit:** set a spending cap in the provider console, and log cost per feature so one bug cannot drain the budget.
- **Output limit:** max tokens stops generation early. Make sure the answer is still complete at that limit.

10. Put it into practice

> **Exercise (20 minutes).** Take any prompt you already use. (1) Count its input tokens with your provider's counter. (2) Run it once and read the output count. (3) Work out the cost with the formula in section 4 using current rates. (4) Shorten the system prompt, cap max tokens, and run it again. (5) Record the new totals and the difference in cost. Write one sentence on what you cut and whether quality held.

Quick check (answers are in the next section)

1. A reply is 65 output tokens and the prompt is 70 input tokens. Which costs more per token at the example rates?
2. Why does the system prompt matter even though you type it only once?
3. Which sampling setting caps the cost of a single call?
4. Your JSON output breaks sometimes. Which control do you lower first?

11. Answers and glossary

1. Output, at $4 per million tokens versus $0.80 for input, so output is 5 times the rate. **2.** It is resent on every call, so it is paid for on every click. **3.** Max tokens. **4.** Temperature, down towards 0.

Term	Plain meaning
Token	A small piece of text the model reads and writes, roughly four characters.
Input tokens	Everything sent to the model in one call.
Output tokens	Everything the model writes back.
Context window	The total tokens the model can hold in one session.
Attention	The model's way of weighing earlier tokens by relevance.
Temperature	How predictable or varied the chosen words are.
Max tokens	The hard limit on output length per call.
Autocompact buffer	Space kept free so older messages can be summarised before the window overflows.

12. Trusted sources to read next

- Anthropic pricing: anthropic.com/pricing (check current rates before any cost estimate).
- Anthropic documentation on models, context windows and token counting: docs.anthropic.com.
- Anthropic prompt engineering guide: docs.anthropic.com (search "prompt engineering").

Prices and model names in this guide were checked against the class material and should be re-checked on the date you publish. Token counts marked "assumption" are for teaching; measure your own before relying on them.

RAG: Giving an LLM Your Own Knowledge

This guide takes you from the problem to a working retrieval pipeline. Read it in order. Every concept is explained the first time it appears, and each section ends with a plain-English summary you can reuse. You do not need the class video.

> **Running example.** A company has a 40-page leave policy. An employee asks: *"How does our leave policy work?"* We follow that one question from the document, through the database, to the answer.

1. Three problems with LLMs

A large language model learned from a big public snapshot of text, and then stopped learning. That causes three problems:

Problem	What it looks like	Example
1. It cannot see your private data	It has never read your files.	"How many leave days do I get?" It guesses a typical answer.
2. Knowledge cutoff	Its knowledge stops at a training date.	"What changed in last week's policy?" It cannot know.
3. Hallucination	It answers confidently, and the answer is wrong.	It invents a clause that does not exist.

Three ways to fix these are covered below: fine-tuning, in-context learning and RAG. They are not interchangeable, and choosing the wrong one is the most common mistake.

2. Fine-tuning: retraining the model

Fine-tuning means continuing the model's training on your own examples, so its internal weights change. Think of it as sending the student back to school.

Three types

- **Supervised fine-tuning (SFT):** you give input and ideal output pairs, and the model learns to copy that style and behaviour.

- **Continued pretraining:** the model keeps reading large amounts of new raw text in your domain, for example medical papers.
- **RLHF (reinforcement learning from human feedback):** people rank answers, and the model learns to prefer the ones people rank higher. This is how many chat models are tuned for helpfulness.

How supervised fine-tuning works, step by step

1. **Data collection:** gather hundreds to thousands of clean examples. Quality matters more than quantity.
2. **Method selection:** choose SFT, a parameter-efficient method, or RLHF, based on your goal.
3. **Training:** run the job on a training platform and watch the loss curve.
4. **Evaluation:** test on examples the model never saw, and compare with the base model.

What fine-tuning fixes

Problem	Solved?	Why
Private data	Solved, partly	The model learns patterns from your data, but it can still mix up facts and cannot cite a source.
Recent data	Partly	Every new fact needs another training run.
Hallucination	Partly	Tuning on good answers can reduce invention, but it does not guarantee correct facts.

3. In-context learning: teaching in the prompt

In-context learning means the model learns a pattern from examples inside the prompt, with **no change to its weights**. Nothing is retrained. The learning lasts only for that request.

Few-shot example: sentiment

```
Below are examples of text labelled with their sentiment.
Use the examples to determine the sentiment of the final text.

Text: "I love this phone. It's so smooth." → Sentiment: Positive
Text: "This app crashes a lot."            → Sentiment: Negative
Text: "The camera is amazing."             → Sentiment: Positive
Text: "I hate the battery life."           → Sentiment: ???
```

The model answers **Negative**. It learned the label pattern from three examples, with no training. This is called **few-shot prompting**. Zero examples is zero-shot, and one example is one-shot.

> **Limit:** every example costs input tokens on every call (see the tokens guide). In-context learning is cheap to try and expensive to scale when the examples are long.

4. RAG: retrieval-augmented generation

RAG means: before the model answers, a search step finds the most relevant passages from your documents, and those passages are placed into the prompt. The model then answers using both its general knowledge and the passages you supplied.

Use the open-book exam picture. An LLM with fine-tuning is a student who studied the textbook months ago. An LLM with RAG is a student who is allowed to open the textbook to the right page during the exam.

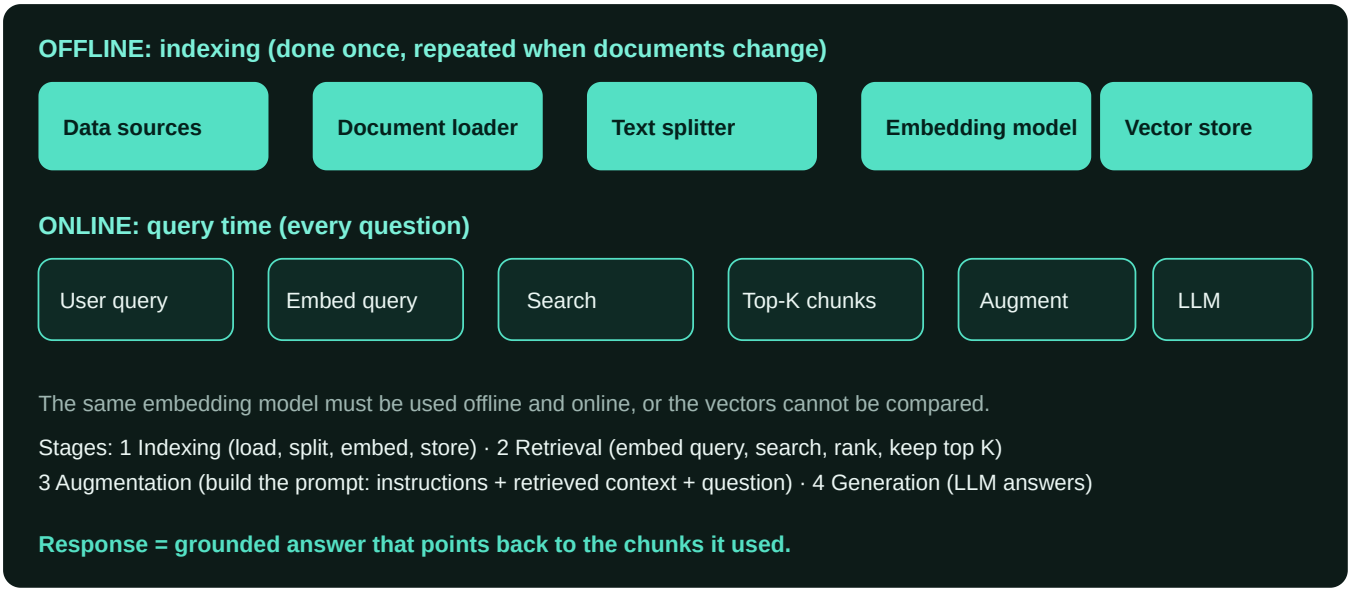


Figure 1. The complete RAG pipeline: an offline indexing half and an online query half.

5. Stage 1: indexing (the offline half)

Step 1: data sources and document loaders

A loader reads each source and turns it into plain text with metadata (file name, page, date, URL). Common sources: PDFs, web pages, YouTube transcripts, cloud storage folders, databases, CSV files. Keep the metadata: you will need it later to filter results and to cite sources.

Step 2: text chunking

A model cannot search a 40-page file as one unit. The text is cut into **chunks**. Each chunk needs to be small enough to be specific, and large enough to hold a complete thought.

Worked example: recursive character splitting

A recursive splitter tries the largest separator first, and only falls back to a smaller one when a piece is still too long:

1. Paragraph break (`\n\n`)
2. Sentence end (`. `)
3. Comma (`, `)
4. Word (space), and as a last resort single characters

Two settings control the result. **Chunk size** is the maximum length of each chunk. **Chunk overlap** is how many characters the next chunk repeats from the end of the previous one, so a sentence cut in half still appears whole in one of the chunks.

The class demo used a 519-character paragraph about machine learning, with chunk size 120 and overlap 15. The first chunk came out as *"Machine learning is a subset of artificial intelligence that focuses on building systems that learn from data"* (about 109 characters). The next chunk started with *"learn from data. Unlike traditional programming..."*, so the words "learn from data" appear in both chunks. Seven chunks came out in total.

Setting	Too small	Too large
Chunk size	Each chunk loses context; answers are fragments.	Search becomes vague; too much irrelevant text goes to the model.
Overlap	Ideas cut at a boundary get lost.	Many duplicate chunks; more storage and cost.

> **Starting point:** chunk size 300 to 800 tokens with 10 to 15 percent overlap for prose policies. Test on 20 real questions and adjust.

Step 3: embeddings: turning meaning into numbers

An **embedding** is a list of numbers (a **vector**, often hundreds or thousands of values) that represents the meaning of a piece of text. Texts with similar meaning get vectors that point in similar directions. An **embedding model** produces these vectors. Common choices include OpenAI's embedding models and open sentence-transformer models, and hosted options such as Google's and Hugging Face's models.

Rule: the same embedding model must be used for the chunks and for the questions. Vectors from different models are not comparable.

Step 4: the vector store

A **vector database** stores each chunk with three things: the chunk text, its embedding vector, and its metadata. It can run locally (for a laptop demo or a small internal tool) or in the cloud (for

shared, large or production systems). Examples include local libraries and managed cloud services; choose one by data size, privacy needs and budget.

6. Stage 2: retrieval (the online half)

Retrieval finds the right context for one question.

1. **Embed the query** with the same embedding model.
2. **Semantic search:** compare the query vector with every chunk vector, usually with **cosine similarity** (how closely two vectors point the same way; 1.0 is identical direction).
3. **Rank** the chunks by similarity.
4. **Keep the top K** chunks. K is the number of chunks you pass on, often 3 to 8.

Retrieval in action

Question: *"How does our leave policy work?"* The vector store holds chunks such as "Office dress code policy", "Annual leave and holiday rules", "Remote work equipment", "Leave policy: sick and casual", and "Employee onboarding steps". Their similarity scores to the question might be 0.91 for annual leave, 0.88 for sick and casual leave, and much lower for the rest. With K = 2, the two leave chunks are passed on. The search matched **meaning**, not the exact words "leave policy", which is why semantic search beats keyword search for questions phrased in different ways.

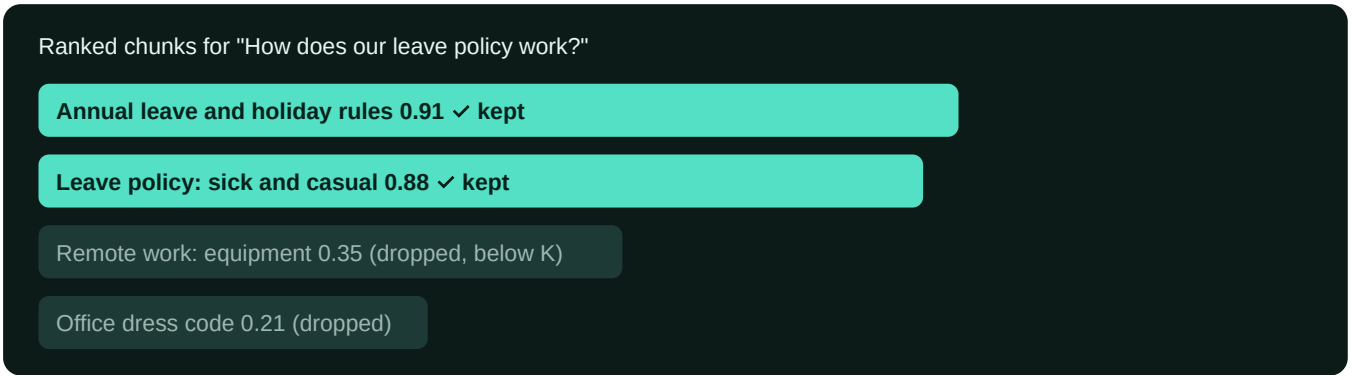


Figure 2. Retrieval keeps the top K. Scores are illustrative.

7. Stage 3: augmentation (building the prompt)

Augmentation is where you combine the instructions, the retrieved text and the question. This is the step that makes RAG work:

```
[System instruction]
You are a helpful assistant. Answer ONLY from the provided context.
If the context is insufficient, say "I don't know."
```

```
[Context]
{retrieved_chunks_text}

[Question]
{user_query}
```

Three rules make this safe. Put the rule about "answer only from context" in the system instruction. Label the context clearly so the model knows what is evidence. Ask the model to name the chunk it used. Without the instruction, the model will fall back to its general knowledge and you will not know which answers came from your documents.

8. Stage 4: generation

The LLM writes the answer from the augmented prompt. Good answers follow three checks: they are supported by the context, they say "I don't know" when the context is missing, and they cite the source. Log the question, the retrieved chunks and the answer, so you can review failures later.

9. How RAG solves the three problems

Problem	How RAG helps	What it does not fix
Private data	Your documents are searched at query time; the model sees them.	Access control: you must still decide who may see which document.
Knowledge cutoff	Re-index a document and the answer updates without retraining.	Re-indexing is still a pipeline you must run and monitor.
Hallucination	The answer is grounded in retrieved text, and you can check it.	Bad retrieval still produces bad answers, so you must measure retrieval.

10. RAG or fine-tuning? Questions and answers

Q11. A formal Hindi chatbot: RAG or fine-tuning?
Start with prompting for tone and language, which is cheap. Use RAG for facts such as policies and prices. Fine-tune only if the style still fails after good prompting and you have enough high-quality examples. The usual answer is RAG plus a clear system prompt, not fine-tuning.

Q12. Can you use RAG and fine-tuning together?
Yes. They do different jobs. Fine-tuning can teach a consistent style or format. RAG supplies

the facts for each question. Many production systems do exactly this, but add fine-tuning last, after RAG is measured.

Q13. RAG for real-time news from the last hour: suitable?
RAG can work only if the index is refreshed quickly enough, with a pipeline that loads and embeds new articles within minutes. Fine-tuning is the wrong tool, because it cannot keep up. Measure the delay between publication and availability in the index.

Q14. RAG over 500 PDFs gives partly correct answers. What is wrong?
Usually retrieval, not the model. Check: are the right chunks in the top K? If not, the chunks may be too large or too small, K may be too low, the PDFs may be scanned images with bad text extraction, or the metadata filter may be too narrow. Fix retrieval first; the model cannot answer from chunks it never received.

Q15. RAG works in English but fails in Hindi. Why?
The embedding model is probably not strong in Hindi, so Hindi questions and Hindi chunks land far apart in vector space. Use a multilingual embedding model, index the Hindi text in the same space, and test with Hindi questions. Check tokenization too: Hindi text often uses more tokens for the same meaning, which affects chunk size.

11. A test plan you can follow

1. Write 20 real questions and the answer you expect, with the source page.
2. For each question, log the top K chunks. Is the right source in the top K? (This is **retrieval accuracy**.)
3. Check whether the final answer is supported by the chunks. (This is **faithfulness**.)
4. Ask 3 questions the documents cannot answer. The correct response is "I don't know".
5. Change one setting at a time (chunk size, overlap, K, embedding model) and re-run the 20 questions.

12. Quick check

1. Which of the four RAG stages happens offline?
2. Why must the same embedding model be used for chunks and queries?
3. Name two things you should store with each chunk.
4. An answer is wrong, but the right chunk was in the top K. Is retrieval the problem?

Answers: 1. Indexing (load, split, embed, store). 2. Vectors from different models cannot be compared. 3. Its text and its metadata, and its vector. 4. No; the chunk was retrieved, so the problem is the prompt or the model's use of it.

13. Glossary and further reading

Term	Plain meaning
RAG	Search your documents first, then answer with the results in the prompt.
Chunk	A piece of a document small enough to search and send.
Embedding	A list of numbers that represents meaning.
Vector store	A database that stores chunks, their vectors and metadata.
Cosine similarity	How closely two vectors point the same way.
Top K	How many best-matching chunks are passed to the model.
Few-shot	Giving examples in the prompt so the model copies the pattern.
Fine-tuning	Retraining the model's weights on your examples.
Hallucination	A confident answer that is not supported by facts.

- Anthropic documentation on prompting and long-context use: docs.anthropic.com.
- Embedding model and vector database documentation from your chosen provider; check current options before you pick one.

Skills, Subagents and AI Agents

Three words get mixed up constantly: a **skill**, a **subagent** and an **AI agent**. This guide defines each one, shows how to decide which to build, and walks through examples from six industries. It also covers the case most teams miss: **neither**. Every example is self-contained and you do not need the class video.

1. The three definitions in one sentence each

Term	One-sentence definition	Analogy
Skill	A saved set of instructions, examples and sometimes scripts that the model loads when a task matches, so you do not retype them.	A recipe card in your kitchen drawer.
Subagent	A helper model run inside a larger task, with its own separate context, usually for one focused job, and it returns a result to the main conversation.	A specialist you hand one question to, who comes back with an answer.
AI agent	A model in a loop that decides its next step, calls tools (search, read a file, send an email), looks at the result, and repeats until the goal is met or it must stop.	A junior employee who can take actions and must be supervised.

2. The key difference

- **A skill is knowledge you reuse.** It does not act on its own. You, or the model, trigger it, and it shapes the answer. It costs tokens only when loaded.
- **A subagent is a worker you delegate to.** It runs a separate task with its own context window, so the main conversation stays clean. It is useful when a job would fill the main context.
- **An agent is a decision-maker that acts.** It chooses actions, calls tools, and loops. It is the most powerful and the most risky, because each step can change something in the real world.

> **Rule of thumb:** if it only needs the right instructions, it is a **skill**. If it needs to do one bounded job on its own, it is a **subagent**. If it must choose its own steps and use tools toward a goal, it is an **agent**. If none of those fits, it is **neither**, and a plain prompt or a normal script is the better answer.

3. The decision flow

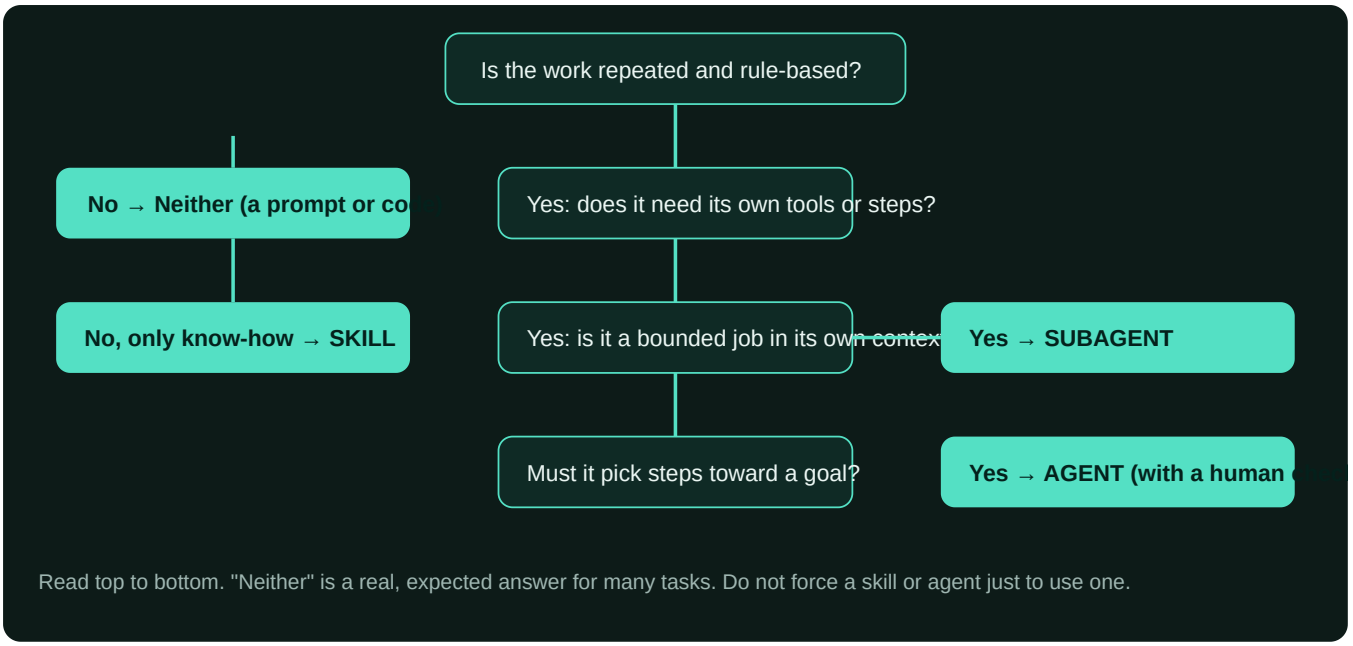


Figure 1. Decision flow. Start with the simplest option and move right or down only when you must.

4. How to create a skill (step by step)

A skill is a folder with one main file, `SKILL.md`. The file starts with a short header that says what the skill is for and when to use it, followed by the instructions.

1. **Pick one repeated task.** Example: "write a product requirements document in our house style". If you do it weekly, it is a skill candidate.
2. **Write the header.** Name, one-line description, and when to use it. The description is how the model decides to load it, so be specific.
3. **Write the instructions.** Role, steps, the required output sections in order, the tone, and what to avoid.
4. **Add two or three examples.** One good input and its ideal output. Examples do more than long rules.
5. **Add a checklist.** The final check before the answer is returned, such as "all required sections present, no invented numbers".
6. **Test on five real inputs**, including one difficult case. Fix the instructions, not the output, so the fix repeats.
7. **Save it** in your skills area (for Claude, under the skills settings) and invoke it by name or by matching request.

```
---
name: prd-copilot
description: Helps a product manager draft a PRD section by section. Use when the
```

```
user asks for a PRD, requirements or product spec.
---
# Role
You are a PRD co-pilot. Draft one section at a time, ask clarifying questions where
depth is missing, and flag gaps before the document is finished.

# Sections, in this order
1. Problem statement (3-5 sentences, no bullet points)
2. Pain points (each tied to a user segment and evidence)
3. Goals and success metrics
4. Scope and non-goals
5. User flow
6. Requirements
7. Risks and open questions

# Checklist before you answer
- Every section present and in order
- No invented numbers; mark assumptions clearly
```

The skill runs only when needed, so it costs tokens on those requests only. It replaces the long prompt you would otherwise paste every time, which is the main token saving.

5. How to create a subagent

A subagent is a named helper with its own description, instructions and allowed tools. Create one when the job is self-contained and would clutter the main conversation, for example "read 40 pages of contracts and list every clause that mentions liability".

1. **Name it for the job**, for example `contract-liability-reviewer`.
2. **Write what it receives and what it returns.** Input: one contract. Output: a table with clause, page and risk level.
3. **Limit its tools.** A reviewer needs read access only. Do not give it write or send permissions.
4. **Ask the main conversation to delegate** that job, and to use only the returned table.

6. How to create an AI agent

An agent needs four things: a goal, a set of tools, a decision rule, and a stopping rule. This is the most important part of the guide.

1. **Define the goal in one sentence**, with a measurable finish, for example "reply to a support ticket with an approved answer, or hand it to a human within 2 minutes".
2. **List the tools, with the least permission each one needs.** Read the knowledge base: yes. Send an email: only after human approval.

3. **Write the decision rule.** "If the ticket mentions a refund over the policy limit, do not answer; route to a human."
4. **Write the stopping rule.** Maximum steps (for example 6), maximum spend, and what counts as done.
5. **Add a human approval gate** before any action that is hard to undo: sending money, sending external messages, changing access, deleting data.
6. **Log every step** (input, tool called, result, decision) so you can explain and debug each run.
7. **Test with fake data first,** then a dedicated test account, then a limited live pilot.

> **Safety for agents:** never give an agent your personal email, your main drive or real customer data while you are learning. Use a test account, minimum permissions and approval before anything external.

7. Worked examples across six industries

Each example shows the right choice and why. The labels "Skill", "Subagent", "Agent" and "Neither" follow the decision flow.

Example A: Retail (e-commerce product descriptions)

Task	Choice	Why
Write a product description in the brand voice	Skill	Same tone and format every time; no tools needed.
Check stock and price, then update the listing	Agent	Chooses steps, reads inventory, writes to the store, needs approval before publishing.
Count the words in a description	Neither	A simple script does this exactly and cheaply.

Example B: Banking and fintech (loan document checks)

Task	Choice	Why
Explain a loan term in plain language	Skill	Fixed style and definitions; no action taken.
Read 80 pages of a loan file and list missing documents	Subagent	A bounded job that would fill the main context.
Chase missing documents, update the file and notify the customer	Agent (human approval before the message)	Multi-step, uses tools, changes records and contacts customers.
Calculate EMI from a rate and tenure	Neither	An exact formula belongs in code, not in a model.

Example C: Healthcare (appointment support)

Task	Choice	Why
Draft a patient-friendly reminder in a fixed, approved style	Skill	Repeated wording, reviewed once by staff.
Summarise a 30-page discharge summary into a checklist	Subagent	Single focused job; reads, returns a table.
Book, move or cancel appointments	Agent (staff approval for cancellations)	Chooses slots and changes a live calendar.
Decide whether a symptom is urgent	Neither (route to a clinician)	Clinical judgement must stay with a qualified person; an AI must not decide alone.

Example D: Education (course and assessment)

Task	Choice	Why
Generate quiz questions in the course format	Skill	Same format each time; the quiz generator skill is one example.
Grade 200 short answers against a rubric	Subagent (one per batch)	Bounded batches keep each context small.
Build a personal study plan from a learner's progress	Agent	Reads progress, picks next lessons, schedules reminders.
Assign a final grade	Neither (teacher decides)	High-stakes judgement belongs to a person.

Example E: Logistics (shipment exceptions)

Task	Choice	Why
Write the standard delay message in the company tone	Skill	Fixed template and tone.
Read carrier tracking logs and find the failed scan	Subagent	Bounded read-and-report job.
Detect a late shipment, reroute it and notify the buyer	Agent (human approves rerouting cost)	Multi-step with money and external messages.
Compute the distance between two warehouses	Neither	A map or distance API is exact.

Example F: Public sector (citizen service requests)

Task	Choice	Why
Explain an application form in simple language	Skill	Approved wording; no action.
Read a request and list which department it belongs to	Subagent	One clear classification job.
Route, log and send status updates across departments	Agent (officer approval for decisions)	Tools, records and communication; a person owns each decision.
Decide eligibility for a benefit	Neither (rule engine plus officer)	Eligibility rules must be auditable and deterministic.

8. Build a skill and an agent: a full walkthrough

The Support Ops case below is the exercise to practise on. Use it to build one skill and one agent, each working on its own.

> **The problem.** A SaaS company handles support through one shared inbox. Five capabilities are needed: (1) read, categorise and route new tickets within minutes, escalating outages and legal or security keywords immediately; (2) make every outbound reply match the company tone and never state a refund or price that is not in the current policy; (3) for an at-risk customer, combine billing, usage logs and past tickets into a one-page briefing before a retention call; (4) grade a weekly sample of closed tickets against a quality rubric and summarise recurring failure patterns; (5) when a customer reports a bug, check the changelog for a known issue and, if none, write it up for engineering with repro steps.

Step 1: map each capability

Capability	Choice	Justification
1. Categorise and route tickets	Agent	Chooses a team from several tools and must escalate immediately; the action (routing) is repeated and changes the workflow. Keep escalation rules in code where possible.
2. Tone-matched, policy-safe replies	Skill (with a check)	Same tone and rules each time; no action taken. The policy check should read the current policy file, not a memorised number.
3. At-risk account briefing	Subagent	A bounded job across three systems that would fill the main context; returns one page.

4. Weekly quality grading	Subagent (batch) plus a skill for the rubric	Batch grading is bounded; the rubric is reusable knowledge.
5. Bug triage against the changelog	Agent, human review before engineering	Searches the changelog, decides whether it is new, and writes a report; a person checks before filing.

Note: no capability needed "neither" here, but step 2's policy number and step 1's escalation rule should be code or data, not model memory. That is the "neither" part of the design.

Step 2: build the skill (reply tone and policy)

```
---
name: support-reply
description: Drafts a customer support reply in the company tone. Use for any
outbound reply to a customer.
---
# Role
You write support replies for a SaaS company. Warm, direct, no jargon.

# Rules
- Never state a refund, discount or price that is not in the policy text provided in
the request.
- If the policy text is missing, write "I'll check this and get back to you" and flag
it.
- Maximum 120 words. One clear next step at the end.

# Examples
Input: "Can I get a refund for last month?" + policy: 14-day refunds
Output: "Thanks for reaching out. Refunds are available within 14 days of purchase,
and yours falls outside that window, so I can't offer one. I'd be glad to help you
with an upgrade or a credit instead."

# Checklist
- Policy figure matches the provided policy text
- No invented numbers
- Under 120 words
```

Test it: give it five real tickets, two with the policy text missing, one asking for a price that is not in the policy. Pass means: no invented figure and a flag where information is missing.

Step 3: build the agent (ticket routing)

1. **Goal:** route each new ticket to one team within 5 minutes, or escalate at once.
2. **Tools:** read the ticket (read only); read the team list (read only); write the route to the ticketing tool (write, but only the route field). No email access.

3. **Decision rule:** if the ticket contains an outage or a legal or security keyword from the escalation list, escalate immediately to the on-call person and stop. Otherwise choose the team that matches the topic.
4. **Stop rule:** a maximum of 4 steps. If no team fits with confidence above the threshold, route to "triage" and flag for a person.
5. **Approval:** escalations notify a person; routes are written automatically only within the rules.
6. **Log:** ticket id, keywords found, team chosen, confidence, and the reason in one sentence.

```
Agent: ticket-router
Goal: route each new ticket to exactly one team, or escalate.
Tools (least permission):
  - read_ticket        (read)
  - read_team_list     (read)
  - set_route_field    (write: route only)
Escalation keywords (checked first, in code): outage, down, data breach, legal, GDPR,
lawsuit
Stop: 4 steps maximum. Unsure → route = triage, flag = needs_human
Log: ticket_id, keywords, team, confidence, reason
```

Test it: send one normal ticket, one with the word "outage", one ambiguous ticket, and one with a broken tool. Pass means: the outage ticket escalates, the ambiguous one goes to triage, and the broken tool stops the run cleanly.

Step 4: check the skill and agent on their own

Each one is tested alone. Do not connect them into a pipeline for this exercise. The skill must produce correct replies on its own; the agent must route correctly on its own, with only the tools it needs.

9. Skills and agents in other tools

Tool	Reusable instructions	Delegated helper	Acting agent
Claude (chat, Desktop and Code)	Skills (SKILL.md) and project instructions	Subagents in Claude Code	Agents with tools, via Claude Code or an agent framework
Lovable	Project knowledge and reusable prompts	Not a separate concept	Connected workflows (for example to n8n) do the acting
ChatGPT	Custom GPTs and Projects with instructions and files	Varies by product	Actions and connected tools, with approval steps

Feature names change often. Check the current documentation for the tool you use before you rely on a name.

10. Common mistakes

- **Making an agent for a fixed template.** If the steps never change, a skill or code is cheaper and safer.
- **Giving an agent write access to everything.** Start read-only, and add one write permission at a time.
- **No stop rule.** An agent with no step limit can loop and spend.
- **Hiding the rules in one long prompt.** Split them: skill for know-how, code for exact rules, agent for choosing steps.
- **Skipping the "neither" answer.** Many tasks need only a prompt or a script.

11. Quick check

1. A team writes the same sales email in the company tone every week. What is it?
2. A model reads 60 invoices and returns a table of mismatches. What is it?
3. A model decides which of three systems to query, then sends a message. What is it?
4. Calculating tax from a fixed table. What is it?

Answers: 1. Skill. 2. Subagent. 3. Agent (with approval before sending). 4. Neither; use code.

12. Glossary and further reading

Term	Plain meaning
SKILL.md	The main file of a skill: header plus instructions.
Subagent	A focused helper with its own context that returns a result.
Agent	A model that chooses steps and uses tools toward a goal.
Tool	An action the agent can call, such as read a file or send a message.
Stop rule	The limit (steps, spend, time) that ends an agent run.
Human-in-the-loop	A person approves an action before it happens.

- Anthropic documentation on agents, tools and skills: docs.anthropic.com.
- Claude Code and skills documentation, with the current feature list; check before you build.

SkillNavora GenAI Launchpad · Written by SkillNavora · Part 4 of the automation series · Uses fake data only

n8n Multi-Agent Onboarding: From Zero to Hosted

You will build a small new-hire onboarding system. An HR form sends a webhook to n8n. A Coordinator agent records the hire in a Google Sheet, asks an Access Agent to plan system access, and asks a Scheduling Agent to plan meetings. Everything uses **fake data** and a **dedicated test account**. Follow the steps in order, and check the "You should see" line after each one.

> **Safety first.** Never connect your personal Gmail, your main Google Drive or real employee data to a practice workflow. Create a test Google account and a test sheet. Email sending stays off until a person approves it.

1. The big picture

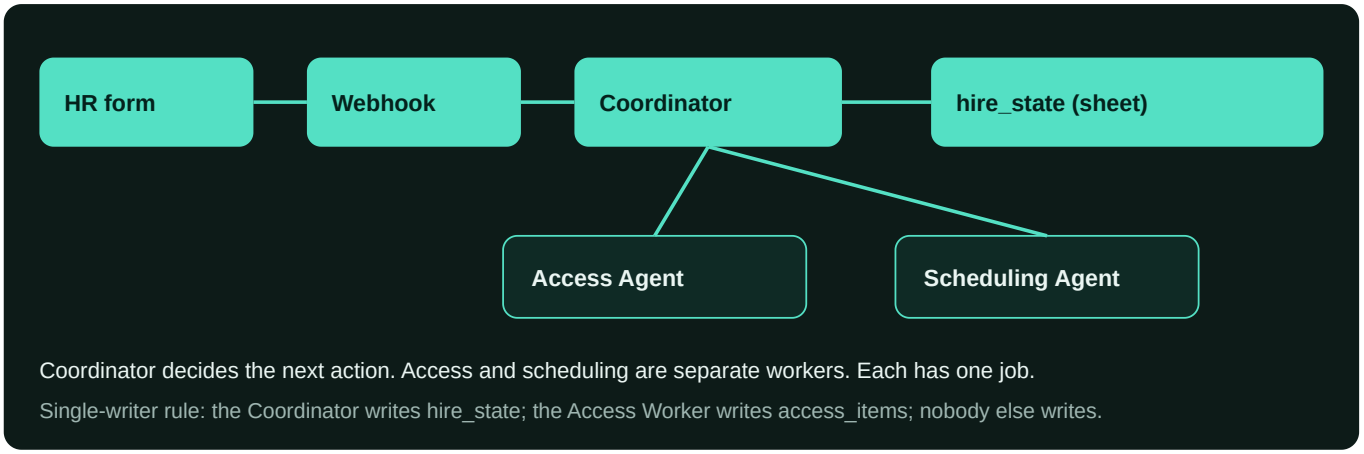


Figure 1. The onboarding system. Arrows show who calls whom.

2. Hosting n8n (choose one path)

n8n can run in three places. Pick one and keep the rest of the guide the same.

Option	Best for	What you manage
n8n Cloud (managed)	Learning, quick start	Nothing on the server; pay per plan
Self-hosted on a VPS (virtual server)	Full control, one fixed monthly server cost	Server updates, backups, HTTPS certificate
Local with Docker	Testing on your own computer only	Your computer must stay on; webhooks need a public address to receive real events

Self-hosting on a VPS: step by step

1. **Choose a VPS plan.** Hosting companies such as Hostinger sell VPS plans. Compare the monthly price, the RAM, and whether a one-click n8n template is included. Prices change; read the live cart before you pay. Hostinger's cart in the class showed a KVM 1 plan with a free domain for the first year, which is a promotion, not a permanent price.
2. **Point a domain or subdomain** (for example `n8n.example.com`) at the server's IP address.
3. **Install n8n** with the one-click template or with Docker. Set an encryption key and a strong admin password.
4. **Turn on HTTPS** so webhook URLs are secure.
5. **Back up** the n8n data folder and the encryption key separately.

You should see: the n8n login page at your domain, over HTTPS, and you can sign in to the editor.

3. Your first workflow: a Webhook test

A **webhook** is a doorbell. The workflow waits until something rings it. Instead of checking repeatedly for new hires, n8n waits, and the HR form sends a POST request when a hire is confirmed.

1. Click **Create workflow**, then add a **Webhook** node.
2. Set **HTTP Method** to POST, **Authentication** to None (for this practice), and **Respond** to Immediately.
3. Copy the **Test URL**. It looks like `https://your-n8n.example.com/webhook-test/<long-id>`.
4. Click **Listen for test event**.
5. In another tab, send a test submission (step 4 below shows the body to use), or paste this into a tool such as an HTTP client:

```
POST https://your-n8n.example.com/webhook-test/<long-id>
Content-Type: application/json

{
  "hire_id": "H-2026-0001",
  "candidate_name": "Test Candidate",
  "department": "Product",
  "role": "PM",
  "manager": { "name": "Test Manager", "email": "manager.test@example.com" },
  "start_date": "2026-09-01",
  "candidate_email": "candidate.test@example.com"
}
```

You should see: the Webhook node shows one item. Open the **JSON** view. Your fields sit under `body`. The `hire_id` is the field that identifies the person in every later step.

> **Test versus production.** The *test* URL works only while "Listen for test event" is open. The *production* URL works all the time, but only after you activate the workflow. Use the test URL while building, then switch the form to the production URL and activate.

Reading the webhook output

Part	What it holds	Use it for
headers	Who sent it, content type	Checking the request came from the form
params, query	URL parameters	Usually empty for a form POST
body	Your data	Everything your workflow needs
webhookUrl, executionMode	Which URL and mode ran	Confirming test versus production

Use the **Schema**, **Table** and **JSON** tabs in the output panel. The JSON view is the most reliable when you are learning.

4. The data contract

A **data contract** is the agreed list of fields and their meaning. Every part of the system uses the same names. Changing a name in one place breaks the others.

Field	Example	Rule
hire_id	H-2026-0001	Unique per hire. The primary key. Never reused.
candidate_name	Test Candidate	Required
candidate_email	candidate.test@example.com	Personal email; required until the work email exists
department, role	Product, PM	Must match a value in the access registry
manager.name, manager.email	Test Manager	Required
start_date	2026-09-01	Must be in the future
run_id	RUN-H-2026-0001-001	Generated by the Coordinator, one per attempt

5. The workbook: four tabs, one writer each

Create a Google Sheet named **onboarding_agents_workbook** using a test Google account. Each tab has **one writer**. This stops two parts of the system from overwriting each other.

Tab	Sole writer	Purpose
hire_state	Coordinator	One row per hire: overall, access and scheduling status
access_items	Access Worker	One row per hire per access item
meetings	Scheduling Worker (not built here)	One row per proposed meeting
access_registry	Hand-maintained (nobody writes it at runtime)	Which apps each role gets, and whether approval is needed

Status words

Tab and column	Allowed values	Terminal?
Overall, access and scheduling status	in_progress, complete, needs_attention	complete and needs_attention end the flow
access_items status	not_requested, requested, awaiting_approval, granted, not_granted	granted and not_granted are final
meetings status	pending, proposed, conflict, confirmed, not_scheduled	confirmed and not_scheduled are final

CSV templates (copy these into the tabs)

Use these headers exactly. Lookups are exact matches, so spelling and capital letters must match.

```
hire_state
hire_id,candidate_name,candidate_email,department,role,manager_name,manager_email,start_date,run_id,overall_status,access_status,sch

access_items
hire_id,run_id,app,level,status,approval_needed,updated_at

meetings
hire_id,run_id,role,meeting_name,duration_minutes,within_days,priority,status,proposed_time,updated_at

access_registry
row_key,app,level,role,drive_role,grant_mode,depends_on
Identity|create|PM,Identity,create,PM,,approval_required,
Identity|create|Designer,Identity,create,Designer,,approval_required,
Identity|create|Developer,Identity,create,Developer,,auto,
Slack|edit|PM,Slack,edit,PM,,auto,Identity|create
Slack|edit|Designer,Slack,edit,Designer,,approval_required,Identity|create
Slack|edit|Developer,Slack,edit,Developer,,auto,Identity|create
Jira|view|PM,Jira,view,PM,,auto,Identity|create
Jira|admin|Developer,Jira,admin,Developer,,auto,Identity|create
Figma|edit|Designer,Figma,edit,Designer,,approval_required,Identity|create
Onboarding Doc|view|PM,Onboarding Doc,view,PM,reader,approval_required,
Onboarding Doc|edit|Designer,Onboarding Doc,edit,Designer,writer,approval_required,
Onboarding Doc|view|Developer,Onboarding Doc,view,Developer,reader,auto,

meetings_template (the reference list, keep it in access_registry or a separate tab)
role,meeting_name,duration_minutes,within_days,priority,purpose
PM,Manager 1:1,60,2,1,Align on 30-60-90 day plan and immediate priorities
PM,Product Vision Walkthrough,45,5,2,Walk through product strategy, roadmap and OKRs
PM,Stakeholder Introduction,30,7,3,Meet key cross-functional stakeholders
Designer,Manager 1:1,60,2,1,Align on design principles and current projects
Designer,Design System Walkthrough,45,5,2,Walkthrough of the design system and tools
Developer,Manager 1:1,60,2,1,Align on engineering standards and sprint cadence
Developer,Tech Stack Walkthrough,45,5,2,Deep dive into the tech stack and dev environment
```

> **Open decisions.** The class workbook flagged one cell to decide before going live: the Developer grant mode for the Onboarding Doc. The sample above uses `auto` for read-only access; confirm your own policy. Delete any example rows before using the sheet for real.

Create the sheet

1. Create the spreadsheet and rename the first tab to `hire_state`.
2. Add the other tabs and paste each CSV header row.
3. Paste the registry rows into `access_registry`.

4. Share the sheet only with your test account.

6. The Google Sheets connection

1. In n8n, open **Credentials** and add a **Google Sheets** credential. Sign in with the test account and approve the access. Give only Sheets access.
2. Add a Google Sheets node, choose **Append Row**, and select the document and tab.
3. Map each field from the webhook body to a column. Use expressions such as `{{ $json.body.hire_id }}`.

You should see: after one test submission, a new row appears in `hire_state`. If it does not, check the document name, the tab name and the credential.

7. The Coordinator agent

The Coordinator is an AI Agent node with a chat model (Anthropic in the class), an **Update row** Google Sheets tool, and two tools that call the Access and Scheduling agents. Its job is to decide the next action, not to do the work itself.

Tool descriptions matter

The agent reads each tool's description to decide when to use it. Write it as a rule: "Update the hire_state row for a given hire_id. Use this to set overall_status, access_status and scheduling_status." Vague descriptions cause wrong calls.

Use the narrowest operation

Google Sheets offers Append, Update, Append or Update, Clear, Create and Delete. Give the agent **Update Row** only. Never give it Clear or Delete. Least permission is the rule.

System message (the instructions)

```
Role: You are the Coordinator for new-hire onboarding.
Steps:
1. Read the hire from the input. Confirm hire_id is present; if not, stop and set needs_attention.
2. Call the Access Worker with these named fields, each separately:
   hire_id, candidate_name, role, manager_name, manager_email, start_date, candidate_email,
   run_id (format RUN-{hire_id}-001).
   Do not join the fields into one string.
3. Call the Scheduling Worker with hire_id, candidate_name, role and start_date.
4. Update the hire_state row with the statuses you received.
Output format (JSON only):
{ "next_action": "dispatch | wait | close", "reasoning": "one sentence" }
Never send email. Never change access directly.
```

Why this helps: named fields prevent one long string that the workers cannot parse. Asking for JSON output makes the next node reliable.

Turn on **Return Intermediate Steps** while building, so you can see each tool call. Turn it off in production if the output gets long.

8. The Access Agent (a separate workflow)

The Access Agent is its own workflow, started by the Coordinator through an **Execute Workflow** or "Call workflow" tool. It receives the named fields from step 7.

1. **Start:** "When Executed by Another Workflow", with the input fields listed (run_id, hire_id, role, manager_name, manager_email, start_date, candidate_email, dispatched_at).
2. **Idempotency check:** a Get Row(s) step reads `access_items` filtered by `hire_id = {{ $json.hire_id }}`. If rows already exist for this hire, stop. This stops the same hire being processed twice.
3. **If node:** true branch means "already processed, stop"; false branch continues.
4. **AI Agent:** reads the role from the input, and uses the **Registry Lookup tool** (read `access_registry`) to find each app for that role.
5. **For each app:** if `grant_mode` is `auto`, write a row with status `requested`. If `approval_required`, write `awaiting_approval` and do not grant.
6. **Write Row tool:** appends to `access_items` only. Never writes to `hire_state`.

7. **Gmail (optional):** send a message only after a person approves. For practice, leave the Gmail tool disconnected.

You should see: running the same hire twice writes rows only once. The second run stops at the If node.

9. The Scheduling Agent (concept only in this build)

The Scheduling Agent reads `meetings`, proposes times for each role's meetings within the `within_days` window, and writes `proposed` rows. It must never confirm a meeting without a person. In this guide, build the Coordinator and Access Agent first; add scheduling once those work.

10. Duplicate prevention and error paths

Risk	Prevention
The same form submitted twice	Check `hire_id` exists before appending a new hire row.
Missing email or hire_id	Validate first; set `needs_attention` and stop.
Wrong role	Look up the role in the registry; if not found, stop with "needs_attention".
Webhook delivery fails	Check the executions list; re-send the test event; use a retry only for safe steps.
Access granted without approval	Rows with `approval_required` stay `awaiting_approval`.

Add an **Error Trigger** workflow that writes "needs_attention" and the error message to `hire_state`, so failures are visible in the sheet, not hidden in logs.

11. Monitor the workflow

- Open **Executions** for each run. A green run may still have made a wrong decision, so read the output of each node.
- Track the **failure rate**. One failure in one production run is a 100 percent failure rate, so judge trends over many runs.
- Track **tokens per run** from the agent's log, and compare with the cost guide in this series.
- Track **run time**. Multi-agent runs take seconds to minutes; set a timeout.

12. The HR Portal: intake form and dashboard

The HR Portal is a small web app (built in Lovable in the class). It has two screens.

Screen 1: New Hire Intake

Section	Fields	Rules
Candidate	Candidate name, candidate personal email	Name required; email must look like an email
Assignment	Department, role, manager	Department and role come from a list; manager from a list
Timing	Start date	Must be in the future
Action	Confirm hire	Generates the hire_id, then sends the webhook POST

Screen 2: Dashboard

A table of hires with columns: hire_id, name, role, start date, overall status, access status and scheduling status. Colour the status (in_progress, complete, needs_attention). Add a "Show webhook status" or "Export as CSV" button if you want operational help.

Build it in Lovable (outline)

1. Create a project and describe the two screens in plain English: "An HR portal with an intake form and a dashboard, using fake data only."
2. Ask for the form fields exactly as in the table above.
3. When the form is ready, paste the **test** webhook URL and ask: "Send the form data to this URL as JSON after Confirm hire."
4. Submit one fake hire and check that it appears in the n8n test panel.

5. Switch to the **production** URL, and activate the workflow in n8n.

> **Do not use real personal data** in a demo app. Fake names and test addresses only, and do not publish a live URL that accepts real submissions.

13. Build order (do not skip steps)

1. Host n8n and sign in.
2. Webhook test: send one fake body and read the JSON.
3. Write one row to `hire_state`.
4. Add the Coordinator with the system message, using a test model key.
5. Build the Access Agent, test duplicates, then connect it.
6. Add the form and the dashboard.
7. Switch to production URLs and activate.

14. Test checklist

Test	Pass when
Valid hire	One row in hire_state, access rows written
Same hire submitted twice	No duplicate access rows
Missing email	needs_attention, no access granted
Unknown role	needs_attention with a reason
Approval-required app	awaiting_approval, nothing granted
Webhook down	Failure visible in executions and in the sheet

15. Quick check

1. Why is `hire_id` the most important field?
2. Which tab does the Access Worker write to, and which does it never write?
3. Why should the Coordinator never be given the Delete operation?
4. What does the If node in the Access Agent prevent?

Answers: 1. It identifies the hire in every step and prevents duplicates. 2. Writes `access_items`; never `hire_state`. 3. Least permission; a mistake could erase the workbook. 4. Processing the same hire twice.

16. Glossary

Term	Plain meaning
Webhook	A URL that starts a workflow when something posts to it.
Test versus production URL	Test works only while you are listening; production works once the workflow is active.
Data contract	The agreed field names and meanings.
Single writer	Only one part of the system may write to each tab.
Idempotency	Running the same request twice gives the same result, with no duplicate rows.
Registry	The reference table of apps and approval rules per role.

- n8n documentation: docs.n8n.io (webhook and Google Sheets node pages).
- Your hosting provider's VPS documentation for installing Docker and HTTPS.

Make.com: Scenarios, Routers and AI Agents

Make is a visual automation tool. You connect apps as boxes on a canvas and draw the flow between them. This guide takes you from your first scenario to the Resume Analysis Agent from the class. It uses fake data and a test account throughout.

1. Words you need first

Word	Meaning
Scenario	One automation: a chain of modules that runs in order.
Module	One step, such as "Watch responses" or "Add a row".
Trigger	The first module, which starts the scenario (a new form response, a schedule, a webhook).
Bundle	One item of data passing between modules.
Router	A split that sends each bundle down one or more paths based on filters.
Filter	A condition on a path, for example "score is above 70".
Connection	The saved login to an app (a credential). Use a test account.
Operation	Spend of the plan's credits per module run. Check the plan page for current credit rules.

2. The Make screen

The left menu has: **Org**, **Scenarios**, **AI Agents**, **Credentials**, **Webhooks**, **MCP Toolboxes**, **Templates**, **Data stores** and **More**. The canvas in the middle is where you build. The bottom toolbar has **Run once**, a schedule toggle (for example every 15 minutes) and the undo and settings icons.

3. Your first scenario: form to sheet

1. Click **Scenarios**, then **Create a new scenario**.
2. Add a **Google Forms** module, choose **Watch Responses**, and connect the test account. Pick a test form with two fields.

3. Add a **Google Sheets** module and choose **Add a Row**. Select a test sheet.
4. Map each form answer to a column using the mapping panel (click a field, choose the answer).
5. Click **Run once**, then submit one test response to the form.

You should see: the module bubbles turn green, the sheet gets a new row, and the output panel shows the values you mapped.

4. Scheduling and settings

- **Schedule:** the toggle sets how often the scenario checks for new items. The class example was every 15 minutes. Shorter intervals use more operations.
- **Sequential running:** keep it on when order matters, so two runs cannot write at the same time.
- **Error handling:** right-click a module to add an error route. Route failures to a sheet row or a message so they are visible.
- **Notifications:** decide when Make emails you about failed runs. Turn on notifications for failures, not for every success.

> **Before you enable a scenario,** check three things: its operations per month against your plan, its notification setting, and what happens on failure. A scenario left running on a schedule keeps spending operations.

5. Routers and filters

A router is a fork in the road. The class example, the Resume Analysis Agent, uses one.

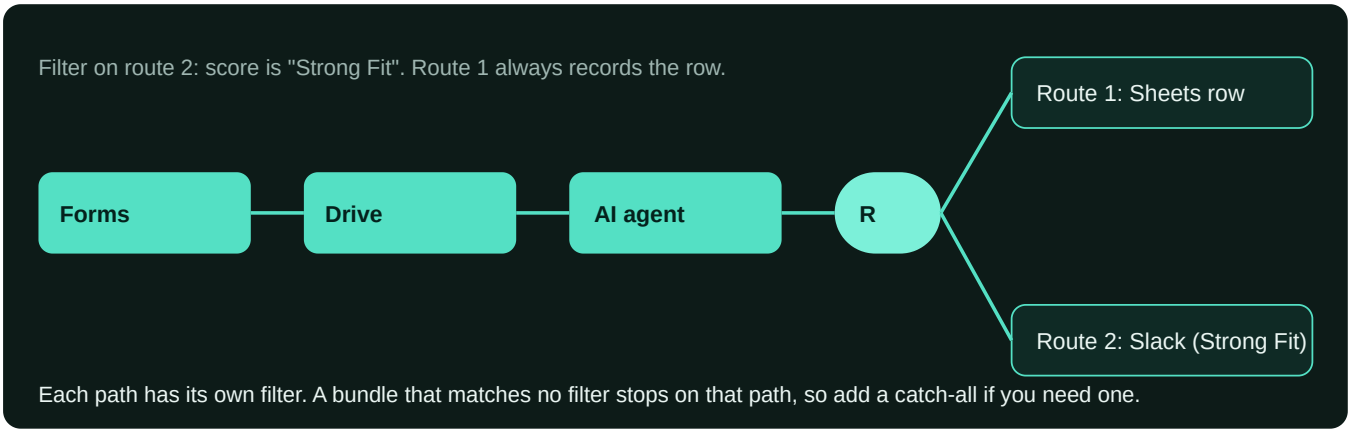


Figure 1. A router splits bundles into paths. Each path has a filter.

1. Click the wrench icon between two modules and choose **Add router**.
2. Add a filter to each route (for example the AI score field equals "Strong Fit").

3. Test with one bundle that matches and one that does not. Confirm only the right route runs.

6. The Resume Analysis Agent (class example)

The class scenario was built to read a candidate's resume and sort the candidate. The full chain is:

1. **Google Forms (Watch Responses):** the candidate submits a form with a resume file link.
2. **Google Drive (Download a File):** the resume file is fetched from Drive.
3. **AI agent:** reads the resume, uses a knowledge source (the job criteria) and returns a fit rating.
4. **Router:** route 1 adds a row to Google Sheets for every candidate; route 2 sends a Slack message only for "Strong Fit" candidates.

The agent in the screenshot also had an **Add tool** option, an **Add MCP** option, a **Chat** option and a knowledge attachment. The AI model module was an OpenAI connection in the class.

> **Privacy.** A resume contains personal data. For practice, use a fake resume with a made-up name and email. Do not send real candidates' files to a test scenario, and do not post Slack messages to a shared channel during testing.

Build checklist

Step	You should see
Forms trigger runs on a test response	One bundle with the form fields
Drive download returns a file	A file name and size in the output
AI step returns a fixed-format rating	One word from your allowed list (for example Strong Fit, Possible, No Fit)
Router sends only matching bundles to Slack	Slack message only for Strong Fit
Failed AI step	Error route writes a row marked "needs attention"

7. Maia: building with a description

Maia is Make's AI assistant (labelled Beta in the class screenshots). You describe what you want in plain English, and Maia proposes modules. The panel offers "Continue where we left off", "Get inspiration" and "Ask a question about Make".

1. Open a scenario and click **Build with Maia**.

2. Describe the goal in one sentence: "When a form is submitted, download the attached file, ask an AI model to rate it, and add a row to a sheet."
3. Review every module Maia proposes. Check each connection, field mapping and filter yourself.
4. Never trust a generated scenario without running it on fake data first.

Maia is a drafting aid. The person who runs the scenario owns its results, so review every step.

8. MCP Toolboxes and AI Agents in Make

MCP (Model Context Protocol) is a standard way for an AI app to use outside tools. Make's MCP Toolboxes group tools that an agent can call. Give an agent only the tools it needs, and read-only tools where possible. The same least-permission rule from the skills and agents guide applies here.

9. Failure, retries and cost

Issue	What to do
Module fails on a bad file	Route the error to a "needs attention" row; fix the input; re-run that bundle.
Scenario runs too often	Lengthen the schedule or use a webhook trigger instead of polling.
Operations run out	Check the plan usage page and reduce the schedule or the number of modules per run.
Duplicate rows	Store the form response ID and skip responses already in the sheet.

Check Make's current plan and credit page before you estimate cost. Plans and credit rules change.

10. Quick check

1. What starts a scenario?
2. What does a filter do on a router path?
3. Why should you use a test connection?
4. What should you do before enabling a scenario that Maia built?

Answers: 1. The trigger module (for example a new form response). 2. Decides whether a bundle goes down that path. 3. So a mistake cannot touch real accounts or data. 4. Run it on fake data and check every module and filter.

11. Glossary and further reading

- Make help centre: help.make.com (scenarios, routers, Maia and MCP pages).
- Make pricing and credit page: check the live page before you estimate operations.

Replit: Build, Run, Share and Deploy an AI App

Replit is a browser-based workspace where you write code, run it, and publish it on the web, with no installation on your computer. This guide goes from your first running page to an AI-powered app with secure keys, saved data, and a deployed address. Plans and prices are deliberately not covered here; check Replit's own pricing page when you need them.

> **Running example.** A small "Dish Description Studio": a form takes a dish name, category and price, calls an AI model, and shows a short description. It is the same food-seller example used in the tokens guide.

Part A. Basics

1. Words you need

Word	Meaning
Repl	A project: its files, its running app and its settings.
Workspace	The browser editor: file tree, code editor, console and preview.
Console	Where the program prints messages and errors.
Preview	The running app shown inside the workspace.
Secret	A private value (such as an API key) stored outside the code.
Deployment	A published copy of the app at a public address.
Replit Agent	An AI that builds or edits the project from your written request.
Environment variable	A named setting the program reads at runtime, such as `AI_API_KEY`.

2. Create an account and your first project

1. Sign up at replit.com. For practice, use a test account, not your work or personal main account.
2. Click **Create Repl**. Choose a template for a web app in a language you know (for example Python with a web framework, or Node.js).
3. Give the project a clear name, such as `dish-studio-practice`.

4. Press **Run**.

You should see: the console shows the server starting, and the preview pane shows a page. If the preview is blank, look at the console first.

3. Learn the workspace in five minutes

Area	What you do there
File tree (left)	Create, rename and delete files. Keep the main file easy to find.
Editor (centre)	Write code. Save happens automatically in most setups.
Console (bottom)	Read output and errors. Read the first error line, not the last.
Preview (right)	See the running page. Refresh after a change if needed.
Packages panel	Add libraries your code imports.
Secrets panel (lock icon)	Store API keys and private values.
Version history	See earlier saved states and roll back a change.

4. Make one change and see it run

1. Open the main file and change the page heading to `Dish Description Studio`.
2. Save, then refresh the preview.
3. If nothing changes, read the console for a red error line.

Common error: a missing package. Read the error ("No module named ..." or "Cannot find module ..."), add the package in the Packages panel, and run again.

5. Build the input form

Add a form with three fields: dish name, category (a dropdown: Breakfast, Lunch, Dinner, Dessert), and price in rupees. Use a submit button. Give every field a visible label. Labels help screen-reader users and make the form easier to use with a keyboard.

```
<form method="post" action="/describe">
  <label>Dish name <input name="dish" required></label>
  <label>Category
    <select name="category">
      <option>Breakfast</option><option>Lunch</option>
      <option>Dinner</option><option>Dessert</option>
    </select>
  </label>
  <label>Price (Rs) <input name="price" type="number" min="1" required></label>
  <button type="submit">Describe my dish</button>
</form>
```

You should see: the form in the preview. Submitting it does nothing useful yet; that is the next part.

Part B. Intermediate: connect an AI model safely

6. Store the key as a secret

An AI call needs an API key from your model provider. Never put the key in the code, in a screenshot, in a chat, or in a public project.

1. Create a key in your AI provider's console, with a monthly spending limit set there.
2. Open the **Secrets** panel and add `AI_API_KEY` with the key as its value.
3. In code, read it from the environment: Python `os.environ["AI_API_KEY"]`; Node `process.env.AI_API_KEY`.

> **If a key leaks,** delete it in the provider console at once and create a new one. Changing the code does not undo an exposed key.

7. Write the AI call with limits

Every call should have three guards: a fixed instruction, a capped output length, and a logged usage count. These make cost predictable (see the tokens guide).

```
import os
# Use your provider's official SDK. Names below are placeholders.
from ai_client import Client

client = Client(api_key=os.environ["AI_API_KEY"])

SYSTEM = ("You write short, appetising menu descriptions for local food sellers. "
          "Warm tone. Maximum 50 words. No prices or claims about health.")

def describe(dish, category, price):
    user = f"Dish: {dish}\nCategory: {category}\nPrice: Rs {price}"
    reply = client.generate(system=SYSTEM, user=user, max_tokens=120,
temperature=0.7)
    print("tokens in/out:", reply.usage.input_tokens, reply.usage.output_tokens)
    return reply.text
```

You should see: after one submission, the console prints the token counts and the preview shows a description of about 50 words.

8. Handle failure gracefully

- If the key is missing, show "AI is not configured" instead of a crash.

- If the call times out, show a retry message and do not retry in a tight loop.
- If the output is empty, show a fallback message and log the input.

9. Save results to a database

To keep a history of generated descriptions, add a small database. Replit offers a built-in database option in the workspace; alternatives include a hosted database service. Store only the fields you need: dish, category, price, description, time, and the token counts. Do not store personal data for a practice app.

Table: descriptions	Type	Note
id	number	Primary key, automatic
dish, category	text	From the form
price	number	Validated: greater than zero
description	text	From the AI call
input_tokens, output_tokens	number	For cost tracking
created_at	time	Automatic

Part C. Advanced: deploy, monitor and collaborate

10. Deploy the app

1. Open the **Deploy** panel and choose the type that fits: a website or web app (always-on address) is the usual choice for this example. Scheduled or background types suit other jobs.
2. Check the build command and run command the panel shows. They should match how you ran the app.
3. Deploy, then open the public address in a private browser window.
4. Test a normal submission, an empty form, a negative price, and a simulated AI failure.

You should see: the same page as in the workspace, at a public address, with secrets working. Deployment options and limits change; confirm them on Replit's documentation before you rely on them.

11. Logs, errors and rollback

- **Logs:** check the deployment logs after each change. Log the request type, the status and the token counts, never the API key.
- **Rollback:** use version history to return to an earlier saved state if a change breaks the app.

- **Health check:** keep one page that returns "ok" so you can see at a glance that the app is up.

12. Collaboration

Invite a teammate to a project only for practice work with fake data. Agree who edits which file. Use version history and comments to explain changes. Do not share secrets with collaborators; they should add their own key for their own testing.

13. Using the Replit Agent well

1. Describe the app in one paragraph: who uses it, what the inputs are, what the outputs are, and what it must never do (for example: never store customer phone numbers).
2. Ask for one feature at a time, then test it.
3. Read every file the Agent changes. Ask it to explain anything you do not understand.
4. Never let the Agent paste a real key into code. Check the secrets panel yourself.

Part D. Other tools you may need

Tool	Why you need it	Safe practice
AI provider account (console)	Gives the API key for model calls	Set a spending limit; use a test key
GitHub	Stores code history outside Replit, and lets you move projects	Use a practice repository with no secrets in it
A database (built-in or hosted)	Stores generated results	Store only the fields you need
Spreadsheet (for example Google Sheets)	Quick data sink or log for non-technical teammates	Test sheet only
Browser and dev tools	Inspect network calls and errors	Never paste keys into screenshots
Email or messaging (optional)	Notifications to a person	Approval before sending; test account only

Part E. When Replit is the right choice

Choose Replit when	Choose another tool when
You want to build and deploy a web app from a browser	You only need a no-code workflow between apps (use n8n or Make)

You are learning to code and want instant feedback	Your app needs heavy, specialised infrastructure from day one
You need a prototype for users to test quickly	You need strict data residency rules that the platform cannot meet (check first)

Part F. Test checklist

Test	Pass when
App runs in the workspace	No red error in the console
Key comes from secrets only	Code contains no key; the call works
Output is capped	Description stops near 50 words
Missing key	A clear message, no crash
Negative or empty price	Validation message; no database write
Deployed address works	Normal path returns a page
Rollback works	An earlier version restores the page

Part G. Common problems

Problem	Fix
"Module not found"	Add the package in the Packages panel and run again.
Key error at runtime	Check the secret name matches the code exactly.
Works in the workspace, fails when deployed	Check the run command and the port setting in deployment.
Blank preview	Read the console; the first red line is the cause.
Output too long, costs rising	Lower max tokens; ask for a word limit; check usage logs.

Part H. Quick check

1. Where should an API key be stored?
2. What is the first thing to read when the preview is blank?
3. Why cap output length on every AI call?
4. What is a safe way to share a practice project?

Answers: 1. In Secrets, read from the environment. 2. The first red line in the console. 3. It keeps cost and response size predictable. 4. Share only fake data, and never share secrets.

Glossary

Term	Plain meaning
Repl	A project in Replit.
Secret	A private value kept outside the code.
Deployment	A published version at a public address.
Environment variable	A named setting the program reads at runtime.
Version history	Saved earlier states you can return to.

- Replit documentation: docs.replit.com.