沧蓝工坊

# AI 编程指南

从基础认知到工程落地

66 讲，一份讲清一件事

中英双语版

Bilingual Edition

# 目录

**引言：为什么写这本书 . . . . . 9**

**第一部分 基础与提问 . . . . . 11**

01 大模型到底是个什么东西 . . . . . 12

02 怎么把话说清楚：提示词入门 . . . . . 15

03 把要求写成清单：结构化提示词 . . . . . 18

04 让 AI 记住前面说过的话 . . . . . 23

05 给 AI 一个身份，它就更像样 . . . . . 26

06 给它看几个例子，比解释半天管用 . . . . . 29

07 让它一步一步想：思维链 . . . . . 32

08 让它按你要的格式输出 . . . . . 35

**第二部分 写代码 . . . . . 39**

09 让 AI 写代码的正确姿势 . . . . . 40

10 让 AI 帮你审代码 . . . . . 44

11 代码跑不起来，怎么让 AI 帮你查 . . . . . 48

12 让 AI 帮你把烂代码理顺 . . . . . 51

13 让 AI 写测试，省一半时间 . . . . . 55

14 让 AI 补注释和文档 . . . . . 58

**第三部分 接入与工程 . . . . . 61**

15 调 API 的第一步：把模型接进你的程序 . . . . . 62

16 一个字一个字往外蹦：流式输出 . . . . . 65

17 出错了怎么办：重试和兜底 . . . . . 69
18 别让账单失控：Token 和成本 . . . . . 73
19 那么多模型，到底选哪个 . . . . . 77
20 上下文窗口：它能一次看多少字 . . . . . 81
21 长文档怎么喂给 AI . . . . . 84

**第四部分 RAG 与 Agent . . . . . 87**

22 向量检索：让 AI 从你的资料里找答案 . . . . . 88
23 RAG 是什么，为什么公司都在做 . . . . . 91
24 从零搭一个能用的知识库 . . . . . 95
25 资料切多大块才合适 . . . . . 98
26 选哪个模型做向量化 . . . . . 101
27 重排序：把最相关的挑出来 . . . . . 105
28 Agent：让 AI 自己动手干活 . . . . . 109
29 让 AI 会用工具 . . . . . 112
30 复杂任务怎么拆成几步 . . . . . 115
31 让 AI 记住以前的事 . . . . . 118
32 MCP：给 AI 装一个统一的插口 . . . . . 121
33 把多个 AI 串成一条流水线 . . . . . 124

**第五部分 部署与调优 . . . . . 127**

34 在自己电脑上跑模型 . . . . . 128
35 公司内部署：数据不出门 . . . . . 131
36 让模型跑得更快 . . . . . 134
37 微调：教它你自己的东西 . . . . . 137
38 微调前先准备好数据 . . . . . 141

39 怎么知道它到底行不行：评测 . . . . . 144

**第六部分 安全、协作与上线 . . . . . 147**

40 安全底线：什么不能做 . . . . . 148

41 别把敏感信息漏出去 . . . . . 151

42 怎么发现它在胡说 . . . . . 155

43 一份清单：AI 编程最常见的坑 . . . . . 158

44 团队一起用 AI 的规矩 . . . . . 161

45 一个能直接开工的项目模板 . . . . . 164

46 攒一个自己的代码片段库 . . . . . 168

47 用 AI 写脚本，把重复活干掉 . . . . . 171

48 一次处理一大批：批处理 . . . . . 174

49 把 AI 接到前端页面里 . . . . . 178

50 把 AI 接到后端服务里 . . . . . 182

51 手机上怎么调 AI . . . . . 186

52 算清楚一个 AI 功能要花多少钱 . . . . . 189

53 上线前必须过一遍的检查表 . . . . . 192

54 上线之后怎么持续改进 . . . . . 195

**第七部分 场景实战 . . . . . 199**

55 用 AI 写周报和办公文档 . . . . . 200

56 用 AI 做数据分析：从表格里问出答案 . . . . . 204

57 把图片、语音、文件交给 AI . . . . . 207

58 用 AI 做翻译：怎么翻得又准又像人话 . . . . . 211

59 用 AI 读懂你的整个项目 . . . . . 215

60 用 AI 做代码迁移和升级 . . . . . 219

61 提示词也要版本管理 . . . . . 223
62 A/B 测试：怎么知道改版真的变好了 . . . . . 227
63 长任务：让 AI 跑几个小时也不跑偏 . . . . . 230
64 AI 的失败模式清单：什么时候别信它 . . . . . 234
65 在团队里推广 AI：怎么带不写代码的同事上手 . . . . . 237
66 一年之后：这套东西会怎么变 . . . . . 241

**English Translation . . . . . 244**

Introduction: Why This Book Exists . . . . . 245
Part One Foundations and Asking . . . . . 248
01 What a Large Language Model Actually Is . . . . . 249
02 How to Say It Clearly: Getting Started with Prompts . . . . . 253
03 Turn Your Requirements into a Checklist: Structured Prompts . . 257
04 Making the AI Remember What You Already Told It . . . . . 262
05 Give the AI a Role and It Comes Out Much Better . . . . . 267
06 A Few Examples Beat a Long Explanation . . . . . 271
07 Let It Think One Step at a Time: Chain-of-Thought . . . . . 276
08 Make It Output the Format You Want . . . . . 281
Part Two Writing Code . . . . . 285
09 The Right Way to Have AI Write Code . . . . . 286
10 Getting AI to Review Your Code . . . . . 291
11 When Your Code Won't Run: Getting AI to Help You Debug . . . 295
12 Getting AI to Clean Up Messy Code . . . . . 299
13 Let AI Write the Tests and Save Half the Time . . . . . 303
14 Let AI Fill In the Comments and Docs . . . . . 307
Part Three Integration and Engineering . . . . . 311

15 Your First API Call: Wiring a Model Into Your Program . . . . 312
16 One Character at a Time: Streaming Output . . . . . 317
17 When It Breaks: Retry and Fallback . . . . . 322
18 Don't Let the Bill Run Away: Tokens and Cost . . . . . 327
19 So Many Models: Which One Do You Actually Pick . . . . . 332
20 The Context Window: How Much Text It Can See at Once . . . . 337
21 How to Feed Long Documents to an AI . . . . . 341
Part Four RAG and Agents . . . . . 345
22 Vector search: getting AI to find answers in your own files . . 346
23 What RAG is, and why every company is building one . . . . . 350
24 Building a working knowledge base from scratch . . . . . 354
25 How Big Should Each Chunk Be? . . . . . 358
26 Which Embedding Model to Pick . . . . . 362
27 Reranking: Pull the Most Relevant Passages to the Top . . . . 367
28 Agents: Letting the AI Do the Work Itself . . . . . 372
29 Getting the AI to Use Tools . . . . . 377
30 Breaking Complex Tasks into Steps . . . . . 381
31 Making the AI Remember Earlier Work . . . . . 385
32 MCP: One Standard Socket for the AI . . . . . 389
33 Chaining Several AIs into One Pipeline . . . . . 393
Part Five Deployment and Tuning . . . . . 398
34 Running a Model on Your Own Computer . . . . . 399
35 Deploying Inside the Company: Data That Doesn't Leave the Building 403
36 Making the Model Run Faster . . . . . 408
37 Fine-Tuning: Teach It Your Own Stuff . . . . . 412
38 Get Your Data Ready Before Fine-Tuning . . . . . 417

39 How to Tell Whether It Actually Works: Evaluation . . . . . 421

Part Six Safety, Teamwork and Launch . . . . . 426

40 The Safety Line: What You Don't Do . . . . . 427

41 Don't Let Sensitive Information Slip Out . . . . . 431

42 How to Catch It Making Things Up . . . . . 435

43 One Checklist: The Most Common AI Coding Pitfalls . . . . . 440

44 Rules for Using AI as a Team . . . . . 445

45 A Project Template You Can Start Work With Right Away . . . . . 449

46 Building Your Own Snippet Library . . . . . 453

47 Write Scripts with AI and Kill the Repetitive Work . . . . . 457

48 Handling a Big Batch at Once: Batch Processing . . . . . 461

49 Hooking AI into a Front-End Page . . . . . 466

50 Hooking AI into a Back-End Service . . . . . 470

51 Using AI on Your Phone . . . . . 474

52 Working Out What an AI Feature Actually Costs . . . . . 478

53 The Checklist You Have to Walk Through Before Launch . . . . . 483

54 How to Keep Improving After Launch . . . . . 487

Part Seven Hands-On Scenarios . . . . . 491

55 Using AI to Write Weekly Reports and Office Documents . . . . . 492

56 Using AI for Data Analysis: Asking a Spreadsheet for Answers . 497

57 Handing Images, Audio and Files to AI . . . . . 502

58 Using AI for Translation: Accurate and Still Human . . . . . 507

59 Using AI to Understand Your Whole Project . . . . . 512

60 Using AI for Code Migration and Upgrades . . . . . 517

61 Prompts Need Version Control Too . . . . . 522

62 A/B Testing: How to Know a Change Really Helped . . . . . 527

63 Long Tasks: Getting AI Through Hours of Work Without Drifting . 532

64 The AI Failure Mode Checklist: When Not to Trust It . . . . 537

65 Bringing AI into a Team: Getting Non-Coders Started . . . . 541

66 A Year from Now: How All of This Will Change . . . . . 546

Conclusion: The Rest Is You, Doing the Work . . . . . 550

Afterword . . . . . 552

**结语：剩下的靠你动手 . . . . . 555**

**后记 . . . . . 557**

# 引言：为什么写这本书

这本书讲一件事：怎么让 AI 真的帮你把活干完。

不是讲 AI 有多厉害，也不是讲它将来会取代谁。那些话你已经听得够多了。这本书只关心一个很具体的问题——你今天手上有一件活，怎么让 AI 帮你做完，而且做得对。

## 写给谁

写给已经在用 AI、但用得不太顺的人。

你可能已经试过：让它写段代码，出来一堆跑不通的东西；让它改个文案，改完还不如原来；让它总结个文档，它把重点全漏了。然后你就觉得，这东西也就那样。

问题通常不在 AI，在你怎么跟它说话。这跟找人帮忙是一个道理：你说「帮我弄一下」，对方只能瞎猜；你说清楚要什么、不要什么、做到什么程度算完，对方一次就能做对。

## 为什么用大白话写

因为大部分 AI 教程写得不像给人看的。

它们喜欢用「提示词工程」「上下文窗口」「检索增强生成」这样的词，然后假设你已经知道这些词是什么意思。结果你读完一遍，还是不知道明天上班该怎么用。

这本书里，每个专业词第一次出现，都会用一句话解释。不解释的词，就是不需要解释的词。如果哪一段你读了两遍还没懂，那是我的问题，不是你的问题。

## 这本书怎么排的

66 章，分七个部分，从最基础的认知一路讲到场景实战和长期跟进。

每一章都是一个独立的问题，不用从头读到尾。结构是固定的：开头三条告诉你学完能做到什么，然后是三个小节的正文，接着是一段能直接复制去用的提示词或代码，再是四个最容易踩的坑，最后是四条自查项。

中间那段「能直接抄的模板」是每章最值钱的部分。哪怕你正文全跳过，只把模板抄走，这本书也没白买。

## 三条读法

刚接触 AI 的，从第 1 章读到第 8 章。这八章讲的是怎么把话说清楚，后面所有内容都建在这上面。

已经在写代码的，直接跳到第 9 章。前面八章当手册，随时回头翻。

要做产品的，从第 15 章开始，一路读到第 53 章的上线检查表，再往后看场景实战。

只想快点解决眼前问题的，翻目录找最接近的那一章，直接跳到「能直接抄的模板」。

## 这本书不干什么

它不教你怎么训练模型，不讲 Transformer 的内部结构，不比较各家模型的跑分。这些事有更专业的书去讲。

它也不承诺你学会之后能拿多少工资、能做出多大的产品。那是你的事，不是这本书的事。

它只做一件事：把「怎么跟 AI 协作」这件事，拆成 66 个你今天就能用上的具体动作。

剩下的，靠你动手。

第一部分

# 基础与提问

第 01 — 08 讲

# 01
# 大模型到底是个什么东西

它不是搜索引擎，也不是数据库。它是一台「猜下一个字」的机器。搞懂这一点，后面所有的技巧你都能自己想明白。

---

**这篇讲什么**

☐ 知道它为什么这么会说话，也知道它为什么会一本正经地胡说

☐ 分得清「它真知道」和「它编的」这两件事

☐ 知道什么活能放心交给它，什么活千万别交

## 01 它其实只干一件事：猜下一个字

你给它一段话，它算出「下一个字最可能是啥」，写下来，再猜下一个。一个字一个字往外蹦，最后就凑成了一段完整的回答。

就这么简单。它没有在「思考」，也没有在「理解」你。它是在海量文字里学会了「什么词后面通常跟着什么词」。

所以它写东西特别顺。因为通顺的文字本来就最常见，它天天见的都是通顺的句子。

- 它是概率机器，不是知识库。它给的是「最像答案的话」，不是「答案」
- 它默认没记性。关掉对话就全忘了，下次得重新交代背景
- 它不知道自己错了。说错和说对，对它来说手感是一样的

## 02 那它为什么会一本正经地胡说

因为它猜的是「下一个字像什么」，不是「这件事对不对」。

一个根本不存在的书名、一个编造的论文编号，读起来照样通顺，所以它照样写得出来，而且写得很有底气。

这个现象有个专门的说法叫「幻觉」。不是它想骗你，是它脑子里根本没有「真假」这根弦。

- 越具体的东西越容易错：人名、日期、数字、链接、法条编号
- 越常见的知识越靠谱：写代码、改文案、解释概念，这些它很稳
- 越新的东西它越不知道：它的知识停在训练那天，之后的事得你告诉它

## 03 什么活能交给它，什么活别交

能交的：改写、总结、翻译、起名字、写样板代码、解释概念、把一堆乱想法整理成结构。这些活就算它做错了，你一眼就能看出来，改改就完事。

别交的：查最新事实、算精确的数、替你做最后决定、碰隐私和密钥。这些活它错了你未必看得出来，等发现就晚了。

判断标准其实就一句话：这事要是错了，会不会有人倒霉？会，就别让它一个人干。

### 能直接抄的模板

在提问最后加这一段，让它自己交代哪儿不靠谱

> 回答完之后，请你自己过一遍：
> 1.
> 里面有没有具体的人名、日期、数字、链接？有的话单独列出来，并说明你有多确定。
> 2. 有没有哪部分你其实不确定、但写得很肯定？直接指出来。
> 3. 不确定的地方就写「这部分我不确定」，不要编一个看起来合理的答案糊过去。

### 常见坑

**× 以为它在帮你查资料**

它不联网（除非你专门给它开了联网工具）。回答全靠训练时记下的东西，可能是旧的、可能是错的。要事实就自己去查，或者给它接上搜索。

**× 一段对话聊了几十轮**

聊到后面它会「忘」前面说过啥。重要的信息每隔几轮重述一次，或者干脆开个新对话，把背景重新交代清楚。

**× 让它直接算数**

它是猜字的，不是计算器。三位数以上的乘除它经常错。要算数就让它写段代码跑一遍，或者你自己用工具算。

× **把密钥、客户资料直接贴进去**

贴进去的内容可能被存下来、被用于训练。真要用，先把敏感部分换成假的（比如把真名换成「张三」、把密钥换成「XXX」），拿到结果再换回去。

## 做完自查

☐ 我知道它是在猜字，不是查资料，所以事实类内容我会自己核实

☐ 我不会把密钥、客户隐私、未公开的方案直接贴进去

☐ 重要的活，我会让它标出哪部分不确定

☐ 我不会指望它算精确的数，也不会指望它知道昨天发生的事

# 02
# 怎么把话说清楚：提示词入门

同一件事，说法不一样，结果差很远。这篇教你用三句话把要求说清楚，让 AI 第一次就给出能直接用的东西。

### 这篇讲什么

☐ 知道一句合格的提示词至少要交代哪三件事

☐ 会用「一个身份加一个样板」把语气和格式定下来

☐ 能把「不要什么」提前说清，少返工两三轮

## 01 先说清三件事：做什么、给谁看、要什么格式

很多人提问就一句「帮我写个周报」。它只能猜，猜出来的东西你八成要重写。

把三件事补上，效果立刻不一样。做什么：把这周三个项目的进度整理成周报。给谁看：部门主管，他只关心风险和下周安排。要什么格式：三段，每段不超过四行。

你会发现，能说清这三件事，说明你自己也想清了。写提示词的过程，其实是逼自己把活拆明白。

- 做什么：动词开头，一句只交一件活。别把写周报和排下周计划塞进同一句
- 给谁看：读者是谁决定用词深浅。给老板看和给新人看，写法完全不同
- 要什么格式：几段、多少字、要不要表格或小标题，直接说

## 02 给它一个身份，再给它一个样板

让它扮演谁，比你说十句形容词管用。

比如「你是一个做了十年招聘的 HR」，它写出来的招聘文案，用词就会往那个方向靠。

更狠的一招是给样板。贴一段你觉得写得好的文字，说「照这个语气写」。样板比形容词准得多：你说「专业一点」，它不知道你要多专业；你给它一段，它一读就懂。

- 身份要具体到年头和场景，「八年经验的初中语文老师」比「教育专家」好用
- 样板一段就够，贴太长它反而抓不住重点
- 样板里有你不想要的东西（比如爱用感叹号），顺手说一句「别学这个」

## 03 说完要什么，还得说不要什么

它有个常见毛病：你让它写三百字，它写八百字；你说别用术语，它还是塞一堆。

因为「不要」这件事，你不说它就当没有。把常见的毛病提前堵住，能省掉一半返工。

常用的几句：不要用感叹号，不要用行业黑话，不要编数据。不确定的地方让它标出来，不要写总结段。

- 禁止项一次列三到五条，列太多它反而顾不过来
- 字数用数字说，「三百字左右」比「简短一点」有用
- 它加了你不想要的内容，别自己删，直接说哪里不要，让它重写

**示例提示词**

> 请帮我写一份本周工作周报。
>
> 做什么：把这周三个项目的进度整理成周报。
> 给谁看：部门主管，他只关心风险和下周安排。
> 要什么格式：分三段，每段不超过四行，不要总结段。
>
> 你的身份：做了八年项目管理的老人，说话直接。
> 照这个语气写：「本周 A 项目延期两天，原因是测试环境挂了。下周一补上。」
>
> 不要做这些：不要用感叹号，不要用行业黑话，不要编造数字。

## 能直接抄的模板

把这个空壳存下来，每次换内容就行

```
做什么：（一句话，动词开头）
给谁看：（读者是谁，他最关心什么）
要什么格式：（几段、多少字、要不要列表）

你的身份：（具体到年头和场景）
照这个语气写：（贴一小段你觉得好的文字）

不要做这些：（三到五条，比如不要用感叹号、不要编数据）
```

## 常见坑

× **只说「写好一点」**

「好」是空的，它只能按最常见的样子写，也就是最平庸的那种。把「好」换成具体动作：开头放一个数字、每段不超过三行、结尾给一句建议。

× **一次问五件事**

它会五件事都做得马马虎虎，还可能漏掉一两件。一次只交一件，做完再做下一件。真要一起给就编上号，让它一条一条回。

× **说话绕，把它当人客气**

「能不能麻烦你帮我看看这个方案有没有可以改进的地方」，它抓不住重点。直接说「挑出最可能被老板否掉的三点，各给一句改法」。

× **不满意就自己动手改**

自己改等于没用上它。直接说哪里不对：「第二段太虚，换成具体的数字」，它会整段重写，比你自己抠字快。

## 做完自查

☐ 我的提示词里写清了做什么、给谁看、要什么格式

☐ 我给了它一个具体身份，或者一段能照抄的样板

☐ 我列了三到五条「不要」，并且给了明确字数

☐ 它第一次给得不好，我先改说法，而不是自己动手改

# 03
# 把要求写成清单：结构化提示词

同一个模型，有人用得像助手，有人用得像抽奖。差别就在有没有把要求写成清单。清单越清楚，它越不会跑偏。

---

### 这篇讲什么

☐ 会把一个模糊需求拆成五块，写成固定格式

☐ 知道哪几块必须写，哪几块可以省掉

☐ 能让它按编号逐条回话，方便你一条条验收

## 01 为什么写成清单，它就不跑偏了

你写一大段话，它读的时候是有侧重的。写在后面的容易被记住，混在句子中间的容易被跳过。同样的要求，写成一段话和写成三行，它执行的准头不一样。

清单把每一条单独拎出来，一条一行，它就没法漏。你验收也方便，一条一条对，缺哪条一眼看出来。

这不是什么高级技巧。就像给实习生派活，口头说一遍他记不全，写成一二三四他就不会错。清单还有个好处：你自己也不容易漏。

- 一条只写一件事。一句话塞两个要求，它经常只做前一个
- 用「1. 2. 3.」编号，别用逗号连成一串
- 顺序按重要性排，最要紧的放最前面

## 02 五块积木，基本够用了

背景：它不知道你的情况，得你告诉它。一句话说清你是谁、这事给谁做。

然后是任务、要求、输出格式、边界。任务一句话，动词开头；要求三到六条，越具体越好；格式写清长什么样；边界写清什么不能做。每块之间空一行，它读起来更清楚。

这五块不必每次用全。简单的活，任务加格式两块就够。复杂的活，五块都得写。拿不准就先写任务和格式，效果不够再补。

- 背景一句就够：「我们是六个人的设计工作室，客户主要是本地餐饮店」
- 要求要能被检查：「用词口语」没法验收，「不出现专业术语」可以验收
- 格式那块最好给一个空壳，让它往里填

## 03 让它照着清单回话

最后一招：让它在回答里带上编号，对着你的要求一条条说怎么做的。它写得越长，这招越省事。

这样你不用通读全文去找，扫一眼编号就知道哪条做到了、哪条没做到。

哪条没做到，直接说「第 4 条没做到，重写」，比笼统说「再改改」有效得多。

- 可以让它先复述一遍要求，确认没理解错，再动手写
- 长任务分两步：先让它给大纲，你点头了再写全文
- 同一份清单存下来，下次换个内容直接复用

**示例提示词**

> 【背景】
> 我们是六个人的设计工作室，客户主要是本地餐饮店。
>
> 【任务】
> 写一段给新客户看的服务介绍。
>
> 【要求】
> 1. 全文三百字以内。
> 2. 不用专业术语，说人话。
> 3. 提到三件具体的事：出图时间、改稿次数、收费方式。
> 4. 结尾留一句让人想联系你的话。
>
> 【输出格式】
> 分三段，每段前面加一个小标题。
>
> 【边界】
> 不要用感叹号，不要承诺做不到的事，不要编价格。
>
> 【回话方式】
> 写完后，按上面的编号 1-4 逐条说明你是怎么做到的。

## 能直接抄的模板

五块积木的空壳，复制下来往里填

```
【背景】
（一句话：你是谁，这事给谁做）

【任务】
（一句话，动词开头）

【要求】
1. （能被检查的具体要求）
2. （能被检查的具体要求）
3. （能被检查的具体要求）

【输出格式】
（几段、多少字、要不要小标题或表格）

【边界】
（不能做什么，三到五条）

【回话方式】
写完后按 1、2、3 逐条说明你是怎么满足的。
```

## 常见坑

### × 把清单写成小作文

每条都写一大段解释，它反而抓不住重点。一条一行，一行不超过二十个字，需要解释的放括号里。

### × 要求写得没法检查

「写得有感染力」这种要求，它只能自己猜。换成「开头用一个具体场景」「每段不超过三行」，做完你能一眼看出对不对。

### × 清单一次给十条以上

条数太多它一定会漏几条，而且你不知道漏了哪条。超过六条就拆两轮，先做主体，再改细节。

### × 只给清单不给例子

清单说「语气亲切」，它还是不知道你要多亲切。附一小段你满意的文字，说「照这个写」，比多加三条要求都管用。

## 做完自查

- ☐ 我这份清单里每条都能被检查，不是一堆形容词
- ☐ 我把任务拆成了一条一件事，没有一句塞两个要求
- ☐ 我给了输出格式，最好是个能直接往里填的空壳
- ☐ 我让它按编号回话，方便我一条条验收

# 04
# 让 AI 记住前面说过的话

它没有记性，关掉窗口就全忘。这篇教你两招：怎么在长对话里少重复，怎么把长期要交代的事存成一份随身文件。

### 这篇讲什么

☐ 知道它为什么会忘，也摸清它忘事的规律

☐ 会写一段背景摘要，开新对话一贴就接上

☐ 会把长期要用的信息做成文件，随用随贴

## 01 它不是忘了，是本来就没存

很多人以为它在偷偷记事。其实不是。它每次回答，只能看到这次对话窗口里的内容。窗口一关，这些就全没了。

窗口是有上限的。聊得太长，前面的内容会被挤出去，越靠前的越先没。具体能装多少看模型，但总有个头。

所以它「忘事」通常发生在聊了很久之后。刚开头交代的那些设定，到后面就不算数了。

- 一个话题一个对话，别在一个窗口里从写简历聊到改代码
- 重要设定说完过几轮再提一次，或者放在每轮提问的开头
- 发现它开始答非所问，就是窗口快满了，该开新的了

## 02 开新对话前，写一张交接单

换窗口不等于从头再来。你可以在关掉之前，让它自己写一份摘要。这事它做得很好，几分钟就完。

这份摘要就是下次的入场券。新窗口一开，先贴摘要，再提新要求，它立刻接得上。

摘要要短。写清你是谁、在做什么事、已经定了哪些、还差哪些。超过一屏就没用了，它读的时候还是会漏。宁可少写两句，也别写成流水账。

- 让它写摘要时限定字数：「两百字以内，只留结论和已定的事」
- 把摘要存到记事本，别指望在聊天记录里往上翻
- 同一件事反复做，摘要可以一直往里加，越用越准

## 03 把长期要交代的，做成一份随身文件

有些话你每次都得说：你的岗位、你的读者、你们公司的用词习惯。这些不该每次重打。重打的次数一多，你就懒得用了。

写成一个纯文本文件，放在桌面。每次开新对话，先把内容贴进去。

贴一次花十秒，比每次重新解释省事得多。写文案、改邮件、做汇报，都能用同一份。用得越久，这份文件越值钱。

- 文件里只放稳定不变的信息，一次性的要求别往里塞
- 每加一条先想一下：三个月后它还成立吗，不成立就别写
- 敏感信息（真名、客户名、密钥）换成代号再写进去

**示例：随身背景文件**

> 【我是谁】
> 在市场部做内容，主要写公众号和产品文案。
>
> 【我的读者】
> 二十五到三十五岁的上班族，没耐心，看前三行就决定要不要往下读。
>
> 【我们的用词习惯】
> 说人话，不用行业黑话。称呼读者用「你」。
>
> 【我不喜欢的】
> 开头铺垫太长、爱用感叹号、结尾强行升华。
>
> 【常用要求】
> 每段不超过三行，全文控制在八百字以内。

## 能直接抄的模板

换窗口前，让它写一份交接单

> 我们这段对话快结束了，我要开一个新窗口接着做。
> 请写一份两百字以内的交接单，包含：
> 1. 我的身份和要做的事。
> 2. 已经定下来的结论（只写定了的，不写过程）。
> 3. 还没解决的问题。
> 4. 下次开始前我需要补充的信息。
> 用短句，分条写，不要客套话。

## 常见坑

× **指望它记住上周说的**

跨对话它什么都不记得。上周的设定，这周得重新贴一遍。把它当成一个每天上班都要重新交代背景的新同事。

× **一个窗口聊一整天**

越聊越糊涂，后面全是废话。做完一件事就开新窗口，用交接单接上，比在一个长窗口里硬撑强。一个窗口只干一件事，成本最低。

× **交接单写成一长篇**

摘要太长，它读的时候照样会漏。两百字以内，只留结论和已定的事，讨论过程全删掉。

× **把客户真名写进背景文件**

文件会被反复贴进对话，等于反复上传。用「A 客户」「B 项目」这类代号，自己心里留一张对照表就行。

## 做完自查

☐ 我知道它只在当前对话里有记忆，跨窗口不会自动带过去

☐ 重要设定我会在提问开头再提一次

☐ 换窗口前，我会让它写一份两百字以内的交接单

☐ 我有一份随身背景文件，敏感信息都换成了代号

05

# 给 AI 一个身份，它就更像样

同一句话，加一句「你是谁」，它写出来的东西就完全不一样。这篇讲身份怎么写、写多细，以及为什么很多人写的身份没用。

### 这篇讲什么

- ☐ 知道身份是怎么影响它说话方式的，不再把它当成客套话
- ☐ 能写出一个具体、有约束力的身份，而不是一句「你是专家」
- ☐ 知道身份改不了事实准确度，什么该自己核对心里有数

## 01 身份不是客气话，是让它换一套说法

给它一个身份，等于告诉它「用谁的口气说话」。它写东西的用词、举例、句子长短，都会往那个方向靠。

同一个问题，你问「这段文案怎么样」，它给你一堆客气的场面话。你说「你是挑刺的甲方」，它就开始一条条挑毛病了。

原因很朴素。它在海量文字里见过无数种说话方式，你给的身份，就是帮它从里面挑出一种来用。

- 身份决定「谁在说」，不决定「说得对不对」。事实错了还是错
- 越具体越管用：说清是哪个行业、干了多久、现在面对谁
- 一个身份管一类活。写周报和写道歉信，别用同一个身份

## 02 一个有用的身份，三样东西就够了

别只写「你是专家」。专家太多了，它不知道该往哪个方向使劲，最后等于没说。

有用的身份有三个零件：干这行的什么人、干了多少年、现在要帮谁解决什么事。三样凑齐，它就知道该用什么词。

再加一句「不要什么」，效果更明显。比如「不要用术语」「不要写客套话」，它收敛得很快。

- 身份、对象、目标，一句话能写完，别铺成一大段
- 写「你是给小学生改作文的老师」，比「你是教育专家」有用得多
- 怕它乱猜，补一句「信息不够就先问我三个问题」

**示例提示词**

> 你是做了八年的物业客服主管，现在要帮我给业主写一条停水通知。
> 业主大多是上班族，看通知的时间很短。
> 要求：
> 1. 先说停水时间和影响范围，别绕。
> 2. 用大白话，不用「敬请谅解」这类套话。
> 3. 不超过 150 字，分三行。
> 4. 如果缺关键信息，先问我，不要自己编时间。

## 03 身份堆得越多，活干得越差

有人喜欢堆一长串：世界顶级、二十年经验、服务过五百强、拿过无数奖。这种写法基本没用。

堆得越多，它越容易开始演。满篇漂亮形容词，能干活的干货反而少了。你要的是活干得好，不是它自我介绍得漂亮。

两三句就够。剩下的篇幅留给任务本身：要什么、给谁看、写多长、什么格式。这四样比身份重要。

- 身份是调料，任务是主菜。主菜没交代清楚，调料救不回来
- 同一个身份用久了顺手，就存成模板，下次直接套
- 身份和风格要求要往一个方向走，别一边要严谨一边要搞笑

## 能直接抄的模板

把方括号填上，就是一条能直接用的身份提示词

> 你是【行业】里做了【几年】的【岗位】，现在要帮【谁】完成【什么事】。
>
> 说话要求：
> 1. 用大白话，不用行业术语；非用不可时，后面用括号解释一句。
> 2. 先给结论，再给理由，理由不超过三条。
> 3. 不要客套话，不要「希望对你有帮助」这类结尾。
> 4. 信息不够就先问我三个问题，别自己猜。
>
> 这次的任务是：【把你的需求写在这里】

## 常见坑

### × 只写「你是专家」

「专家」两个字对它几乎没有约束，它还是按默认方式回答。换成「你是给连锁餐饮写菜单的文案，读者是点外卖的上班族」，输出立刻不一样。

### × 身份和任务打架

一边说「你是严谨的律师」，一边让它「写得活泼点、多玩梗」，它会顾此失彼，两头都不像。身份和风格要求得往一个方向走。

### × 以为给了身份就不会出错

身份只改说话方式，不改事实准确度。它照样会编数据、编法条、编案例编号。要准确的内容，还是得你自己核对。

### × 每句话前面都重复一遍身份

同一段对话里说一次就够，它记得住。反复强调只占篇幅，把真正的要求挤没了。换新对话时再补一次就行。

## 做完自查

- ☐ 我的身份里写清了「干这行的人 + 干了多久 + 帮谁做什么」
- ☐ 我补了一句「不要什么」，比如不要术语、不要客套话
- ☐ 身份控制在两三句，剩下的篇幅留给了具体任务
- ☐ 我知道身份不改事实准确度，关键数字和引文我自己核过

# 06
# 给它看几个例子，比解释半天管用

你说十遍「写得活泼点」，不如直接给它三条你满意的旧文案。这篇讲例子从哪挑、放几个、放在哪，以及为什么给了例子反而更糟。

### 这篇讲什么

☐ 能用两三条样本把「我想要的风格」讲清楚，不再靠形容词

☐ 知道怎么挑例子、放几个，避免它学到一堆毛病

☐ 能让它只学风格、不照抄样本里的具体信息

## 01 形容词它听不懂，例子它看得懂

「专业一点」「接地气一点」这种话，每个人心里的尺子都不一样。它只能猜你的尺子，猜错很正常。

给它三条现成的样本，尺子立刻就清楚了。它会照着样本的句子长短、用词习惯、段落结构往下写，命中率高很多。

这个做法有个说法叫少样本提示（就是先给它几个样板，让它照着抄格式）。名字不用记，记住「给例子」就行。

- 一个像样的例子，胜过五句风格描述
- 凡是格式类的活，比如写周报、整理表格、改邮件，给例子几乎是最靠得住的办法
- 口头形容只能定方向，例子才能定标准

## 02 挑例子只看一条：你自己满意

别拿「差不多」「还能用」的东西凑数。例子就是标准，标准低了，产出一定跟着低。

数量上两到三个最划算。只给一个，它可能抓不准规律；超过五个，既占篇幅，又容易把它带偏。

三个例子最好覆盖不同情况。比如写商品标题，一个短的、一个带价格的、一个突出卖点的，它就明白哪里可以变。

- 例子统一成同一种格式，别一条表格一条长段落混着给
- 标清楚「示例一」「示例二」，最后写「现在按这个风格处理下面的内容」
- 反面例子也能用，但要说明「这条不要这样写，因为什么」

**示例：三条样本加一条新任务**

> 下面是我满意的三条朋友圈文案，先看它们的共同点：
>
> 示例一：周五下午三点，办公室的空调终于不吵了。
> 示例二：会议改到明天，我多睡了一小时，值。
> 示例三：楼下面包店打折，我买了三个，全当晚饭。
>
> 共同点：都很短，都有具体时间或数字，不喊口号，结尾不推销。
>
> 现在按这个风格，写三条关于「公司楼下新开咖啡店」的文案。只学风格，不要用上面出现过的人名和地点。

## 03 例子给错了，比不给还糟

最常见的问题是例子互相矛盾。一条正经一条搞笑，它学哪个都像半成品，最后四不像。

第二个问题是例子太长。你贴三篇两千字的文章，它抓不住重点，还容易整段照搬，你改起来比重写还累。

解决办法很土：把你真正想要的那一点圈出来。比如加一句「注意这三条的句子都在二十字以内」，它就有了准星。

- 给完例子先让它复述规律，确认看懂了再让它干活
- 样本里的具体人名、地点、数字，明确告诉它不要搬
- 挑不出好例子时，先让它帮你从旧文案里归纳出三条规律

## 能直接抄的模板

手上没有现成样本时，用这段让它自己先归纳

> 我下面给你 3 条我满意的样本，先别急着写新内容。
>
> 第一步：把这三条的共同点列出来，包括句子长短、语气、结构、有没有用数字和具体时间。
> 第二步：等我确认后，再按这些共同点写 3 条新内容。
> 第三步：新内容里不要出现样本里出现过的人名、地点、数字。
>
> 样本如下：
> 【把三条样本贴在这里】
>
> 新任务：【写在这里，说清给谁看、多长】

## 常见坑

× **拿「还行」的例子凑数**

例子是标准，标准低了产出就低，还会让你以为是 AI 不行。宁可只给两条你真满意的，也别塞五条一般的。

× **例子格式不统一**

一条表格一条长段落，它学不到规律，只会把两种写法混在一起。先统一成同一种格式，再交给它。

× **只给例子，不说任务**

它可能直接续写，或者把样本改两个字交给你。给完例子一定补一句「现在按这个风格处理下面这件事」。

× **样本太长，被整段照抄**

长样本最容易被搬。把样本压到三五句，或者明确写「只学风格和结构，内容必须全部重写」。

## 做完自查

☐ 我给的是自己真满意的样本，不是手边随便抓的

☐ 例子统一了格式，两到三条，标了「示例一」「示例二」

☐ 我说清了「只学风格，不要抄里面的具体信息」

☐ 我明确写了新任务是什么，没让它自己猜要干什么

# 07
# 让它一步一步想：思维链

碰到要算、要比、要多条件判断的活，直接问答案它很容易错。让它把中间步骤写出来，正确率会明显不一样，而且你能看出它错在哪。

### 这篇讲什么

☐ 知道什么活值得让它写步骤，什么活纯属浪费时间

☐ 能自己把步骤定死，而不是只丢一句「仔细想想」

☐ 学会抽查中间步骤，不被一个看着通顺的结论骗过去

## 01 它是边写边想，不写就没想

它是一边往外写一边往下想的。写出来的字，就是它的思考过程，没写出来的部分等于没有。

复杂问题让它一步到位，其实是在让它蒙。让它先把条件列出来，再一步步往下推，它每一步都能看着上一步写。

这个做法叫思维链（就是让它把中间的推理步骤一条条写出来，而不是直接甩个结论给你）。

- 算钱、比方案、多条件判断，这几类活最值得让它写过程
- 过程写出来你还能查。错了知道错在第几步，不用整段重来
- 简单活别用，纯属拖时间，回答还会变啰嗦

## 02 步骤你来定，比让它自由发挥稳

最省事的办法是加一句：「先分步推理，最后单独给结论。」加了就比不加强。

更稳的办法是把步骤定死。比如「第一步列已知条件，第二步说用哪个公式，第三步代入算，第四步检查单位」。

再让它把结论单独放在最后一行。这样你验收快，不用在几百字推理里找答案。

- 「一步一步来」比「仔细想想」有用，后者它经常只是嘴上答应
- 要求它把每一步的中间结果写出来，别只写「然后得出结果」
- 步骤定四到六步最合适，再多它就开始凑数

**示例提示词**

> 先别给答案，按下面四步来：
> 第一步：把题目里给出的条件一条条列出来，数字和单位都要带上。
> 第二步：说明要用哪个公式或哪条规则，为什么用它。
> 第三步：代入计算，每一步的中间结果都写出来。
> 第四步：单独一行写「结论：」，只放最终答案。
>
> 如果信息不够，在第一步就停下来问我，不要自己假设。
>
> 题目：【把题目贴在这里】

## 03 它写得通顺，不代表它算对了

写文案、改错字、翻译这类活，让它写推理只会更慢更啰嗦，一步能办的事没必要铺开。

还有一点得留意：步骤看着合理，不代表真对。中间某一步编了个数，后面全跟着错，而且读起来照样顺。

所以长推理要抽查。挑中间两三步自己验一遍，优先看数字、单位、日期这几步，这三样最容易出问题。

- 先看步骤有没有硬伤，再看结论。顺序反了容易被结论带跑
- 同一个题让它算两遍，两次结论不一样，说明这题它不稳
- 数字多的活，干脆让它写段代码跑一遍，比推理靠谱

## 能直接抄的模板

复杂问题的通用模板：先拆、再算、最后回头验

> 按这个顺序回答，不要跳步：
> 1. 复述：用一句话说清我问的是什么。
> 2. 条件：列出已知信息，缺什么就标「缺少」。
> 3. 思路：说明打算怎么解，为什么这么解。
> 4. 计算：一步一步算，每步都写出中间结果。
> 5. 检查：回头核对数字和单位，指出最可能出错的地方。
> 6. 结论：单独一行，只写最终答案。
>
> 如果信息不够，在第 2 步就停下来问我，不要自己假设。
>
> 我的问题：【写在这里】

## 常见坑

### × 只加一句「仔细想想」

这句话约束太弱，它经常还是直接给答案。改成「按 1、2、3 步写，最后单独给结论」，效果才稳得住。

### × 步骤写得越细越好

步骤太多它会凑数，编出没用的环节撑篇幅。四到六步够用，多出来的往往是废话，还容易把关键那步埋掉。

### × 看到步骤通顺就信了

推理过程读着顺，也可能是编的。数字、单位、日期这几步一定自己验一遍，别只看最后那个答案。

### × 简单活也让它长篇推理

改错字、调格式、换个说法，这类活写推理只会拖时间。一句话能办的事，直接说，别加流程。

## 做完自查

- ☐ 要算、要比、要多条件判断时，我让它写了中间步骤
- ☐ 我把步骤定成了具体的几条，不是只说「仔细想想」
- ☐ 我让它把结论单独放在最后一行，方便核对
- ☐ 我抽查了中间的数字和单位，没有直接抄结论

# 08
# 让它按你要的格式输出

同一件事，让它输出表格、清单还是
JSON，结果差很远。这篇教你把「长什么样」说清楚，省掉九成返工。

### 这篇讲什么

☐ 能写出让 AI 一次就给对格式的指令

☐ 会用两行示例代替一堆形容词去描述格式

☐ 知道批量处理时怎么分批跑、怎么快速发现它漏了几条

## 01 它为什么总给你一大堆废话

你不说格式，它就按最常见的样子写。最常见的回答长什么样？开头一句客套，中间几段解释，结尾再来一段总结。

这不算它不听话，是它没得选。它只能猜「下一个字最可能是什么」，你不给框，它就往最顺的路上走。

所以别写「简洁一点」「专业一点」这种话。它没法知道你要多简洁。你得说清几行、几个字段、什么顺序。

- 要求越像「样子」，它越容易照做：几行、几列、字段叫什么、要不要表头
- 「简洁」「专业」「清楚」这类词没法量化，等于没说
- 它天生爱加开场白和总结段。要么明说不要，要么写死「只输出表格，不要其他文字」

## 02 最快的办法：给它看一个样子

描述格式很难，示范格式很容易。你直接给它两行样例，它照着抄的能力很强。

样例里最好把边界情况也放进去：没数据怎么填、不确定怎么写、太长怎么截。它看到就会照做。

字段名要用你后面真要用的名字。别写「名称」，写「客户名」；别写「金额」，写「金额（元）」。

- 样例给两行就够。一行看不出规律，三行以上它容易抄串
- 让它把不确定的地方写成固定字样，比如「未知」，比留空好，留空你后面很难筛

**示例：用两行样例说清格式**

> 按下面的格式输出，只输出表格，不要开场白和总结：
>
> 序号 | 客户名 | 需求 | 优先级 | 金额（元）
> 1 | 张三 | 报价单 | 高 | 12000
> 2 | 李四 | 网站改版 | 未知 | 未知
>
> 规则：原文没提到的写「未知」，不要自己推测。

## 03 一次只跑一小批，边跑边看

别一上来就扔给它两百条。先给五条试跑，看格式对不对，再放量。

条数一多它容易偷懒：中间几条合并、字段少一个、最后几条干脆不写。这是常见现象，不是你运气差。

稳妥的做法是每次二十到三十条，跑完自己扫一眼行数。行数不对，说明它漏了。

- 让它按编号输出，跑完对一下编号有没有断，比逐条读快得多
- 要粘进表格软件就让它出CSV（用逗号分列的纯文本格式）。别让它画带竖线的表，粘过去容易错列

## 能直接抄的模板

批量整理信息时用这个，把方括号里的换成你的内容

> 把下面这些内容整理成表格，只输出表格，不要任何解释。
>
> 字段（顺序固定）：编号 | 客户名 | 需求 | 优先级（高/中/低） | 金额（元）
>
> 规则：
> 1. 编号按我给的原样保留，不要重新编号。
> 2. 原文没提到的写「未知」，不要推测。
> 3. 金额只留数字，不要带「元」和逗号。
> 4. 优先级判断标准：三天内要结果=高，一个月内=中，其余=低。
>
> 格式示例：
> 1 | 张三 | 报价单 | 高 | 12000
> 2 | 李四 | 网站改版 | 未知 | 未知
>
> 下面是内容：
> [粘贴你的原文]

## 常见坑

### × 只说「格式整齐点」

它不知道整齐是几列几行。要说清字段名、顺序、数量；说不清就给它看两行样例，比形容词管用十倍。

### × 一次让它处理几百条

条数一多它就偷懒，中间合并、末尾漏写。每次二十到三十条，跑完先数行数，不对就重跑那一段。

### × 让它输出好看的表格样式

带竖线的表格粘进 Excel 经常错列，还得手工拆。要粘表格就直接要 CSV，用逗号隔开。

### × 没说清没数据时怎么写

不说它就自己编一个填上，看着还挺合理。提前定好「未知」这类固定写法，后面才好筛出需要补的。

## 做完自查

☐ 我的指令里写清了字段名、顺序和数量，没用「简洁」「专业」这种词

☐ 我给了至少两行样例，包括没数据时怎么填

☐ 批量处理时我分批跑，并且数过输出行数对不对

☐ 要粘进表格软件的内容，我让它出 CSV 而不是画表格

第二部分

# 写代码

第 09 — 14 讲

09

# 让 AI 写代码的正确姿势

它写代码很能干，但你不说清环境，它就给你一段跑不起来的代码。这篇讲怎么把需求说成它能照着做的样子。

---

### 这篇讲什么

- ☐ 能把需求说成一段 AI 照着就能写的说明
- ☐ 知道怎么把报错贴回去，让它一轮改对
- ☐ 会先要一个最小能跑的版本，再一步步加功能

## 01 先交代环境，别让它猜

同一段需求，在 Python 3 和老版本里写法可能完全不同。你不说，它只能按最常见的写。

所以开口第一句就把环境说清：什么语言、什么版本、装了什么库、在哪儿跑。比如「Python 3.10，只用标准库，在 Windows 上跑」。

还要说清输入输出。数据从哪来，是读文件还是手输；结果打印到屏幕，还是写成文件。这些它猜不准。

- 库能少用就少用。指定「只用标准库」，能省掉一大半装环境的时间
- 说清运行方式：命令行跑、双击跑、还是在网页里跑，三种写法差别很大
- 如果这段代码以后要给别人用，先说清你希望它怎么被调用，写法会不一样

## 02 一次只让它做一件事

新手常犯的错：一次让它写一个完整系统。几百行代码，一处不对你根本找不到在哪。

正确顺序是先跑通最小的一步。比如先「读进这个文件，打印前五行」。跑通了，再加筛选，再加导出。

每加一步就让它只改这一处，别让它重写整段。改得少，你才看得懂改了什么。

- 让它把代码分成几小段，每段前面一行注释说明干什么
- 复杂逻辑先让它写一句思路，你确认了再写代码。思路错了，代码再漂亮也白搭
- 让它顺手给一份能直接跑的测试数据，省得你自己造

## 03 报错原样贴回去，别只说「跑不了」

「报错了」这三个字它没法用。它需要看到完整的报错文字，尤其是最后一行和那个行号。

贴的时候三样一起给：完整报错、相关代码、你做了什么操作。少一样它就开始猜，猜错你还得再来一轮。

改完还是错，就把新的报错再贴一遍，并且说「上次那个改法没解决」。它不会记得你没说的话。

- 报错里最有用的是最后一行，那行写着错在哪、什么类型
- 别自己转述报错，转述会丢关键信息，原样复制粘贴最快
- 改完让它写一句「这个改法假设了什么」，以后出错你才知道从哪查起

**示例：把报错贴回去的说法**

> 我按你给的代码跑了，还是报错。
>
> 报错原文：
> [整段粘贴，包括最后一行]
>
> 我跑的命令：python clean.py
> 我的代码：
> [粘贴相关代码]
>
> 请只改出错的那几行，不要重写整段。改完告诉我改了哪里、为什么这么改。

## 能直接抄的模板

要代码时照这个填，越具体它写得越准

> 帮我写一段代码。
>
> 环境：Python 3.10，Windows，只用标准库，不用 pandas。
> 输入：同目录下的 orders.csv，UTF-8 编码，第一行是表头，列有
> 订单号、客户、金额、日期。
> 要做的：
> 1. 读进这个文件。
> 2. 只保留金额大于 1000 的行。
> 3. 按客户分组，算出每个客户的总金额。
> 4. 按总金额从大到小排，打印前 10 行。
> 输出：直接打印到屏幕，不要写成文件。
> 要求：金额列可能有空值，遇到就跳过这一行并计数，最后打印「跳过 N
> 行」。关键步骤加一行中文注释。
>
> 先只给我代码，不要解释。如果我哪里说得不清楚，先问我一句再写。

## 常见坑

### × 只说「帮我写个脚本」

它不知道语言、版本、数据长什么样，只能给你一个通用模板，你还得大改。把环境和输入输出说清，一次就能用。

### × 一次要一整个系统

几百行代码里一处小错就让整个跑不起来，你很难定位。先要一个能跑的最小版本，跑通再加下一步。

### × 报错只说「跑不了」

它拿不到信息只能瞎猜，来回好几轮。把完整报错、相关代码、你的操作三样一起贴，通常一轮就能改对。

### × 让它重写整段来修一个 bug

整段重写会把已经跑通的部分也改掉，可能带出新问题。让它只改出错的那几行，改完自己跑一遍。

## 做完自查

☐ 我在指令里写清了语言、版本、能不能用第三方库

☐ 我说清了数据从哪来、结果要输出到哪

☐ 我先要最小能跑的版本，跑通之后才加功能

☐ 贴报错时我贴的是原文，没有自己转述

# 10
# 让 AI 帮你审代码

把代码丢给它问「有问题吗」，它多半挑几个小毛病就完事。这篇讲怎么让它按你的重点去查，并给出能直接改的建议。

### 这篇讲什么

- ☐ 能让它按你关心的几类问题去查，而不是随便翻翻
- ☐ 知道怎么让它把问题分级，先修要紧的那几条
- ☐ 会用它看懂自己接手的旧代码在干什么

## 01 先让它讲明白这段在干嘛

拿到一段别人写的代码，你连它干什么都不清楚，就别急着让它挑错。先让它用大白话讲一遍。

讲完你再问：「它有没有做到你刚才说的这些？哪里没做到？」这样它挑的错才靠谱。

顺序反了容易出问题。直接问「有没有 bug」，它会给一堆看着专业、其实不影响运行的建议，你反而更乱。

- 让它用三句话说清输入、处理、输出。说不清，通常说明代码本身就有问题
- 让它标出哪几行最关键，你重点看那几行

## 02 给它一份检查清单

「帮我看看」太宽，它只能挑最显眼的。你给几类具体问题，它就往这几类里查。

常用的四类：会不会崩、会不会算错、有没有安全隐患、跑起来慢不慢。一次查两类就够。四类一起上，它每条都写得很浅。

慢不慢主要看一处：循环里有没有反复查数据库、反复读文件。这种写法数据一多就卡住。

- 会不会崩：空值、除以零、文件不存在、类型不对
- 会不会算错：边界值、取整方式、日期跨月跨年
- 有没有隐患：密码写死在代码里、查询语句用字符串拼出来

**示例：把检查范围收窄**

> 只查这两类问题，其他先不用管：
> 1. 会不会崩：空值、除以零、文件不存在、类型不对。
> 2. 会不会算错：边界值、取整方式、日期跨月跨年。
> 每条都要写清：在第几行、什么情况下会出问题。

## 03 让它分级，别一次给你五十条

不设上限，它会把你没问的都写上，看得你头大。先说清：最多给五条，按严重程度排。

分级标准让它自己定也行，但要它说明理由。你只看前两条，剩下的先放着。

每条建议都要它给出改法，最好直接给改后的那几行。只说「建议加异常处理」，等于没说，你还得自己想怎么写。

- 让它分三档：必须改（会出错）、建议改（以后难维护）、可以不改（写法习惯）
- 写法习惯类的一律先跳过，改多了容易引入新问题
- 说清使用场景：「每天跑一次，一万行左右」，它才知道什么算慢

## 能直接抄的模板

审代码时用这个，把场景和重点换成你的

> 下面这段代码请帮我审一遍。
>
> 背景：这是我维护的一个脚本，每天跑一次，数据量大约一万行，Python 3.10。
>
> 请按这个顺序做：
> 1. 先用三句话说明这段代码在干什么（输入、处理、输出）。
> 2. 然后只查这两类问题：会不会崩、会不会算错。
> 3. 最多列 5
> 条，按严重程度排序，每条写清：在第几行、什么问题、什么情况下会出问题。
> 4. 每条都给改后的代码，只贴改的那几行，不要重写整段。
>
> 写法习惯、命名、注释这类问题先不用提。
>
> 代码如下：
> [粘贴]

## 常见坑

### × 直接问「有没有问题」

问题太宽，它会挑一堆不影响运行的小毛病，把真正的坑淹掉。先说清你最怕哪种错，让它只查那几类。

### × 只贴一小段

变量从哪来的、这个函数被谁调用，它看不到就只能猜。至少把整个函数和它的调用处一起贴上去。

### × 相信它说的「这里不安全」

它经常把正常写法说成漏洞，也可能漏掉真问题。每条你都要问一句「什么情况下会出问题」，答不上来就先放着。

### × 把它给的建议全改一遍

几十条一起改，很容易改出新问题，而且你分不清是哪条改坏的。一次只改一条，改完跑一遍再动下一条。

## 做完自查

☐ 我先让它讲清这段代码在干什么，再让它挑错

☐ 我告诉了它运行场景和数据量，它才知道什么算慢、什么算崩

☐ 我限制了条数，并要求它按严重程度排序、给出行号

☐ 它指出的每条我都问过「什么情况下会出问题」，没有照单全收

# 11
# 代码跑不起来，怎么让 AI 帮你查

报错不是让你怕的，是让你抄的。这篇讲怎么把报错原文和出问题的那几行交给 AI，让它一次只改一处。

### 这篇讲什么

☐ 能把报错整理成 AI 看得懂的材料，不用来回瞎猜十几次

☐ 知道该贴哪几行、不该贴哪几行

☐ 能判断它给的改法靠不靠谱

## 01 先把报错原文抄给它，一个字都别改

程序崩了会吐一大段英文，这叫报错信息（就是告诉你哪一行、因为什么停下来的那段话）。别翻译，别总结，整段复制。

很多人只写一句「我的代码报错了，怎么办」。这句话里没有一行有用信息，AI 只能瞎猜，猜完你试，来回十几次，半小时就没了。

报错里最值钱的是文件名和行号。有这两个，它能直接指到你出问题的那一行。

- 最后一行通常最关键，别只贴第一行
- 贴之前把密钥、内网地址换成 XXX

## 02 代码别整份贴，贴能跑起来的那一小段

一个项目几千行，全贴进去它看不完，也看不准。哪个函数（一段能反复调用的小程序块）出的事，就贴那几行。

实在不知道是哪儿的，就一步步往回删。删到不报错了，问题就在刚删掉的那块。这笨办法比让 AI 猜快。

顺手说清三件事：你想让它干什么、它实际干了什么、你用的哪种语言。

- 写上「我以为输出 5，结果输出 None」，比十行描述都管用

- 三五行能复现问题的例子最值钱

**最小复现就长这样**

```
rows = [1, 2, 3]
print(rows[5]) # IndexError: list index out of range
```

## 03 让它一次只改一处，改完你要能讲出来

最怕它一口气改八行，然后说「这样应该就好了」。好了你不知道为什么好，坏了更不知道坏在哪。

直接要求：一次只给一个改法，说清原来错在哪、改完会变成什么。你改一行，跑一次，看结果再问下一句。

跑通之后再让它用三句话解释这段代码现在怎么工作的。你讲不出来，下次同样的坑还得再踩一遍。

- 让它标出改动的行，别给整段重写
- 同一个问题试两次不通就换思路，别耗半小时

## 能直接抄的模板

报错排查用这个，方括号里的换成你的内容

```
我的代码跑不起来了，按下面的信息帮我查：
【语言】Python，用的是 pandas
【我想干什么】把表格里空着的行删掉
【实际干了什么】直接报错退出，一行都没删
【完整报错】
（贴原文，从第一行贴到最后一行，别改）
【出问题的代码】
（贴能复现问题的那几行，不要贴整个项目）

请这样回答：
1. 先用大白话说这段报错什么意思，别用术语。
2. 指出最可能出问题的是哪一行，为什么。
3. 一次只给一个改法，说清改完有什么变化。
4. 信息不够就先问我，不要猜。
```

## 常见坑

× **只说「跑不起来」**

没报错、没代码、没语言，AI
只能回一堆通用建议。最少给三样：完整报错、出问题的那几行、你用的语言。

× **把整个项目打包丢过去**

几千行它看不完，只能挑着看，很可能挑错。先缩到能复现问题的最小的那几行，三五行最好。

× **让它「顺手优化一下」**

一次改八处，跑通了不知道哪处起了作用，跑不通不知道回退哪处。一次只改一处，改完立刻跑。

× **不看行号就照抄**

它说的行号是按你贴的那段算的，不是你文件里的真实行号。对不上就自己数，别删错行。

## 做完自查

☐ 报错我贴的是原文，没自己总结

☐ 贴的代码能单独跑起来复现问题

☐ 它给的每个改法我都能说出为什么

☐ 改完我先跑一遍，确认原功能还在

# 12
# 让 AI 帮你把烂代码理顺

烂代码不一定跑不动，但一改就出事。这篇讲怎么让它先读懂再动手，一次只理顺一个函数，改完你还能对得上账。

### 这篇讲什么

☐ 能先把看不懂的代码读懂再动手

☐ 知道怎么限定范围，一次只动一个函数

☐ 能看懂它的改动，敢判断哪处该收、哪处退回

## 01 先让它读，不许改

拿到看不懂的代码，第一句话别是「帮我优化」，而是「先讲讲这段在干什么，一行行说」。

这一步花不了两分钟，但能救你半天。它讲的时候，你会发现自以为的逻辑和实际的对不上。

它要是讲错了，正好说明这段代码有多绕。连它都要猜，人更得先读懂再动手。

- 让它标出「这几行我拿不准」，别硬编解释
- 顺手问一句：这段最可能在哪儿出错

## 02 一次只理顺一个函数

理顺（行话叫重构，意思是不改功能，只把代码整理得更好读）最忌讳整文件重写。一次动一个函数，跑通再动下一个。

给它定三条底线：功能不许变、对外的名字不许改、不许顺手加新功能。不写清，它一定给你加戏。

函数太长就让它先拆。超过五十行、缩进拐了三道弯的，基本都该拆成几个小函数。

- 先挑你最常改的那个函数下手，收益大风险小
- 名字也让它改：func1、tmp 这种换成看得懂的说法

## 03 要「改前改后」对照，别要一整段新代码

让它按三段输出：原来长这样、现在长这样、为什么这么改。你一行行对着看，才知道它动没动你的逻辑。

看到它把判断顺序换了、把异常吞了、把十行循环压成一行，就要多问一句。什么情况下行为会不一样？

改完自己跑一遍原来的功能。有测试就跑测试，没测试就手动点三下关键路径。

- 它说「这样更优雅」时，追问具体好在哪，答不上就别改
- 一次提交只放一个函数的改动，出事好回退

**让它按这个格式给你**

```
原来：
def f(a, b):
if a == None:
return b

现在：
def add_or_fallback(a, b):
if a is None:
return b

为什么改：名字看不出用途，判断写法不严谨。
行为变化：没有。
```

## 能直接抄的模板

想理顺一段代码时，整段复制给它

> 下面这段代码我不太敢改。先别动它，按顺序来：
> 【第一步】逐行讲一遍它在干什么，用大白话。拿不准的行直接标出来，不要硬猜。
> 【第二步】指出最可能出问题的地方，最多三条。
> 【第三步】只挑第一个问题，给我一个改法，按这个格式：
> 原来：……
> 现在：……
> 为什么这么改：……
> 会有什么行为变化：……
> 【限制】
> 1. 功能不许变，对外的名字不许改。
> 2. 不许顺手加新功能，也不许顺手修别的 bug。
>
> 代码：
> （贴在这里）

## 常见坑

### × 一句「帮我重写一下」

它会连功能一起改。看着更短更漂亮，实际行为已经变了。要说清：功能不许变，只改可读性。

### × 不留备份就覆盖

先把原文件复制一份，或者提交一次。理顺时把能跑的代码改成不能跑的，是最常见的事故。

### × 修 bug 和改风格混在一起

分两次做。混在一次里，出问题你分不清是逻辑错了，还是整理时手滑了。

### × 它改的名字你不认识

名字漂亮但你不知道啥意思，等于没理顺。不懂就问，问完改成自己看得懂的说法。

## 做完自查

☐ 改之前我留了备份，或者提交了一次

☐ 一次只动一个函数，动完就跑一遍

☐ 它说的每处改动我都能讲出为什么

☐ 功能没变，对外的名字没改

# 13
# 让 AI 写测试，省一半时间

测试不是给领导看的，是给你下次改代码时兜底的。这篇讲怎么让它先列清单再写测试，专挑会出事的地方测。

### 这篇讲什么

☐ 能先要一份该测什么的清单，再让它动笔

☐ 知道坏数据和边界才是重点，不再只测顺路的情况

☐ 跑出红的（失败）时能分清是代码错还是测试错

## 01 先让它列清单，别急着写代码

测试（就是几行自动检查的代码，跑一下就知道功能有没有被改坏）。测试写得好不好，全看测了什么。第一步不是写，是列。

让它先输出一张表：这函数会被怎么用、正常情况有哪些、哪儿容易出岔子。看表比看一百行测试有用。

清单里缺什么你补什么。你最清楚自己上周在哪儿踩过坑，就让它专门为那个坑写一条。

- 正常情况占三成，坏数据和边界占七成
- 边界值（就是卡在极限上的数，比如空字符串、0）要单列

## 02 坏数据才是重点

大部分人写的测试只测「输入 1 和 2，输出 3」。这种测试永远绿，也永远抓不到问题，等于白写。

真正会出事的是这些：空值、负数、超长文本、格式不对的日期、重复提交、断网。让它按这份清单一条条写。

每条测试都要有断言（就是那句「结果应该等于多少」的判断，对不上就算失败）。没断言的测试等于没写。

- 一条只测一件事，名字要能读出它在防什么

**生成的测试长这样（Python 例子）**

```
import pytest

def test_normal_input():
assert parse_age("18") == 18

def test_empty_raises():
with pytest.raises(ValueError):
parse_age("")
```

## 03 跑一遍，让它解释红的那些

测试写完一定要跑。跑出红的（失败的）才是好消息，说明它真抓到了东西。

把失败的输出整段贴回去，问它：是代码错了，还是测试写错了。这两种处理完全相反，必须让它明确回答一个。

最怕它为了让测试变绿，偷偷改断言的期望值。跑之前先看一眼它有没有动 assert 那一行。

- 测试文件单独放，别和业务代码混在一个文件里
- 每次改完代码先跑一遍，十秒钟的事

## 能直接抄的模板

让它写测试时，用这个开头

```
帮我给这个函数写测试，先别写代码：
【第一步】先列一张表：这函数会被怎么调用、正常情况有哪些、坏数据有哪些、边界在哪。
【第二步】我确认后按清单一条条写，每条只测一件事。
【第三步】每条都要有明确断言，不许用「大概对就行」的判断。
【限制】
1. 不许改动被测函数本身。
2. 跑不过时先说清是代码错还是测试错，别改期望值凑绿。

函数：
（贴在这里）
```

## 常见坑

× **只测顺路的情况**

「输入 1 和 2 输出 3」这种测试永远绿，也永远抓不到问题。空值、负数、超长、格式不对才是重点。

× **让它改测试来凑绿**

跑不过时它很可能悄悄把期望值改掉，测试绿了，问题还在。跑之前看一眼断言那行动没动。

× **生成一大堆但一次没跑**

没跑过的测试等于没写，还可能自带错误。写完立刻跑，红的才是好消息。

× **测试写得自己都看不懂**

看不懂的测试，下次不敢改也不敢信。一条只测一件事，看不懂就让它重写。

## 做完自查

☐ 我先要了清单，确认过该测什么才让它写代码

☐ 坏数据和边界值我单独列了，不只测正常情况

☐ 写完立刻跑了一遍，不是直接收下

☐ 它没动被测的代码，也没偷偷改断言

# 14
# 让 AI 补注释和文档

老代码没注释，三个月后自己都看不懂，新人更不敢动。这篇教你让 AI 把注释和文档补齐，还不会乱改你的代码。

### 这篇讲什么

☐ 能让 AI 给一段没注释的代码补上注释，逻辑一行不动

☐ 知道注释该写什么、不该写什么，不会被废话注释刷屏

☐ 能用 AI 生成 README 和函数说明，自己改两笔就能交出去

## 01 先让它讲一遍，再让它写进去

别一上来就说「帮我加注释」。先让它把这段代码讲一遍：干什么用的、输入什么、输出什么、哪几行最容易出错。

它讲错了，说明它没看懂。它没看懂，写的注释也一定是错的。它讲对了，你再让它把刚才讲的内容整理成注释，贴回代码里。

这个顺序很关键。讲解的时候它必须把逻辑从头到尾走一遍，比直接憋注释靠谱得多。你顺手还把这段代码审了一遍。

- 先讲解，后写注释，中间你自己判断它到底看懂没有
- 讲解里冒出「大概」「应该是」这类词，说明这段它没把握，重点盯
- 一次只喂一个文件或一个函数，别把整个项目扔进去

## 02 注释写原因，不写翻译

好注释只写代码本身看不出来的东西。比如为什么这么写、这个参数有什么坑、这段跟哪个业务有关。

坏注释是把代码用中文再念一遍。比如 i 加一，它写「i 增加 1」。这种注释除了占地方，一点用都没有，还容易过期。

它默认就爱复述代码，你不说它一定那么干。所以提示词里要写死一句：不许复述，只写原因和注意事项。

- 函数开头写清三件事：干什么、参数什么意思、返回什么
- 有坑的地方单独标一句，比如「这里必须判空，上游可能传空值」
- 一眼就懂的赋值、循环、打印，不用加注释

**示例提示词**

> 下面是一段代码。请按顺序做：
> 1. 先用三句话说清这段在干什么、输入什么、输出什么。
> 2. 指出最容易出错的一到三行，说明为什么。
> 3. 等我确认后，再给每个函数加注释。
> 4. 注释只写「为什么这么写」和「有什么坑」。不许把代码翻译一遍。
> 5. 不要改动任何一行代码，只加注释。

## 03 文档最容易瞎编，先把真话喂给它

最容易翻车的是让它写 README（项目说明书，别人打开代码库先看的就是它）。它手里没有真信息，基本靠猜。

它没见过你的项目怎么启动、跑在哪个端口、依赖怎么装。于是照着常见套路写一套，看着很像，实际跑不通。

正确做法是先把真实信息贴给它：启动命令、目录结构、要配哪些环境变量。你给真话，它才写得对。

- 先贴真实的启动命令和目录结构，再让它动笔
- 不确定的地方让它写「待补充」，不许自己编一个填上
- 写完对着实际环境跑一遍，命令跑不通就是编的

## 能直接抄的模板

给整个模块写说明文档时用这段，复制改两处就能用

> 下面是我的项目目录结构和启动命令，请据此写一份 README。
> 目录：
> （粘贴目录树）
> 启动命令：
> （粘贴你真实在用的命令）
>
> 要求：
> 1. 只写我给你的这些信息，不要补充你没看到的东西。
> 2. 不知道的地方一律写成「待补充」，绝对不要编一个看起来合理的答案。
> 3. 结构固定为：这个项目是干什么的、怎么装、怎么启动、每个文件夹负责什么。
> 4. 全文用短句，不要用「强大」「高效」这类空话。

## 常见坑

### × 让它顺手把代码也改了

你说「加点注释」，它可能顺手重构一遍，逻辑就悄悄变了。提示词里写死「只加注释，不改代码」，拿到结果先用对比工具看一眼改动了哪几行。

### × 整份项目一次丢进去

文件一多它就顾此失彼，前面的看完后面忘了。一次一个文件，最多一个模块，写完再换下一个。

### × 注释塞满废话

每行都加注释，代码反而更难读。只给函数开头、复杂判断、有坑的地方加，普通赋值和循环放过。

### × README 直接发出去

里面的安装命令、端口号、配置项大概率是编的。照着真机跑一遍，跑不通的自己改掉再发。

## 做完自查

- ☐ 我让它先讲一遍代码，确认它看懂了再让它写注释
- ☐ 提示词里写了「只加注释，不改代码」，拿到结果我对比过
- ☐ 注释写的是原因和坑，不是把代码翻译一遍
- ☐ README 里的每条命令我都真跑过一遍

第三部分

# 接入与工程

第 15 — 21 讲

# 15
# 调 API 的第一步：把模型接进你的程序

在网页上聊天只能一次一句，接进程序才能批量干活。这篇带你走完第一次调用：密钥放哪、请求怎么发、结果怎么看。

---

### 这篇讲什么

☐ 知道密钥该放在哪、绝不能放在哪，不会一提交就泄露

☐ 能发出第一次请求，把模型的回答打印在自己的屏幕上

☐ 看懂返回结果里哪部分是回答、哪部分是这次花掉的字数

## 01 API 就是把这层包装拆掉

你在网页上跟模型聊天，本质也是发请求。只是网页帮你把请求包好了，你只负责打字。

API（程序之间说话的接口）就是把这层包装拆掉。你自己发一段规定格式的文字过去，它回你一段规定格式的文字。

所以接 API 不难。要搞清的只有三件事：地址填哪、密钥放哪、请求长什么样。这三件搞清了，换别家也是同一套路。

- 地址：服务商给你的那个网址，一般叫 base_url
- 密钥：一长串字符，相当于账号密码，泄露了别人花你的钱
- 格式：主流服务商的请求格式差不多，学会一家换别家很快

## 02 第一版代码，只求跑通

第一版别写复杂。目标只有一个：发一句话，把回答打印出来。跑通了再往上加东西。

常见失败就三种：密钥写错了、地址多了或少了斜杠、字段名拼错。报错信息基本会直接告诉你哪一个。

跑通之后先别急着封装成函数。用最土的写法连跑三次，确认稳定了再整理，能省掉很多来回。这一步的意义是把变量一个个排除掉：跑通一次，你就知道地址、密钥、格式该长什么样。

- 先看状态码：200 是成功，401 多半是密钥不对，429 是请求太频繁
- 取回答用 choices[0].message.content，别漏掉那个 [0]
- 把 usage 记下来，跑批量任务之前先估一下总量

**示例代码（Python，把三处中文占位换成你自己的）**

```
import requests

resp = requests.post(
"你的服务商地址/v1/chat/completions",
headers={"Authorization": "Bearer 你的密钥"},
json={
"model": "服务商给你的模型名",
"messages": [{"role": "user", "content": "用一句话解释什么是缓存"}],
},
timeout=60,
)

print(resp.status_code)
print(resp.json()["choices"][0]["message"]["content"])
```

## 03 返回的东西里，你只看两个地方

返回的是一段 JSON（一种带花括号的数据格式，程序读起来很方便）。你不用全看，只关心两个地方。

choices 里面装的是回答。那个 content 字段就是模型说的话，取出来直接能用。

usage 里面是这次花了多少字，输入一段、输出一段分开算。跑批量之前先看一眼这个数，心里有底。

- 先看状态码再看内容，状态码不对时内容里全是错误说明
- 把返回内容整个存一份日志，出问题时有东西可查
- 同一次请求里，带的历史消息越长，花掉的字数越多

## 能直接抄的模板

换新服务商时先跑这段提示词，让它给你一个最小脚本

> 请帮我写一个最小的 Python 调用示例，要求：
> 1. 只用 requests，不要用任何第三方 SDK。
> 2. 服务商地址、密钥、模型名都从环境变量读，不要写死在代码里。
> 3. 发一句「你好」，打印状态码和返回的 content。
> 4. 必须加 timeout。
> 5. 告诉我 401、429、500 这三种状态码分别该去查什么。
> 6. 不要写重试逻辑，不要封装成类，就要一个能直接跑通的脚本。

## 常见坑

× **密钥写死在代码里**

一提交到代码仓库就等于公开了。放环境变量（系统里存的一个值，程序能读到，但不会出现在代码里），或者放本地配置文件，再把那个文件加进忽略列表。

× **把真密钥贴给 AI**

让 AI 帮你调代码时，把密钥换成「你的密钥」四个字再贴。它不需要真密钥也能帮你把代码改对。

× **以为一次请求就是一轮对话**

API 没记性，它不知道你上一句说了什么。要多轮对话，就得把之前的每条消息都带上，一起发过去。

× **不设超时**

网络卡住时程序会一直干等，整个任务就挂在那儿。所有请求都加超时，一般 60 秒够用，超长文本可以放宽。

## 做完自查

- ☐ 密钥没有出现在代码里，也没进过代码仓库
- ☐ 我先跑通了最小示例，看到模型的回答才继续往下写
- ☐ 每个请求我都设了超时
- ☐ 我知道多轮对话要自己把历史消息带上

# 16
# 一个字一个字往外蹦：流式输出

等整段回答出来要十几秒，屏幕一直转圈，人以为卡死了。流式输出让字一个个冒出来，第一秒就能看到东西。这篇讲怎么接。

### 这篇讲什么

☐ 知道流式输出是什么，也明白为什么等待感会消失

☐ 能在请求里打开流式，把一小片一小片的碎片拼成完整回答

☐ 避开三个坑：空片报错、不判断结束、断线全丢

## 01 为什么等一整段出来这么难受

默认情况下，模型把整段话写完才一次性返回。写三百字可能要十几秒，这十几秒你的界面完全没反应。

流式输出（服务端边写边发，你边收边显示）就是把这个过程摊开。它写一个字，你收到一个字，你看到的就是它正在写的进度。

总时间一点没变短，但你的感觉完全不同。第一秒屏幕上就有字，人就不会觉得卡，也不会反复去点刷新。所以这不是性能优化，是体验优化。

- 总耗时一样，但第一屏出现的时间从十几秒变成一秒内
- 打字机效果不是装饰，它在告诉用户「我在干活」
- 长回答、长文档、写代码，这三种场景收益最大

## 02 打开它，其实就是加一个开关

在请求里加一个字段，告诉服务端「别攒着，写一点发一点」。字段名一般是 `stream`，值设成 `true`。

打开之后，返回的不再是一整段 JSON，而是一行一行的数据。每行开头带着 `data:` 这几个字符。

每行里面是一小片回答，通常只有几个字。你要做的就是把这些小片按顺序拼起来，拼成完整的一段。

- 请求里要开，客户端读取时也要设成流式，两处都得改
- 开头几片经常是空的，里面只有角色信息，没有文字
- 最后会来一行结束标记，看到就停，别一直干等

**示例代码（Python，边收边打印）**

```
import requests, json

resp = requests.post(
"你的服务商地址/v1/chat/completions",
headers={"Authorization": "Bearer 你的密钥"},
json={
"model": "服务商给你的模型名",
"messages": [{"role": "user", "content": "写一段一百字的自我介绍"}],
"stream": True,
},
stream=True,
timeout=60,
)

full = ""
for line in resp.iter_lines():
if not line:
continue
text = line.decode("utf-8").removeprefix("data: ")
if text.strip() == "[DONE]":
break
chunk = json.loads(text)
piece = chunk["choices"][0]["delta"].get("content", "")
full += piece
print(piece, end="", flush=True)

print("\n总字数：", len(full))
```

## 03 拼的时候容易出岔子

每一片可能只有半个词，甚至只有一个标点。别急着对单片内容做判断，先老老实实拼成一整段再说。

取内容的时候要给个默认值。有些片里根本没有文字字段，你直接取会报错，程序当场停掉，前面收的全白费。

收尾那行不是回答，是个特殊标记。不停下来，循环会一直空转，或者一直挂着不返回。挂住的时候还不报错，最难查。

- 单片内容不能拿去做关键词判断、翻译、格式校验
- 网络断了流就断了，界面上要保留已经收到的部分
- 边收边把内容存下来，每收一片就刷新一次输出缓冲

## 能直接抄的模板

让它帮你写一个稳一点的流式接口

> 请帮我写一个流式输出的调用示例，要求：
> 1. 用 requests 流式读取，边收边打印，不要等全部收完。
> 2. 每收到一片立刻输出并刷新缓冲，做出打字机效果。
> 3. 跳过空片和只有角色信息的片，遇到结束标记就停止。
> 4. 把收到的文字同时累加到一个变量里，结束时打印总字数。
> 5. 加异常处理：网络中断时保留已经收到的部分，并提示「回答中断了，可以重试」。
> 6. 不要用第三方 SDK，不要封装成类。

## 常见坑

### × 只开了一半

请求里写了 stream 为 true，但客户端读取时没设成流式，程序照样会等所有数据到齐才返回，等于白开。两处都要改。

### × 对每一片单独处理

一片可能只有半个词，单独拿去匹配关键词、做翻译、做校验，全是错的。先拼成完整句子再处理。

### × 不做结束判断

流不会自己停，你得看到结束标记主动退出循环。少了这一句，程序可能一直挂着不返回，排查起来还很难找。

× **断线就全丢**

用户已经看了三百字，网一断界面清空，体验极差。边收边存，断了就提示可以重试，已经收到的内容留着。

## 做完自查

- ☐ 请求里开了流式，客户端也设成了流式读取
- ☐ 我是把碎片拼成完整回答之后，才做后续处理的
- ☐ 代码里判断了结束标记，也处理了没有文字的空片
- ☐ 网络中断时，用户已经看到的内容不会消失

# 17
# 出错了怎么办：重试和兜底

AI 出错不是意外，是常态。这篇教你把报错分成三类，各自配一套重试和兜底的办法，让活能稳稳跑完。

### 这篇讲什么

☐ 能一眼分清是网络抽风、格式跑偏，还是这事它根本干不了

☐ 会写重试逻辑：错几次、隔多久再试、什么时候必须放弃

☐ 每处调用都有兜底方案，不至于一崩就整个流程卡死

## 01 先分清是哪一种错

AI 出错大致三种。第一种是接口没通：超时、限流、返回一堆看不懂的字。这种重试基本能好。

第二种是格式不对：你要JSON，它给你一段散文；你要三个字段，它给你五个。这种改提示词能好。

第三种是它真干不了：资料里根本没有这个信息，或者这活超出它的能力。这种重试一百次也没用。

还有一种更麻烦的：它答得很像样，但内容是错的。这种根本不报错，只能靠你自己看一眼。

- 网络类的错等一下再试，格式类的错把要求写死再试，能力类的错换思路别硬试
- 同一个错连试三次还一模一样，就是提示词或任务本身有问题，先改这个
- 把每次报错原文存下来。它比「失败了」三个字有用一百倍

## 02 重试要有个数：三次，隔几秒

最省事的写法是「最多试三次，间隔逐次拉长」。第一次等 2 秒，第二次 5 秒，第三次 10 秒。

为什么要拉长？因为多数报错是对方忙。你立刻再撞一次，多半还是撞墙。

三次都失败就停手，别再试了。无限重试只会把账单和日志一起撑爆。

- 只重试值得重试的错。密钥过期、参数拼错这种，重试一万次还是错
- 重试要幂等（同一个请求发多次，结果一样）。发通知、下单、扣钱这类，重试前先查一遍是不是已经成功了
- 连续失败超过设定次数就告警。别让它悄悄一直错下去
- 每次重试前先把这次请求记一笔，失败三次后你能顺着日志查到是哪条输入

**示例代码：三次重试，间隔逐次拉长**

```
import time

def ask(prompt, tries=3):
waits = [2, 5, 10]
for i in range(tries):
try:
return call_model(prompt)
except (TimeoutError, RateLimitError) as e:
print(f"第 {i + 1} 次失败：{e}")
if i < tries - 1:
time.sleep(waits[i])
return None # 交给兜底处理
```

## 03 兜底：出错也得给个交代

兜底就是「主路走不通时走哪条」。最简单的兜底是给个默认值：总结生成不出来，就返回原文前三百字。

再进一步是降级：主模型超时就换一个更快的模型，或者干脆跳过这一步，先把别的做完。

最要紧的是别静默失败。出错要留痕、要有地方能看到，不然用户只看到页面一片空白。

- 每处调用都想一句「它挂了会怎样」，答案不能是「整个流程停住」
- 兜底结果要标出来，让人知道这是兜底，不是正常结果
- 能缓存就缓存。同一份输入算过一次，别每次都重新问一遍

- 日志里至少写清三件事：哪一步失败、报错原文是什么、最后走了哪条兜底路

## 能直接抄的模板

让 AI 帮你把一段代码补上重试和兜底

> 下面这段代码会调用 AI 接口，现在没有任何出错处理。请帮我加上重试和兜底：
> 1. 只对超时和限流重试，最多三次，间隔分别是 2 秒、5 秒、10 秒。
> 2. 参数错误、密钥错误这类不要重试，直接报出来。
> 3. 三次都失败时走兜底：返回一个写死的默认值，并打一条带原始报错的日志。
> 4. 把重试次数和最终结果记下来，方便我事后查。
> 5. 改完把改动的地方逐条说明，别整段重写。
>
> 代码：
> （贴你的代码）

## 常见坑

### × 一报错就重试

报错原因没看就重试，等于闭着眼睛撞墙。先读一遍返回的那行字，多数错它自己写清楚了：是超时、是额度用完、还是参数拼错了。

### × 无限重试

写个循环一直试下去，碰上对方持续故障就成了死循环，账单和日志一起爆。给次数封顶，比如三次，超了就走兜底并告警。

### × 在会重复扣钱的地方重试

发通知、下单、扣款这类操作，重试可能做两遍。重试前先查一次状态，或者给每次请求带一个唯一单号，让对方按单号去重。

### × 出错就静默吞掉

包一层出错处理但什么都不做，程序是不崩了，问题也藏起来了。至少要打日志、记次数，让人能在监控里看见。

## 做完自查

- ☐ 我能说清这次报错属于哪一类，而不是笼统地说「它坏了」
- ☐ 重试次数有上限，间隔逐次拉长，不是原地狂试
- ☐ 每处调用都有兜底，出错时流程还能往下走
- ☐ 兜底结果会被标出来，出错会留下日志，不会悄悄消失

# 18
# 别让账单失控：Token 和成本

AI 是按字收费的，你塞得越多付得越多。这篇教你算清一次调用花多少钱，再用几个小习惯把账单压下来。

### 这篇讲什么

☐ 知道钱花在哪：发过去的和返回来的都算钱

☐ 会用「少塞背景、管住输出、挑对模型」三招降成本

☐ 能给一个功能算出每月大概开销，不至于被账单吓一跳

## 01 先搞懂它怎么算钱

AI 按 token 收费。token 是它的计价单位，可以理解成「一小段字」，一个汉字差不多一到两个。

关键是你发过去的话和它返回的话，两头都要算钱。你贴进去一份五千字的文档，它读一遍就收费。

所以省钱的第一原则不是少问，而是少塞。背景资料能摘成三百字，就别贴三千字。

另外，很多平台是按「每千 token」报价的。看单价时先统一单位，别把千次当成一次来算。

- 输出一般比输入贵。让它少废话，比让它少读更省钱
- 每轮对话都会把前面的历史重新算一遍。聊到二十轮，你等于在反复付前十九轮的钱
- 长文档先自己摘一遍再给它，往往能省掉一大半

## 02 三个立刻见效的省钱动作

第一个：管住输出长度。直接写「不超过 200 字」「只给三条，每条一句话」，别让它写八百字你只看三行。

第二个：及时开新对话。一件事干完就重开，别在一个对话里塞十件不相干的事。

第三个：挑对模型。改文案、整理表格这种，便宜的中小模型完全够；只有真难的活才上最强的。

- 把固定背景写成一段模板，每次粘一段，比反复解释省事也省字
- 要它「只给答案，不要解释过程」，输出能砍掉一半
- 批量任务先拿十条试跑，看花了多少，再决定要不要跑一万条
- 同样的活，提示词从三百字压到八十，钱跟着降，效果往往还更稳

**示例：把提示词收紧，输出自然就短了**

```
任务：把下面的会议记录整理成待办清单。
输出格式：只给三条，每条一行，格式为「谁 - 做什么 - 什么时候」。
字数：总共不超过 100 字。
不要写前言，不要总结，不要解释你的思路。

会议记录：
（贴记录，控制在 500 字以内）
```

## 03 给一个功能算笔月账

算法很简单：单次调用花的钱，乘以每人每天用几次，再乘以人数，再乘以 30 天。

举个例子。假设一次调用输入约 800 个 token、输出约 400 个，一天跑 500 次。一个月就是一万五千次，按公开单价一乘，你就知道这笔钱值不值。

算完通常会发现问题不在用户量，而在单次调用的浪费上。把单次压下来，比砍功能管用。

单价各平台不一样，动手算之前去官网查一眼，别凭印象估。

- 先设个月度上限，撞到就停或降级，别等账单来了才知道
- 把每次调用的 token 数记进日志。哪一步最贵，一眼就看出来
- 月成本超预期时，先查日志里最贵的那一步。浪费通常集中在一两处，改它就行

## 能直接抄的模板

算清一个功能的月成本，直接套这段

> 帮我算一下这个功能的月成本，按下面步骤走：
> 1. 估单次调用：输入大概多少 token，输出大概多少 token。
> 2. 按公开单价分别算出输入和输出的钱，加总得到单次成本。
> 3. 估使用量：预计多少人用、每人每天几次。
> 4. 单次成本 × 每天次数 × 人数 × 30，得到月成本。
> 5. 给三个能把单次成本降下来的具体改法，每条说明大概能省多少比例。
> 我给的数字有缺的地方，你先按保守值假设，并明确标出这是假设。
>
> 功能说明：
> （一句话说清这功能干什么）

## 常见坑

### × 把整份文档贴进去

几十页的文件全塞进去，读一遍就按全文收费，而大部分内容跟问题无关。先自己摘出相关两三段，或者先分段总结、再拿总结去问。

### × 一个对话聊到底

每轮都会重算前面的历史，聊得越久单次越贵。一件事一个对话，办完就重开，必要背景重新粘一段。

### × 所有活都用最强的模型

改错别字、分个类这种活，最强模型和普通模型结果差不多，价格差好几倍。按难度分档：难的用强的，日常用便宜的。

### × 不设任何上限

没预算、没告警，等发现时已经跑了一大笔。给账号设额度上限，给程序加每日次数上限，撞线就降级或停。

## 做完自查

☐ 我知道输入和输出都收费，所以不把无关的长文整篇塞进去

☐ 我会写清输出长度和格式，不让它自由发挥写一大篇

☐ 一件事一个对话，办完就重开，不把历史越拖越长

☐ 上线前我算过单次成本和月成本，也设了额度上限

# 19
# 那么多模型，到底选哪个

贵的未必合适，便宜的也未必不行。这篇给你一张按活挑模型的对照表，外加一个三十秒就能做完的选择流程。

### 这篇讲什么

☐ 知道模型之间真正差在哪三件事上，不被参数和榜单带跑

☐ 能按手里的活直接挑出一档，不用每次纠结半天

☐ 会用自己五道真题实测，用结果说话，而不是听宣传

## 01 模型之间差在哪三件事

第一是会不会想。有的模型擅长多步推理，写代码、算逻辑、找矛盾比较稳；有的只是文笔好。任务一复杂，差距立刻显出来。

第二是看不看得懂图。要读截图、扫描件、图表，就得挑支持图片输入的。

第三是快不快、贵不贵。同一档能力里，有的响应快价格低，有的慢一些但更稳。

- 榜单排名只能当参考。它测的题，跟你手上的活常常不是一回事
- 参数多大不代表更适合你。够用就行，大的通常更慢更贵
- 模型会下线、会改名、会升级。别把选择写死在代码里，留个开关随时能换
- 同一个模型，开了「深度思考」会明显更慢更贵。简单的活别开，开了纯属浪费

## 02 按活挑，别按名气挑

日常文字活，改文案、写邮件、润色、翻译、起标题：用便宜的中小模型，快且够用，量大了也不心疼。

要动脑子的活，写代码、排查报错、做多步分析、核对长文里的矛盾。这种上最强的模型，多花的钱省下的是你的时间。

要读图的活，看截图报错、认发票、读表格照片：先确认它支持图片输入，不支持再强也没用。

还有一类最容易被忽略的活：把一大段乱糟糟的话整理成表格。这种便宜模型就够，别浪费最强的。

- 整本长文档：挑上下文窗口大的（一次能塞进去的字数上限），不然中途会被截断
- 要严格格式（JSON、表格）：挑听话的，能只输出格式、不加废话的那种
- 要等结果给人看：优先挑快的。慢三秒，人可能就走了
- 别用「哪个模型最好」这种问法。AI 的回答会跟着网上的风评走，落不到你的活上

## 03 三十秒定下来：同一道题跑两家

别猜。从你真实的工作里挑一道题，最好是之前踩过坑的那道。

同一个提示词原样发给两三个候选模型。结果并排放着看三点：能不能直接用、有没有硬错、花了多久。

能直接用、没错、够快，就是它。测过一轮，往后半年不用再纠结。

测的时候每次都在新对话里跑。上一题的上下文会干扰这一题，结果就不准了。

- 至少测五道题。一道题看不出差别，容易碰上运气
- 把测过的题和结果存下来。模型一更新，拿同一套题重跑，好坏立刻见分晓
- 记录时顺手写上日期。半年后模型换代了，你能对得上当时测的是哪一版

**示例：对比测试记录表，复制到表格里用**

> 测试题：把这段客户投诉改写成 100 字以内的回复，语气客气但不要道歉过头。
>
> | 模型 | 能不能直接用 | 有没有硬错 | 用了多久 | 备注 |
> |---|---|---|---|---|
> | A | 改两处 | 无 | 8 秒 | |
> | B | | | | |
> | C | | | | |

## 能直接抄的模板

拿不准的时候，用这段让 AI 帮你选

> 我手上有个真实任务，请帮我挑模型，别给排行榜，按下面说：
> 任务：_____
> 一次要处理的内容大概多长：_____
> 需不需要它读图片：需要 / 不需要
> 我能不能等：可以等十几秒 / 要几秒内出结果
> 用量：每天大概 _____ 次
>
> 要求：
> 1. 先说这活最吃哪一项能力（推理、读图、速度、长文），只挑一个最主要的。
> 2. 给「够用就行」和「要最好」两档建议，各说一句为什么。
> 3. 说清选错档会出什么问题，我该怎么发现。

## 常见坑

### × 只看榜单选模型

榜单测的多是考试题，跟你的活不一样。花半小时用自己五道真题测一遍，比看十个榜单准。

### × 所有活都上最贵的

改错别字、分类、格式转换这类活，便宜模型做得一样好，价格差好几倍。按难度分档，能省下大部分钱。

### × 把模型名写死在代码里

模型会下线、会改名、会涨价。把模型名放到配置里，留一个一键切换的开关，换的时候不用改代码。

### × 想靠换模型救烂提示词

要求写得含糊，换十个模型一样含糊。先把任务、格式、例子写清楚，再考虑换模型。

## 做完自查

☐ 我按活挑模型，不是按名气或榜单挑

☐ 要读图、要长文、要快，这三个需求我先确认过支不支持

☐ 我用自己五道真题实测过，不是凭感觉选

☐ 模型名写在配置里，随时能换，不写死在代码里

# 20
# 上下文窗口：它能一次看多少字

它一次能看多少字是有上限的。搞懂上限在哪、怎么估，你就知道为什么长文档聊着聊着就崩，也知道该怎么拆。

### 这篇讲什么

☐ 知道「上下文窗口」到底限制了什么，不再以为它能记住你所有资料

☐ 会粗估一段材料占多少额度，提前判断塞不塞得进去

☐ 知道它聊到一半开始跑偏时，具体该怎么救回来

## 01 它一次只能看这么多字

上下文窗口（就是它一次对话里能同时看到的全部文字，你发的和它回的都在里面）是有上限的。这个上限一般用「词元」来算。

词元（token，AI 把文字切成的小碎片，一个字或半个词算一个）是它的计量单位。你发的话、它回的话、你贴的文件、系统给它的设定，全都从这一个额度里扣。

额度用完了会怎样？最早说的话被挤出去。它不会提醒你，但你再问它「我前面说过啥」，它就开始编。

- 主流模型一般能一次看几万字到几十万字，具体看型号，用之前先试一下
- 它回话也占额度。让它写长文，能用来装材料的地方就更少
- 额度是「一次对话」的，不是永久的。开个新对话就重新算

## 02 怎么估你的材料占多少

最省事的办法是看字数。中文一个字大概算一个到一个半词元，英文一个单词大概一个多词元。

所以一份三万字的报告，粗略按四万到五万词元准备。你手上模型的额度要是只有三万，那就塞不进去。

不用算得特别准。你要的只是一个判断：这堆东西是能一次塞进去，还是必须拆开。

- 一页 A4 中文大概八百到一千字，十页就是一万上下
- 表格、代码、网址特别吃额度，同样字数占的位置更多
- 拿不准就先贴一半试，看它回得顺不顺

## 03 额度快满了，它就开始犯糊涂

聊得久了你会发现它变笨：前面提的要求忘了，刚纠正的错又犯一遍。这不是它偷懒，是那些话被挤出窗口了。

还有个反常识的地方：材料塞得越满，它抓重点越差。中间那部分最容易被忽略，开头和结尾它记得最清楚。

所以别追求一次全塞进去。塞八成、留点余量，回答质量往往更好。

- 重要的要求放在最前面，或者最后再复述一遍
- 长对话聊到关键节点，让它先总结成一段，再开新对话接着聊
- 发现它开始忘事，就别硬聊了，重新交代背景更快

## 能直接抄的模板

贴长材料之前，先让它给你算一下额度

> 我下面要发一段材料，你先别急着回答，按这个顺序来：
> 1. 先告诉我这段材料大概多少字、多少词元，估算就行。
> 2. 告诉我按你现在的上下文窗口，装进去之后还剩多少余量。
> 3. 如果余量不到三成，先建议我怎么拆，等我确认了再动手。

## 常见坑

× **以为它记得你之前发的所有文件**

窗口是有限的，超了就丢最早的内容。要它一直记着的资料，每次对话开头重新贴一遍，或者只贴相关的几页。

× **一次把整本手册贴进去**

塞满之后它抓不住重点，中间的内容基本被忽略。先自己筛出相关的两三章，再贴。

× **只拿字数当标准**

同样字数，表格和代码占的位置更多，中文和英文也不一样。估的时候多留三成余量。

× **它犯糊涂了还在硬聊**

那是窗口满了。让它把结论总结成一段，开个新对话贴进去继续，比反复纠正快得多。

## 做完自查

☐ 我知道窗口是有限的，超了会丢最早的内容

☐ 贴长材料前，我会先估一下字数和余量

☐ 重要的要求我放在开头或结尾，不放中间

☐ 它开始忘事时，我会总结后开新对话，而不是硬纠正

# 21
# 长文档怎么喂给 AI

几十页的文档整份扔进去，只会拿到一份废话摘要。这一篇讲怎么切、怎么问、怎么核对，让它真把文档读进去。

### 这篇讲什么

☐ 会把一份长文档拆成它吃得下的几块，而不是整份硬塞

☐ 知道该问什么样的问题，才能拿到有出处的答案

☐ 会做一套能反复用的文档问答流程

## 01 先别急着整份扔进去

拿到一份五十页的 PDF，多数人的第一反应是整个上传，然后打一句「帮我总结一下」。结果拿到一段谁都能写的空话。

问题不在它笨，在于你要的东西太模糊。整份文档里九成内容跟你这次要干的活无关，它只能平均用力。这就像让人在一本书里找一句话，却不告诉他要找哪句。

顺序应该反过来：先说你要干什么，再决定给它看哪一部分。这一步花两分钟，能省下后面好几次返工。

- 先问自己一句：我要从这份文档里拿到什么？一句话说不出来就先别问
- 能定位到章节的，就只贴那几章
- PDF 里的表格和图，转成文字经常是乱的，贴之前自己扫一眼

## 02 切成块，一块一块问

文档超过一两万字就切开，按章节切最省事，每块控制在几千字。切成五六块一般刚好，每块它都能一口气看完。

每块问同一个问题，最后让它把几块的答案合起来。这样它每次只需要专心看一小段，反而答得准。

切的时候别把一句话从中间切断，也别把标题和正文分开。标题是它认路的路标，一分开它就不知道这段在讲哪块。

- 每块开头加一句「这是某报告的第 3 章，讲成本结构」，它就不会看糊涂
- 每块的答案都要求带上原文里的一句话当证据
- 切完先拿一块试，答得不对就改问题，别一口气全跑完

## 03 让它给出处，你才好核对

文档问答最怕的是它把几段内容缝在一起，说得像那么回事，其实原文没这句。

解法很简单：每个结论都要求它引用原句。引不出来，就说明那段是它自己编的。它不会主动告诉你这段其实不对题。

你核对的时候不用读全文，只看它引的那几句，几分钟就能验完。这一步别省，省下的时间迟早要还。

- 要求它写清第几页或第几节，方便你翻回去
- 它说「文档里没提到」时，先确认你是不是真的没贴那部分
- 涉及数字和日期，一定回原文对一遍

## 能直接抄的模板

文档问答的标准问法，填空就能用

> 下面是《文档名》的第 3 部分，原文在【】之间。
> 【把这一块原文贴在这里】
>
> 请只根据【】里的内容回答，不要用你自己的知识补充。
> 1. 用三句话回答我的问题：这里写你的问题。
> 2. 每个结论后面标出依据的原句，原文没有依据的不要写。
> 3. 如果【】里没有相关信息，直接说「这部分没提到」，不要猜。

## 常见坑

### × 整份 PDF 上传就问总结

它会给你一份平均用力的空话，重点全丢。先想清楚你要什么，只贴相关章节。

× **一次问十个问题**

它会把答案搅在一起，每个都答一半，你还得挨个追问。一次问一个，答完再问下一个。

× **不问出处**

它会自然地把几段内容缝起来，读着很顺但原文没这句。每个结论都要它引原句，你抽查两句。

× **扫描版 PDF 直接贴**

图片型 PDF 提出来是乱码或者空的，它等于什么都没看到。先做文字识别，或者干脆截图让它看图。

## 做完自查

☐ 我先说清要从文档里拿什么，再决定贴哪部分

☐ 超过两万字的文档我按章节切开，一块一块问

☐ 我要求它每个结论都引原文，并且自己抽查几句

☐ 数字、日期、人名我都会回原文核对

第四部分

# RAG 与 Agent

第 22 — 33 讲

# 22
# 向量检索：让 AI 从你的资料里找答案

你有几百份自己的文件，AI 怎么知道该看哪一份？靠把文字变成数字，按意思像不像来挑。这篇讲清流程和坑。

---

**这篇讲什么**

☐ 明白按意思找和按关键词找差在哪，什么场景该用哪种

☐ 知道搭一个查自己资料的问答，最少需要哪四步

☐ 知道它最容易在什么地方出错，以及怎么少出错

## 01 关键词搜不到的东西，它能搜到

你在公司文档里搜「报销」，只会跳出写着「报销」两个字的文件。写着「费用申请流程」的那份，搜不到。

向量检索（把每段文字变成一串数字，意思相近的文字数字也相近）解决的就是这个问题。所以搜「报销」，它能找出「费用申请流程」，一个字都对不上也照样命中。

关键词搜索比的是字面，它比的是意思。但要注意：它找的是最像的几条，不一定是对的，最后还得人看。

- 一段话、一小节做成一条最合适，整篇文档做成一条就太粗了
- 同一件事换个说法它也能找到，比如「请假」和「休假申请」
- 它不懂否定和范围，问「不含税的报价」容易找错

## 02 整个流程就四步

第一步，把资料切成小段，一段几百字。第二步，每段过一遍模型，换成那串数字存起来。这步叫建索引，做一次，以后一直能用。

第三步，你提问时把问题也换成数字，去库里找最接近的几段。第四步，把这几段原文塞给模型，让它照着答。

关键在于第四步：模型答的是你资料里的原文，不是它脑子里记的。所以它才说得出你公司内部的规定。

- 切段时让每段自带一点上下文，比如开头带上所属章节的标题
- 找回来的段数别太多，三五段够用，太多反而冲淡重点
- 让它答完列出用了哪几段，方便你翻原文

## 03 它最容易错在两个地方

第一个：切段切坏了。一句话被切成两半，或者一张表被打散，找回来也是残缺的，它只能猜。

第二个：找回来的是字面像、意思不对的段落。它照样会照着这段硬答，而且答得挺顺。

所以这套东西适合资料里本来就有明确答案的问题。需要算、需要比较、需要下判断的，它不擅长。

- 资料要更新：文件改了索引得重建，不然它答的还是旧版
- 先拿十个你已知答案的问题试一遍，答对了再给别人用
- 答不上来时让它说「资料里没有」，比编一个强

## 能直接抄的模板

让模型只根据找回来的资料回答

> 下面是从我的资料库里检索到的几段内容，按相关度排序。
> 【片段1】粘贴原文
> 【片段2】粘贴原文
> 【片段3】粘贴原文
>
> 请只根据上面这几段回答我的问题：这里写你的问题。
> 1. 答案里每句话都标出来自哪个片段，格式写成（片段2）。
> 2. 如果这几段加起来还不足以回答，直接说「资料不足」，并告诉我还缺什么信息。
> 3. 不要用你自己的知识补充，也不要推测。

## 常见坑

× **以为建好索引就一劳永逸**

资料一改，索引就过期了。定个更新节奏，比如每周重建一次，或者文件一改就重建。

× **把整篇文档当成一条**

一条几百字才合适。整篇做成一条，找回来是一大坨，模型抓不住重点。

× **拿它问要推理的问题**

它只会找意思像的段落，不会算也不会比较。要分析就让它先找出原文，再单独问。

× **不问出处就信**

它会把找回来的段落缝得很顺，看着像原文其实不是。要求它标出片段编号，自己抽查两三条。

## 做完自查

☐ 我知道它按意思找，不按字面找，所以换个说法也能命中

☐ 我的资料切成了几百字一段，不是整篇一条

☐ 我要求它标出答案来自哪一段，并且自己抽查

☐ 文件更新后我会重建索引，不拿旧结果当准

# 23
# RAG 是什么，为什么公司都在做

RAG 就是让 AI 先查你的资料再回答。这篇讲清它到底解决了什么问题，以及为什么公司都在做这件事。

### 这篇讲什么

☐ 能用一句大白话跟同事讲清 RAG 到底在干什么

☐ 知道哪类问题适合上 RAG，哪类纯属白花钱

☐ 能看懂供应商方案里哪些说法是在唬人

## 01 说白了就是「先翻资料，再动笔」

RAG 的全称是「检索增强生成」。说白了，就是让 AI 先去翻你的资料，再照着资料回答。

拆开看就两步。检索，就是先在你自己的资料库里搜一遍。增强，就是把这些搜到的内容一起交给模型。

你问一个问题，它先搜出几段最相关的内容，再连问题一起交给模型，让它照着这几段回答。跟人写报告一个道理，只是它翻得比你快。

- 它不改模型本身，只是提问时临时给它塞资料
- 模型可以随时换，资料库才是你的东西
- 资料更新了答案就跟着变，不用重新训练

**它的一次回答长这样（示意）**

```
用户问题：试用期能不能延长？

第一步 检索：在员工手册里搜到 3 段最相关的内容
第二步 拼接：把问题 + 这 3 段一起交给模型
第三步 生成：只根据这 3 段回答，并标出来源

最终回答：可以延长一次，具体天数见公司规定。
（来源：员工手册 • 第 4 章 试用期管理）
```

## 02 为什么公司非得做这个

因为大模型不知道你公司的事。产品报价、内部流程、客户合同，它一概没学过。

你当然可以把资料整段贴进去问。但一份手册几十万字，塞不进去。每次重贴，又慢又贵。

RAG 正好卡在这个缝里：只把最相关的那几段送进去。又快，又省钱。

- 答错能查：它引用了哪段资料，你一眼能翻回去
- 权限能管：谁能看哪份文件，在检索那一步就能卡住
- 不用重训模型，改资料就能改答案

## 03 什么时候别上 RAG

如果问题只需要常识，或者你手上根本没像样的资料，上 RAG 就是白花钱。

还有一种常见误会：拿它当计算器、当审批流程。这些活它干不了，检索再准也没用。

判断标准很土：你手上有 20 份以上、能写成文字、经常被人翻的资料，才值得做。

## 能直接抄的模板

判断你的场景该不该做，把这段发给 AI 让它帮你算账

```
我要判断一个场景值不值得做 RAG（让 AI
先查我的资料再回答）。请按下面的信息帮我判断，不要泛泛而谈：

1. 我手上能用的资料：___ 份，大概 ___ 万字，更新频率是 ___
2. 每天大概有多少人要问这类问题：___ 人，每人 ___ 次
3. 现在他们是怎么找答案的：___
4. 答错一次的代价大概是什么：___

请回答三件事：
- 这个场景适合做 RAG 吗？给明确结论和理由。
- 如果不适合，更省钱的办法是什么？
- 如果适合，第一步先做什么、先别做什么。
```

## 常见坑

### × 以为做了 RAG 就不会瞎编

检索没搜到相关内容时，它照样会凭记忆编。正确做法是在提示词里写死：资料里没有就说「资料里没写」，不要自己补。

### × 整个文件夹一股脑塞进去

里面混着过期的旧版、会议纪要、废稿，搜出来的东西越搜越乱。先清资料，再做检索，这一步省不掉。

### × 只看答案，不看它引用了啥

没有出处就没法追责，也没法发现问题。要求它每条结论后面标出是哪份文件、哪一段。

### × 指望它替人做判断

它给你的是「资料里怎么说」，不是「这事该怎么办」。合同能不能签、报价批不批，还得人来定。

## 做完自查

☐ 我能用一句大白话跟同事讲清 RAG 是干什么的

☐ 我知道自己手上有多少份资料、多久更新一次

☐ 我问过自己：不做 RAG 有没有更便宜的办法

☐ 我要求了它标出处，并且会随手抽查几条

# 24
# 从零搭一个能用的知识库

不用写代码也能搭起来。这篇按顺序讲清五步：清资料、切块、入库、接问答、自己考它一遍。

### 这篇讲什么

☐ 能在一台普通电脑上跑通一个会回答问题的小知识库

☐ 知道每一步该准备什么、卡住了去哪儿找原因

☐ 能用 10 个问题测出它到底靠不靠谱

## 01 第一步：先把资料收拾干净

别急着上工具。先建一个文件夹，只放真正要用的资料：产品文档、常见问题、操作手册。

把重复的、过期的、草稿全挑出去。同一份文件只留最新一版，文件名写日期。

这一步花的时间，后面全会还给你。资料脏，后面怎么调都救不回来。

- 统一格式：全部转成纯文本或 PDF，别混扫描件
- 一份文件只讲一件事，标题写清楚
- 先攒 10 到 50 份，够用就行，别贪多

## 02 第二步和第三步：切块、入库

长文档要切成小段，一段几百字。切太长，搜出来的内容一半是废话。切太短，一句话被拆散，意思就断了。

切的时候按标题、按段落切，别硬按字数砍。每段前面带上文件名和章节名。

然后每段变成一串数字存起来。这叫向量（就是把一段文字变成一串数字，意思相近的文字数字也相近）。这一步工具会自动做。

- 段落之间留点重叠，上一段结尾的两句话放进下一段

- 表格单独处理，别让它被切碎
- 入库前抽查 5 段，看切出来的东西读得通不通

## 03 第四步和第五步：接上问答，然后自己考它

把检索和模型接起来。先搜 3 到 5 段，再让模型只根据这几段回答，并要求标出处。

接完别急着上线。先自己出 10 个问题。5 个资料里明确写了，3 个资料里没写，2 个要跨两份文件。

答对几个，心里就有数了。资料里没写的那 3 个，它要是硬编，就回去改提示词。

- 答不出来先看检索：是没搜到，还是搜到了没答好
- 没搜到就调切块和关键词，搜到了没答好就调提示词
- 每次改完重跑这 10 个问题，别凭感觉说「好像好点了」

**10 个测试问题这样记，改完就重跑**

```
1. 退货要几天到账？（资料里有）
2. 发票开错了怎么改？（资料里有）
3. 会员能开发票吗？（资料里有）
4. 周末客服上班吗？（资料里有）
5. 换货要运费吗？（资料里有）
6. 能开发票给国外公司吗？（资料里没写）
7. 能上门取件吗？（资料里没写）
8. 学生有折扣吗？（资料里没写）
9. 买了会员又退货，会员费退不退？（要跨两份文件）
10. 大件商品坏了怎么处理？（要跨两份文件）

记分：答对 1 分，编造内容扣 2 分。8 分以上再用。
```

## 能直接抄的模板

让它只根据资料回答的固定开头，每次提问前粘一遍

> 你只能根据下面提供的资料回答，不许用资料以外的知识。
>
> 规则：
> 1. 资料里没有的内容，直接回答「资料里没写」，不要猜。
> 2. 每条结论后面标出来源，格式是「（文件名 • 章节名）」。
> 3. 几段资料互相矛盾时，把矛盾的地方指出来，不要自己选一个。
>
> 资料：
> {把检索到的几段内容贴在这里}
>
> 问题：{你的问题}

## 常见坑

× **一上来就想做全公司的知识库**

范围越大，资料越脏，越做不完。先挑一个部门、50 份文件做通，跑顺了再扩。

× **抄别人的切块数字**

别人的 800 字不一定适合你。先按 500 字左右试，抽 5 段读一读，读不通就调。

× **搜不到就怪模型**

八成是检索没捞到东西，模型再强也没用。先把搜出来的段落打印出来看一眼。

× **上线以后就不管了**

资料会过期，答案会跑偏。每月抽 10 个真实提问重跑一遍，答错就补资料。

## 做完自查

☐ 我的资料清过一遍，过期的和草稿都挑出去了

☐ 我抽了 5 段切好的内容读过，意思没被切断

☐ 我出了 10 个测试问题，其中 3 个资料里根本没写

☐ 它答「资料里没写」时，我知道该补资料还是改提示词

# 25
# 资料切多大块才合适

切太大，搜出来的内容一半是废话；切太小，一句话被拆散。这篇给你一个起手值和三个调整信号。

### 这篇讲什么

☐ 能给自己的资料定一个具体数字，不再凭感觉

☐ 知道搜不准时该调大小，还是该改别的

☐ 能给不同类型的资料分别设不同的切法

## 01 先给个能用的起手值

不用纠结理论。中文资料，先按 300 到 500 字一块试。这个范围能覆盖大多数手册、政策和说明文档。

为什么是这个数。400 字左右，大概刚好讲完一个小问题，又不至于混进三四个话题。

块与块之间留 50 到 100 字重叠。上一段结尾的话，下一段开头再出现一次，防止意思被切断。

- 问答列表：一块就放一问一答，200 字以内
- 操作手册、条款：按小标题切，300 到 600 字
- 会议纪要：按一次发言或一个话题切，别按字数砍

## 02 三个信号，告诉你切错了

信号一：搜出来的段落里，一半内容跟问题无关。说明块太大，把不相关的东西也捞上来了，往下调。

信号二：搜出来的只有半句话，前后都不挨着。说明块太小，往上调，同时加大重叠。

信号三：明明资料里有，就是搜不到。这多半不是大小问题，是关键词没对上，先别急着改数字。

- 一次只改一个参数，改完拿同一批问题重跑
- 在 200 字到 800 字之间慢慢试，别一步跳到几千字
- 把每次的数字和得分记下来，别凭印象

## 03 给每一块配张身份证

光有正文不够。每块前面加上文件名、章节名、更新日期。搜的时候更容易对上，答的时候也能标出处。

举个反例。一块内容写着「有效期 30 天」，不加文件名，谁也不知道是退货的还是优惠券的。

加上这几行，等于给每块配了张身份证。搜得准一些，出了问题也追得回去。

- 文件名写全，别用「文档1」「最终版」这种
- 日期写清楚，同一份文件有多个版本时尤其重要
- 表格转成「项目 | 数值」再切，别让一行被拆开

**切出来的一块，长这样**

> 【文件】员工报销手册_2024版.pdf
> 【章节】第三章 差旅报销
> 【更新】2024-06
> 【正文】
> 出差住宿按城市分三档报销，A 档一线城市，B 档二线城市，C 档其他城市。
> 每档上限见附表，超出部分自理。报销时需附酒店发票原件，发票抬头必须
> 与公司名称一致，否则财务会退回重开。

## 能直接抄的模板

让 AI 帮你把一份长文档切块，直接给指令

> 下面是一份文档，请帮我切成适合做检索的小块。要求：
>
> 1. 按小标题和自然段切，不要按字数硬砍，一块控制在 300 到 500 字。
> 2. 每块开头加三行标签：【文件】【章节】【更新】。
> 3. 相邻两块之间保留 50 字左右重叠，也就是上一块的结尾在下一块开头重复一次。
> 4. 表格单独成块，转成「项目 | 数值」的形式。
> 5. 切完列出每一块的字数和它讲的是什么事，我要检查有没有切歪。
>
> 文档内容：
> {贴在这里}

## 常见坑

× **所有资料用同一个块大小**

问答列表和长条款的节奏完全不同。至少分两类：问答类一块一问，文档类按章节切。

× **为了「完整」把块调到几千字**

块越大，捞上来的无关信息越多，模型容易被带偏，还更费钱。宁可多切几块。

× **切完不读一遍**

机器切出来的东西经常断在半句话上。抽 5 到 10 块从头读一遍，读着别扭就调。

× **搜不准就死磕块大小**

有些问题搜不到跟大小无关，是提问用词和资料对不上。试试让模型先把问题改写成几个关键词再搜。

## 做完自查

☐ 我给自己的资料定了具体数字，不是「差不多」

☐ 问答类和文档类我用了不同的切法

☐ 我抽了 5 块从头读过，没有断在半句话上

☐ 每次调完我都拿同一批问题重跑，并且记了分数

# 26
# 选哪个模型做向量化

向量化模型不是越贵越好，也不是榜单第一就适合你。挑错了，后面检索出来的全是垃圾。这篇给你一套半小时能自己试出来的选法。

### 这篇讲什么

☐ 知道自己的场景该看哪几项，不被榜单排名带偏

☐ 能用十来条真实问题，半小时比出哪个模型更合适

☐ 知道换模型的代价，也知道什么时候根本不用换

## 01 先弄清这一步在干什么

向量化（把一段文字变成一串数字，意思相近的文字，数字也相近）是搜索和问答的第一步。文字变成数字之后，电脑才能算出「这两句话像不像」。

这一步选错了，后面全废。模型把「报销流程」和「费用怎么报」算成两串差很远的数字，你再怎么调后面的步骤也救不回来。

所以挑模型的第一个原则：别问哪个最强，先问你库里的话，它认不认得。模型好不好，最终看它在你这批内容上的表现，不是别人的评价。

- 同一个模型算出来的向量才能互相比，混着用会全乱
- 换了模型，之前存下的所有向量都得重算一遍
- 维度（一串数字的长度）不是越高越好，够用就行

## 02 按你的内容挑，别按名气挑

库里如果是中文合同、中文客服记录，就优先挑中文稳的模型。拿英文榜单第一名去跑中文，经常还不如一个专门做中文的。

库里有中英混着的内容，比如技术文档、跨境订单，就挑多语言的。这类模型把中文英文放在同一套数字里，用中文搜英文内容也能搜到。

内容里代码、表格、编号多，就先拿这类内容单独测一遍。通用模型对「SKU-1024」这种串不敏感，可能搜不出原文。

- 短句、标题、商品名这类内容，轻量模型就够，快而且便宜
- 长段落、法律条款、说明书，用能力强的模型，差别很明显
- 别一上来就上最大的，先用小的跑通，再决定要不要换

## 03 半小时做一次自己的小测

准备 10 到 20 条你真实遇到过的问题，每条标好「正确答案是哪一段」。这就是你的考题。

同一批内容用两三个候选模型各跑一遍，看正确答案排在第几位。排第一算对，排到第五名之后基本等于没搜到。

看比例，别看个例。20 条里对 17 条和只对 11 条，差别一眼就看得出来，不用算什么复杂指标。

- 考题一定要来自真实用户，不要自己编教科书式的问法
- 答案排在前三才算搜到，别用「前十里有」安慰自己
- 测完把结果记下来，以后升级模型拿同一批题再跑一次

**示例代码（把模型名换成你实际的）**

```
# 用同一批问题，看两个模型谁把答案排得更前
from openai import OpenAI
client = OpenAI()

def embed(texts, model):
r = client.embeddings.create(input=texts, model=model)
return [d.embedding for d in r.data]

questions = ["报销几天到账", "发票丢了怎么办", "试用期能请假吗"]
answers = ["报销一般三个工作日到账", "发票丢失需提交情况说明",
"试用期请假需提前一天申请"]

for m in ["候选模型A", "候选模型B"]:
qv, av = embed(questions, m), embed(answers, m)
for i, q in enumerate(qv):
scores = [sum(a * b for a, b in zip(q, v)) for v in av]
order = sorted(range(len(scores)), key=lambda j: -scores[j])
print(m, questions[i], "-> 排第", order.index(i) + 1, "位")
```

## 能直接抄的模板

拿这段问 AI，把测试方案和及格线定下来

```
我要给一批中文内容选向量化模型（把文字变成数字，方便算相似度）。
我的内容是：（写清楚，比如「300 条客服问答，中文，每段 100 字左右」）
请帮我：
1. 列出 3 个适合的候选方向，说明各自适合什么内容，不要只推最贵的。
2. 给我 15 条测试问题，要像真实用户会问的，不要教科书式的。
3. 告诉我怎么判断哪个更好，给出具体标准和一条及格线。
4. 说明如果我以后想换模型，需要重做哪些事。
每一条都要落到具体数字上，不要给笼统建议。
```

## 常见坑

### × 拿英文榜单当标准

榜单测的多是英文任务，中文长句、行业词、缩写它根本测不出来。正确做法是用你自己库里的 20 条问题实测，中文内容只看中文结果。

× **中途换模型不重算**

新模型算出的向量和旧的不是同一套数字，混着比相似度结果全是乱的。换模型必须把库里所有内容重新跑一遍，一条都不能漏。

× **段落切得太碎**

一句话切一段，向量里没有上下文，搜出来的东西看着像却没用。按意思完整的小段切，一段 200 到 500 字，前后留一点重叠。

× **只测三条就下结论**

三条碰巧全对，不代表模型行。至少 20 条，覆盖你库里最常见的几种问法，再看对的比例够不够。

## 做完自查

☐ 我用的是自己库里的真实问题做测试，不是别人给的示例

☐ 我知道换模型要重算全部向量，并且预留了这段时间

☐ 我按内容类型（中文、多语言、代码）挑模型，没跟着榜单走

☐ 我先用轻量模型跑通，再决定要不要换更强的

# 27
# 重排序：把最相关的挑出来

向量搜索捞回来的一堆结果，顺序经常是乱的。加一步重排序，把真正能回答问题的几条顶到最前面，AI 才不会答偏。

### 这篇讲什么

☐ 知道什么情况必须加重排序，什么情况加了白花钱

☐ 能把「先捞几条、最后留几条」这两个数定下来

☐ 会用一段提示词做重排，不写代码也能上手

## 01 为什么搜出来的顺序不对

向量搜索（把文字变成数字，按数字的远近找相似内容）是粗筛。它快，但它只看大概像不像，不看这段到底有没有用。

粗筛最常见的毛病是：真正有用的那条排在第 8 位，前面 7 条全是「话题沾边但答非所问」。你直接把前三条喂给 AI，它自然答偏。

重排序就是补这一步：先把候选放宽到几十条，再用一个更细的模型逐条判断「这条到底能不能回答这个问题」，然后重新排一遍。它不改候选，只改顺序，所以快、便宜、也容易试。

- 粗筛负责别漏，重排负责别错，两步分工不一样
- 候选给太少，重排没东西可挑；给太多，又慢又贵
- 重排只处理那几十条候选，成本是可控的

## 02 先捞三十条，再留三条

一开始就把两个数定死：先召回多少条，最后留几条。一般来说，召回 30 到 50 条，重排后留 3 到 5 条交给 AI 读。低于这个数，很多真正相关的段落根本没进候选。

为什么最后只留 3 到 5 条？塞太多，AI 抓不住重点，还容易被不相关的段落带偏，答得越长越容易跑题。

如果问题需要跨好几段拼答案，比如「对比 A 和 B 两款」，就留 6 到 8 条，并且尽量来自不同文档。这类问法最怕候选全来自同一篇。

- 召回数至少是最终条数的 5 倍，不然重排等于没做
- 同一篇文档里相邻的段落，先合并成一条再排
- 给 AI 留一条「都不相关就直说」的出口，别逼它硬答

## 03 不写代码也能做重排

候选只有几十条的时候，你完全可以让 AI 直接读一遍再排序。这比调模型接口简单得多，普通场景效果够用。

关键是把判断标准说清楚：不是问「哪条和问题最像」，而是问「哪条能直接回答问题」。这两件事经常不一样。问「最像」，你捞回来的多半是同一段话的改写；问「能回答」，答案才浮上来。

让它只输出编号，别让它复述原文。原文你自己库里有，让它复述既费钱又容易改字。

- 一次别塞超过 50 条候选，多了它会漏看
- 结果里要它给一句理由，你才知道它凭什么排这个顺序
- 取原文时按编号去库里取，不要让 AI 把内容重写一遍

**示例提示词**

```
下面是一个问题和 30 条候选资料。请只做一件事：挑出最能回答这个问题的 4 条。

判断标准：
1. 能直接回答问题的排前面，只是话题沾边的排后面。
2. 内容重复的只留一条。
3. 一条都答不了，就输出「无相关内容」。

输出格式：只输出编号加一句理由，不要复述原文。

问题：{用户的问题}
候选资料：
1. {内容}
2. {内容}
...
```

## 能直接抄的模板

拿这段问 AI，把召回数和最终条数定成具体数字

```
我在做一个问答工具：用户提问，我先搜出候选资料，再让 AI 回答。
我的资料是：（写清楚，比如「200 篇产品文档，每段 300 字左右」）
用户常问的是：（举例，比如「这个功能怎么开」「和另一个功能有什么区别」）
请帮我定：
1. 每次先搜出多少条候选合适，为什么。
2. 最后交给 AI 几条合适，为什么。
3. 哪种问法需要多给几条，哪种一两条就够。
4. 给我 3 个办法，判断重排到底有没有起作用。
每一条都要落到具体数字上，不要给笼统建议。
```

## 常见坑

### × 候选只捞三条就重排

重排是在候选里挑，候选里本来就没有正确答案，排得再准也白搭。先把召回放宽到 30 条以上，再让重排去挑。

### × 让重排模型复述原文

复述又慢又容易改字，关键数字经常被抄错。只让它输出编号和一句理由，原文从你自己的库里按编号取。

× **把所有候选全塞给 AI 读**

几十条一起塞，它会漏看、抓错重点，费用还高。先用重排筛到 3 到 5 条，再让它读。

× **加了重排却没对比过**

加了不等于变好。留 20 条测试问题，记录加重排前后正确答案排第几。没提升就去查是不是候选太少、段落切得太碎。对比时固定同一批题，别换题。

## 做完自查

☐ 我先捞 30 条以上候选，再重排到 3 到 5 条

☐ 我让重排只输出编号和理由，不复述原文

☐ 我留了「无相关内容」的出口，不逼 AI 硬答

☐ 我用同一批测试问题对比过加重排前后的差别

# 28
# Agent：让 AI 自己动手干活

从「你问一句它答一句」，变成「你说目标它自己跑」。这篇讲清它怎么干活、你怎么交代，以及怎么不让它把事情搞砸。

### 这篇讲什么

☐ 知道一件事该交给 Agent 自己跑，还是你拆成几步更稳

☐ 能把目标、工具、停止条件这三样交代清楚

☐ 知道哪些操作必须留一道人工确认，不能放手

## 01 它跟你平时用 AI 有啥不一样

平时用 AI 是你问一句它答一句，中间怎么走全靠你。Agent（让 AI 自己决定下一步做什么、自己调用工具把活干完）是把中间那些步交给它。

它会自己看情况：先查资料，发现不对换个搜法；写完代码自己跑一遍，报错了自己改。你只在开头说目标，结尾看结果。

代价是它自己拿主意。你交代得含糊，它就按自己的理解一路跑下去，等发现跑偏，可能已经过了几十步。所以交代的功夫，比挑模型重要。

- 它的能力上限不只看模型多聪明，更看你能给它什么工具
- 工具越多它越容易乱用，先给两三个够用的
- 每多给一个工具，就多一种它想歪的方式
- 能连跑多少步要设上限，不然它会在原地绕圈

## 02 把话说成三件事：目标、工具、停在哪

目标要写成能验收的样子。「帮我整理一下客户反馈」不行，「把 500 条反馈按问题类型分成 5 类，每类写一句总结」才行。

工具要说明白它能碰什么、不能碰什么。能读文件不能删文件，能查数据库只能查不能改。没写清楚的权限，它默认自己是有的。

停在哪最容易被忘。告诉它「最多试 5 次」「找不到就停下来告诉我」，比让它自己判断什么时候算完靠谱得多。

- 目标里带数量和格式，它才知道什么时候算做完
- 写明「做不到就停下并说明原因」，这句能省很多返工
- 让它每步留一行记录，出问题你能看出是哪步歪的

## 03 哪些活能放手，哪些必须留道闸

能放手的：查资料、整理格式、写草稿、跑测试、在本地文件里改代码。这些做错了你一眼看得出来，改回来也便宜。这类活它做得比人快，也比人耐心。

必须留闸的：发邮件、下单付款、删文件、改线上数据、对外发布。这些一旦错了，收不回来。

做法很简单：这类动作让它先把要做的事写出来给你看，你点确认它再执行。多花你十秒，省掉一次事故。

- 先在小范围试，比如先跑 12 个文件，跑通了再放到 500 个上
- 同一个错法试 3 次还不行，就该停下来问人，不是继续试

**示例任务说明（可以直接套）**

```
目标：把 D:\报告 里的 12 个 csv 合并成一张表，按「日期」排序，输出到
合并.xlsx。

你可以用：读文件、写文件、运行 python。
你不能：删除任何原文件，不能联网。

做法：
1. 先列出 12 个文件的表头，确认列名一致；不一致就停下来告诉我。
2. 合并后报告总行数，以及有没有空值。
3. 每做完一步，写一行日志说明你做了什么。

停止条件：同一个错误试了 3 次还不行，就停下来说明卡在哪，不要自己换方案。
```

## 能直接抄的模板

不确定该不该交给 Agent 时，先拿这段问 AI

> 我想让 AI 自己动手完成一件事：（写清楚你想干什么）
> 请帮我判断：
> 1. 这件事适合让 AI 一步步自己跑，还是我自己拆成几步更稳？说理由。
> 2. 如果交给它，需要给它哪些工具？哪些工具绝对不能给？
> 3. 帮我写一段任务说明，包含目标、可用工具、禁止动作、停止条件。
> 4. 列出这件事最容易出错的 3 个地方，以及我该怎么提前拦住。
> 要求：目标写成能验收的样子，带数量和输出格式。

## 常见坑

### × 目标写得太虚

「帮我优化一下」这种话，它会自己定义什么叫优化，然后交给你一个你不要的东西。改成能验收的说法：数量、格式、判断标准都给出来。

### × 一上来就给它一堆工具

工具越多，它选错的概率越高，出了问题也难查。先给两三个必须的，把一个跑稳了再加第二个。

### × 没设停止条件

它会反复试同一个错法，跑几十步把额度耗光。写明「同一错误试 3 次就停」，并要求它停下时说清卡在哪，别自己换方案接着试。

### × 让它直接动线上数据或对外发东西

删了、发了都收不回来。这类动作一律改成先写出要做什么，等人确认再执行。

## 做完自查

- ☐ 我的目标里有数量、格式和验收标准，不是一句「优化一下」
- ☐ 我只给了必须的工具，没给能删东西、能对外发消息的权限
- ☐ 我写了停止条件：试几次不行就停下来告诉我
- ☐ 涉及付款、删除、对外发布，都留了人工确认这一步

# 29
# 让 AI 会用工具

光靠嘴说，它只能猜。给它接上搜索、计算、跑代码这几只手，答案才会从「像对的」变成「真对的」。

### 这篇讲什么

☐ 知道哪几类活必须让它动手，不能让它凭印象说

☐ 能一句话把工具递到它手上，并说清什么情况下必须用

☐ 会检查它到底用没用工具，不被一句「我查过了」糊过去

## 01 它出厂时只有一张嘴，没有手

大模型（就是平时跟你聊天的那个 AI）出厂只会干一件事：接着你的话往下写。它不能上网，不能打开你的文件，也不能真的按计算器。

所以你说「帮我查下明天上海的天气」，它不会去查。它会照着「天气回答长什么样」编一段，格式全对，数字全错。

这不是偷懒，是它压根没有手。你得先把工具递过去，它才知道自己能动手。

- 能上网搜：查新闻、查价格、查一份文档里到底有没有这句话
- 能跑代码：算数、处理表格、批量改文件名
- 能读文件：你上传的 PDF、Excel、截图，它才看得见

## 02 把工具递过去，再说清什么时候必须用

光打开开关不够。你得在话里写清楚：遇到什么情况必须去查，不许凭印象答。

比如「你回答前先搜一下最新资料」这句太软，它可能搜一下就走个形式。改成「凡是涉及价格、日期、型号，先搜，搜不到就说搜不到」，它才会真去搜。

工具叫什么、怎么调用，你不用背。你只要说人话：让它查、让它算、让它跑一段代码。

- 给它一条「不确定就查」的硬规矩，比说「尽量准确」有用得多
- 让它把查到的来源贴出来，你一眼就能看出它是不是真查了
- 算数这类活直接说「写段代码算，别心算」

**示例提示词**

> 你现在可以联网搜索。请遵守三条：
> 1. 凡是价格、日期、型号、政策条文，必须先搜，不许凭记忆回答。
> 2. 每条结论后面标出来源链接。
> 3. 搜不到就写「没搜到可靠来源」，不要猜一个填上。

## 03 怎么看出它真用了，还是在装

最省事的办法是让它写过程：搜了哪些词、开了哪个页面、中间结果是多少。

第二个办法是给它一道不查就一定答错的题。比如问某个小软件昨天更新的版本号，它要是张口就来，那就是没查。

第三是看结果对不对得上。数字对不上、链接点不开、日期是去年的，基本就是编的。

- 让它先列「我准备怎么查」，你点头再让它动手
- 重要结论要两条独立来源，别信单条
- 它说「我查过了」不算证据，来源链接才算

## 能直接抄的模板

复制这段，让它每次动手前先说清计划

> 这个任务你可以用工具（搜索、跑代码、读我上传的文件）。
> 开始前先用三行说清：
> 1. 你打算用哪个工具、查什么、怎么查。
> 2. 哪几步不需要工具，你可以直接写。
> 3. 哪一步你最可能搞错。
> 我回「可以」之后你再动手。

## 常见坑

### × 以为它自动联网

多数对话窗口默认不联网，开关藏在设置里。它不联网时照样答得像模像样，你根本看不出来。用之前先确认开关开了，或者直接问它「你现在能上网吗」。

### × 把「查一下」当命令

太模糊，它可能只是把记忆里的东西重说一遍。要说清查什么、查到什么程度算够、搜不到怎么办。

### × 让它算钱算日期

它是猜字的，两个小数一乘就容易错。涉及金额、工期、天数，让它写代码算，并把中间结果打给你看。

### × 工具给了没给权限

接了搜索却没配密钥、传了文件却没开读文件的功能，它只能干看着。跑不通先查工具连没连上，别急着改提示词。

## 做完自查

- ☐ 我知道它默认没手，涉及实时信息会先确认联网开关开了
- ☐ 我会在话里写清「什么情况必须查」，不只笼统说一句准确点
- ☐ 我会让它贴来源链接，而不只听它说「我查过了」
- ☐ 算钱、算日期、算工期，我会让它写代码跑，不让它心算

# 30
# 复杂任务怎么拆成几步

一件大事直接丢给它，出来的东西通常没法用。先让它拆成小步，一步一验收，成品质量立刻不一样。

### 这篇讲什么

☐ 能把「做个方案」这种大话，拆成它真能干的五六步

☐ 知道每步该给它多少信息，不会一次塞爆

☐ 会做中间验收，早早发现跑偏，不用整篇重写

## 01 为什么大事不能一次丢过去

你让它「写一份完整的季度复盘」，它只能猜你想要什么。猜错了，你拿到的是一篇很工整但没用的东西。

还有个麻烦：一次要它干太多，它干到后面会松。前面写得细，后面开始糊，最后一段经常是套话。

拆开就不一样了。每步都小到它能认真做完，你也能一眼看出对不对，错了只改一小块。

- 一步只干一件事：要么定框架，要么找材料，要么写正文
- 每步的产出你能看懂，看不懂就是拆得还不够细
- 一步的活控制在半页纸以内，超了就再拆

## 02 先让它拆，别自己坐那儿想

很多人自己憋步骤，憋半天。这活其实可以交给它，它拆得常常比你想得细。

你就说：这件活分几步，每步产出什么、需要我先给什么。它列完你挑一挑、改改顺序，比从零想快得多。举个例子：它排的第三步是「找材料」，那你就先扔两份旧复盘给它，别等它自己编。

有个细节：让它把「需要你拍板的地方」单独标出来。这样你就知道哪几步得盯着。

- 让它标出哪步最容易返工、为什么
- 步骤别超过 6 步，超了就合并，不然你自己都记不住
- 拆完先别跑，你改一遍顺序再开始

**示例提示词**

> 我要做的是：<把任务写清楚>。
> 先别动手写。请你：
> 1. 把它拆成不超过 6 步，每步一句话说清产出是什么。
> 2. 标出哪几步需要我先给材料或做决定。
> 3. 标出哪一步最容易做偏，为什么。
> 等我确认顺序后，再开始第一步。

## 03 一步一验收，别等最后才看

每步做完，你花两分钟扫一眼：方向对不对、有没有漏掉关键信息。不对就当场改，别攒着。改的时候说清改哪一句，别只丢一句「再优化一下」。

验收说三句话就够：这步哪里好、哪里不对、下一步改什么。不用写长评语。比如「框架可以，但第三段要加数据，下一步补两张表」，这就够了。

还有个小技巧：让它每步结束时用三行写下「这步定了什么」。最后拼起来就是整件事的骨架，回头看特别省事，跟领导汇报时照着念就行。

- 第一步的框架没定下来，绝不往下走
- 每步产出单独存一份，改坏了能退回去
- 同一步跑偏两次以上，说明这步还拆得太大

## 能直接抄的模板

卡住的时候，用这段让它把一步再拆细

> 这一步我总觉得太大，做出来不对。请你把它再拆成 3 个小动作，每个动作写清：
> 1. 我给它什么输入
> 2. 它输出什么
> 3. 怎么判断它做对了
> 拆完先给我看，我说行再往下。

## 常见坑

### × 一口气把要求全写完

提示词写了两千字，它反而抓不住重点。先给目标和几条硬约束，细节留到对应那一步再说。信息分批给，它记得更牢。

### × 拆完不检查就一路跑

步骤顺序错了，后面全白干。框架那步一定要你点头，别让它自己往下冲。

### × 改一段却让它重出全文

前面好的部分会被顺手改坏。要说清「只改这一节，其他原样保留，不要动」。

### × 拆得太碎

拆成二十步，光来回沟通就累死，还容易前后矛盾。三到六步最合适，一步对应一个能验收的产出。

## 做完自查

- ☐ 我把大任务拆成了不超过 6 步，每步都有能看懂的小产出
- ☐ 第一步的框架我确认过了，才让它往下走
- ☐ 每步结束我花两分钟验收，不等最后一起看
- ☐ 让它改局部时，我会补一句「其他部分原样保留」

# 31
# 让 AI 记住以前的事

它没有长期记忆，一次对话也装不下太多东西。学会做背景卡和阶段小结，隔天接着干也不会断片。

### 这篇讲什么

☐ 知道它为什么会忘，忘在哪一步

☐ 能给一个项目做张背景卡，新开对话十秒接上

☐ 会把聊出来的结论落成小结，下次直接喂回去

## 01 它不是忘了，是压根没存

每次开新对话，对它来说都是第一次见你。上次聊了三小时，它一个字都不记得。

同一次对话里也有上限。聊得太长，最早的几轮会被挤出去，它只记得后半段，还答得挺自信。

这个上限叫上下文窗口（就是它一次能同时看在眼里的字数上限）。塞满了，旧的就被挤掉，而且它不会提醒你「这段我忘了」。

- 越靠前的内容越容易被挤掉，重要的事别只放在开头
- 一个对话别干好几件不相关的事，混在一起互相干扰
- 它说「我记得你之前说过」时要留个心眼，可能记串了

## 02 做一张背景卡，十秒接上

背景卡就是一个固定格式的小文件，写清你是谁、在做什么、守什么规矩。每次开新对话，第一句就把它贴进去。别嫌麻烦，这十秒钟能省你半小时的重讲。

只留它真用得上的：项目是什么、读者是谁、风格要求、已经定下来的决定、别再犯的错。

别写成长篇自传。一屏能看完最合适，超过一屏它反而抓不住重点。格式固定下来更好用，你每次只改里面那几行字。

- 写「已经定了什么」，比写「我想要什么」更有用
- 把踩过的坑写进去，它能少犯一半重复的错
- 项目有变化就回头改这张卡，别只在对话里口头补一句

**背景卡模板**

> 【项目】给公司内部写一份新人上手指南
> 【读者】刚入职的应届生，完全不懂业务
> 【风格】短句，多举例，不用行话
> 【已定】按「第一周 / 第一个月 / 第三个月」分三段
> 【不要】不要写公司历史，全文不超过 3000 字
> 【上次的坑】它爱自己编流程名称，涉及流程一律先问我

## 03 把聊出来的东西存下来

一次长聊最值钱的往往不是最后的成品，而是中间定下的那些决定。这些必须落到文件里。比如「风格就照这版写」「数据只用去年那份」，定下来就记一笔。

最简单的做法：每聊到一个阶段，让它输出一段「阶段小结」，你复制进记事本或者文档。

下次接着干，把小结和背景卡一起贴进去，它就能接上，你不用从头讲一遍，也不怕它记岔。

- 小结只留结论和理由，过程删掉
- 一个任务一个文件，别全堆在同一个文档里
- 文件名带上日期，回退时找得到

## 能直接抄的模板

聊到一半，用这段让它整理出能带走的小结

> 先停一下，把到目前为止的进展整理成小结，格式如下：
> 1. 已经确定的事（分条列，每条一句话）
> 2. 还没定、需要我拍板的事
> 3. 你踩过或我纠正过的问题
> 4. 下一步该干什么
> 只写结论，不要复述过程，控制在 300 字以内。

## 常见坑

### × 一个对话用到天荒地老

聊到几十轮后它开始忘前文，还容易把不同话题搅在一起。一个任务一个对话，做完就开新的，把小结带过去。

### × 关键信息埋在长对话中间

中间的内容最容易被挤掉。重要约束要么放进背景卡反复贴，要么每次提问时再重述一句。

### × 指望它自己记住你的偏好

有的产品带记忆功能，但它记住的可能是你随口一句玩笑。重要偏好写进背景卡，别赌它记对了。

### × 小结写成流水账

把整段对话复制下来没用，下次贴进去照样占地方。只留结论、理由和待办，过程一律删掉。

## 做完自查

- ☐ 我知道新开对话它什么都不记得，所以第一句先贴背景卡
- ☐ 一个任务一个对话，做完就存小结，不在一个窗口里干三件事
- ☐ 重要的约束我会写死在背景卡里，不靠它自己记
- ☐ 每次阶段小结我只留结论和待办，不留过程

# 32
# MCP：给 AI 装一个统一的插口

AI 想读你的文件、查你的库，以前每个工具都得单独接一遍。MCP 是一个统一插口，接一次，换个客户端还能接着用。

### 这篇讲什么

☐ 知道 MCP 解决的是「AI 摸不到你电脑里的东西」这个具体麻烦

☐ 能看懂一份 MCP 配置，说清每一行在干什么

☐ 能自己接上一个文件或数据库工具，并按顺序验通

## 01 以前每接一个工具，都得重写一遍

你让 AI 读一下你电脑里的那张表，它读不了。你让它查一下公司数据库，它也查不了。它只能看见你贴在对话框里的那点字。

老办法是给每个工具、每个客户端单独写一套对接。工具五个、客户端三个，就得写十五套，谁改一处别人全得跟着改。

MCP 是个约定，全称 Model Context Protocol。作用是让 AI 和外部工具说同一种话。有了它，工具方做一次，谁都能接。那十五套对接代码，一下压成一套。

- 它管的是「AI 怎么拿到外面的东西」，不管 AI 本身聪不聪明
- 对你最直接的变化：装工具从写代码变成了填几行配置
- 同一个工具，今天在编辑器里用，明天换个客户端，配置基本照抄

## 02 配置就三段：叫什么、怎么启动、给什么参数

一份 MCP 配置，说到底只回答三个问题。这个工具叫什么名字，用哪条命令把它启动起来，启动时要递什么参数。

下面这段是本地文件工具的典型写法。它是个跑在你电脑上的小程序，客户端负责把它拉起来，再问它「你有哪些本事」。

名字随便起，你自己认得就行。参数里的目录就是它被允许碰的范围，写哪个目录，它就只能看见哪个目录。

**示例配置：放进客户端的配置文件里**

```
{
"mcpServers": {
"my-files": {
"command": "npx",
"args": ["-y", "@some-org/files-mcp"],
"env": { "ROOT_DIR": "D:/docs" }
}
}
}
```

## 03 装完先别急着用，分三步验

第一步，重启客户端，打开设置里的工具清单。能看见你起的那个名字，说明握手成功了。

第二步，让它干一件小事，比如「列出 D:/docs 里的文件名」。能报出真实文件名，才算真通了。

第三步才让它读内容、写内容。一步一验，出问题时你才知道卡在哪一层，不用瞎猜。

- 清单里看不到，多半是命令写错，或者那个程序压根没装
- 能连上但一用就报错，多半是权限：目录没授权，或者密钥没填
- 让它动文件之前，先用只读方式跑一遍

## 能直接抄的模板

第一次接上新工具，用这段话让它自己报状态

```
我刚接上了一个叫〈工具名〉的 MCP
工具。请按顺序做三件事，每做完一件就停下来告诉我结果：
1. 列出你现在能调用的工具，只写名字和一句话作用，不要展开。
2. 用其中只读的那个，读一下〈具体目录或表名〉，告诉我你实际看到了什么。
3. 如果第 2 步失败，把报错原文贴出来，并判断是路径问题、权限问题还是参数问题。
```

## 常见坑

### × 以为装了 MCP 就能上网

MCP 接的是你本地或内网的工具，不是互联网。要查实时信息，得另外给它接搜索类工具，这两件事别混在一起想。

### × 一口气装十几个工具

工具越多，AI 挑错工具的概率越高，也更费上下文（它一次能读进去的字数）。先装一个，用顺了再加第二个。

### × 上来就给写权限

能读就能改，改错了不好回退。第一次接，只给一个测试目录的读权限，跑通了再放开写。

### × 密钥写在配置里还传上 git

配置文件经常被一起提交。密钥走环境变量，或者单独放一个不提交的文件里，别图省事直接写死。

## 做完自查

- ☐ 我知道 MCP 只是把工具接进来，不会让模型本身变聪明
- ☐ 我的配置里没有明文密钥，也没提交到公共仓库
- ☐ 新接的工具，我先只给只读权限，验证通过再放开写
- ☐ 出问题时，我能分清是没连上、没权限，还是参数写错了

# 33
# 把多个 AI 串成一条流水线

一个提示词干不完的活，拆成三段交给三个角色，中间结果存成文件。出错能定位，改起来不用从头再来。

### 这篇讲什么

☐ 能把一件大活拆成 3 到 5 个能单独验收的小步骤

☐ 能给每一步定死输出格式，让下一步直接接得上

☐ 知道哪些环节必须留人工检查，哪些可以放心自动跑

## 01 一条流水线，就是把大活切成几小段

写一份行业调研，让 AI 一次写完，结果往往是这样：看着像样，但数据假、结构乱。

你想改一处，它整篇重写，改完前面又乱了。活越大，一次做完越不靠谱。拆开就好办：第一步只找素材，第二步只挑重点，第三步只写成稿。

每步只干一件事，干砸了一眼能看出来，退回这一步重跑就行。三段是最常见的长度，超过五段你自己就管不过来。先按三段跑一遍，跑顺了再拆细。

- 每段的输入和输出都要落在纸上，别只活在对话里
- 一段只给一个角色，别让同一步又找资料又下结论
- 每段结束都能单独验收，不合格就退回这一段重跑

## 02 中间结果得是下一段能直接吃的格式

流水线最容易断的地方，不是模型不行，是上一段的输出下一段读不懂。

所以每一步都要把格式写死。比如第一步统一输出「事实 + 出处」，第二步就照这个格式往下挑。

格式定死还有个好处。你随时能打开中间文件看一眼，知道跑到哪一步、哪条数据不对。不用重跑整条线。

- 每一步的产出单独存一个文件，名字带序号，比如 01_素材、02_要点
- 格式里用固定小标题和固定分隔符，别写「大致这样」这种模糊要求
- 在提示词开头加一句「只输出下面这个格式，不要解释」，能省掉一半返工

**三段提示词骨架，直接改字就能用**

```
# 第一步：只找素材，不下结论
输入：<原始资料>
输出格式（严格遵守，不要加别的话）：
## 事实
- 一句话事实 | 出处
## 存疑
- 一句话 | 为什么存疑

# 第二步：只挑重点
从上面「事实」里挑 5 条最能支撑结论的，按重要性排序。
每条后面加一句为什么重要，不写整句，只写短句。

# 第三步：只写成稿
用这 5 条写成 600 字短文，每条都要在文中有对应句子。
不许新增事实。写完后单独列出你用了哪 5 条。
```

## 03 哪儿必须留一个人来看

不是每一步都值得自动跑。凡是改错了代价大的地方，都要插一个人工卡点。

常见三个卡点。素材进来时看一遍真假，结论出来时看一遍站不站得住。比如素材里那个数字，AI 说是去年的，你一查是前年的，这种错只有人看得出来。

第三个是成稿发出前，看一遍语气对不对。卡点不是浪费，你花两分钟扫一眼，比错误滚到成稿再返工省事得多。

- 数字、引用、法条，一律人工核一遍再往下走
- 对外发布的稿子，最后一道必须是人签字
- 跑顺了以后再逐步减卡点，别一上来就全自动

## 能直接抄的模板

把三步接起来跑一次的完整说法

> 我们分三段做，每段做完等我确认再进下一段。
>
> 【第一段·找素材】只做一件事：从下面资料里摘事实，一条一行，格式是「事实 | 出处」。没有出处的写「无出处」，不要补资料里没有的内容。资料：〈粘贴〉
>
> 【第二段·挑重点】只做一件事：从上面的事实里挑 5 条最能支撑结论的，按重要性排序，每条后面加一句为什么重要。
>
> 【第三段·成稿】只做一件事：用这 5 条写成 600 字短文，不许新增事实。写完单独列出你用了哪 5 条。

## 常见坑

### × 一个提示词包办到底

出错时你分不清是找错了、挑错了还是写错了，只能整篇重来。拆成段，哪段烂就退回哪段。

### × 中间结果只在对话里传

聊到后面它会忘，前面辛苦找的素材就丢了。每段结束就把结果存成单独文件，下一步从文件读。

### × 格式没约定死

上一段写成散文，下一段就没法按条处理。用固定小标题和固定分隔符，别写「大致这样就行」。

### × 一上来就全自动

前几步还没稳就让它跑通宵，第二天发现全是废稿。先手动跑三遍，每步都稳了再串成一条线。

## 做完自查

- ☐ 我能说清这条流水线几步，每步的输入和输出是什么
- ☐ 每一步的产出都存成了文件，不依赖对话记忆
- ☐ 我知道哪三个地方必须我自己看一眼
- ☐ 某一步出问题时，我只重跑那一步，不从头再来

第五部分

# 部署与调优

第 34 — 39 讲

# 34
# 在自己电脑上跑模型

不联网、不花钱、资料不出门，代价是模型小一点、慢一点。先看你的电脑扛不扛得住，再决定要不要折腾。

---

**这篇讲什么**

☐ 能判断自己的电脑跑不跑得动，值不值得折腾

☐ 能把一个本地模型装起来，在命令行里跟它对话

☐ 知道本地小模型适合干什么，哪些活别交给它

## 01 先想清楚：你为什么要在本地跑

三个理由站得住。资料不能出门，比如病历、合同、客户名单。环境不能联网。长期高频用，想省点钱。

三个理由站不住：听说很酷；手上其实没有敏感资料；指望它比云端更聪明。

同样的硬件，本地能跑的模型一定比云端小。它更笨、更慢，这是换来的代价，不是设置没调好。本地跑最大的好处是资料不出门，这一条比省钱重要。

- 资料涉密、涉隐私，本地跑是刚需
- 日常写东西、改文案，用云端更划算
- 指望本地小模型追平云端最强那档，会失望

## 02 看你电脑的两个数：内存和显存

模型跑起来要先整个装进内存。装不下就跑不动，或者慢到没法用，这一步没法绕。粗略讲，模型文件多大，就得留出差不多的空闲内存。

显存（显卡自己的内存）比内存快得多。有独立显卡优先用显卡，没有就用内存硬扛，速度差好几倍。

省空间的办法叫量化（把模型里的数字精度压一压，文件能小一半以上）。聪明程度掉一点点，日常用完全够。

- 先去任务管理器看清内存和显存，再决定挑多大的模型
- 同一档模型，量化版更小更快，先试量化版
- 装不下就往下换一档，别硬撑，卡死比慢更烦人

## 03 装起来其实就三步

第一步，装一个本地跑模型的软件。这类工具会替你处理显卡驱动、模型格式这些麻烦事，不用自己编译。

第二步，在模型列表里挑一个带 7b、8b 这种小数字的。b 指十亿参数，数字越小越省资源。文件名后面常带 q4、q8 这种标记，那是量化档位，数字越小文件越小。

第三步，在命令行里把它跑起来，随便问一句。答得太慢就换小一档，答得太差就换大一档试试。

- 先用小模型把流程跑通，再换大的，别一上来就下最大的那个
- 模型文件通常好几个 G，挑网络好的时候下，别用手机热点
- 跑之前把别的吃内存的软件关掉，浏览器标签页也算

**装好软件后，在命令行里执行**

```
ollama list # 看已经下载了哪些模型
ollama run <模型名> # 启动对话，模型名从软件官网的列表里挑
ollama ps # 看模型有没有真跑起来、占了多少显存
```

## 能直接抄的模板

本地小模型容易啰嗦，用这段开场白把它管住

```
你是一个本地运行的小模型，输出越短越好。规则：
1. 先给结论，一句话，不超过 30 字。
2. 再给理由，最多 3 条，每条一行。
3. 不确定就直接写「我不确定」，不要补一段解释。
4. 不要客套话，不要重复我的问题，不要做总结。
现在等我提问。
```

## 常见坑

### × 拿它查最新消息

本地模型的知识同样停在训练那天，而且不会自己联网。要新信息，得你自己把资料贴进去。

### × 把整本长文档丢进去

本地模型一次能读进去的字数通常比云端小，塞太多它会漏。先切成小段，一段一段喂，每段只问一件事。

### × 只装模型不管散热

笔记本长时间满载会降频，越跑越慢。垫高机身、插上电源再跑，别用电池硬扛。

### × 以为装完就一劳永逸

换模型、改参数都得重启服务。改完先用 `ollama ps` 确认新模型真加载了，别在旧模型上白测半天。

## 做完自查

- ☐ 我说得出自己在本地跑的理由，不是因为新鲜
- ☐ 我确认过内存和显存，选的模型装得下
- ☐ 长文档我会切段喂，不会整本塞进去
- ☐ 涉及事实和数据，我照样自己核对，不因为它在本地就放心

# 35
# 公司内部署：数据不出门

有些数据不能上外网。这篇讲清本地部署到底要什么、要花多少钱、哪些公司真的需要，以及一个普通人怎么先试起来。

### 这篇讲什么

☐ 能判断自己手上这份数据到底能不能往外发

☐ 知道本地部署需要哪些东西，各自要多少钱、多少时间

☐ 会用一台普通电脑先跑起来一个小模型试效果

## 01 先问一句：数据到底能不能出门

本地部署的意思是，模型跑在你自己的电脑或自己公司的服务器上，不用连外网。你输入的东西，一个字节都不发给别人。

要不要这么做，看数据是什么。公开的产品文案、网上的资料、自己写的练习代码，用在线服务完全没问题，又便宜又快。

但客户名单、身份证号、合同原文、还没发布的财务数据，这些都不能出门。一旦贴到外网，就是事故，不是风险。

- 先分三档：能公开的、内部用的、绝对不能外发的
- 能公开的那一档，别本地部署，浪费钱
- 绝对不能外发的那一档，就算公司没规定，你也别贴

## 02 本地跑起来，到底要什么

要三样东西：一台有显卡的机器、一个能装模型的软件、一个能聊天的界面。三样都齐了，你就能用。

显卡是关键。模型本身是一堆数字。跑起来要把它全塞进显存（显卡上那块专用内存，比普通内存快得多）。显存不够，模型就跑不动，或者慢到没法用。

软件这块不用怕。现在有现成的工具，装好之后下载一个模型文件，点一下就启动。整个过程不用写代码。

- 普通办公笔记本：只能跑很小的模型，效果一般，适合试水
- 带独立显卡的台式机：能跑中等模型，写文案、改代码够用
- 公司服务器加专业显卡：能跑大模型，但要人维护，还得有人管权限

## 03 拿一台机器先试，别一上来就立项

很多人第一步就走错：开会、写方案、报预算，三个月过去，一行代码没跑。

正确做法是反过来的。找一台闲置的带显卡的机器，装上工具，下一个小一点的模型，拿你真实的活试半天。

试完你会知道两件事：这个效果够不够用，以及你真正需要多大的机器。这两个答案，比任何方案都值钱。

- 先用真实任务试，别用「你好，介绍一下你自己」这种题
- 至少试 20 条真实任务，才看得出稳不稳
- 效果不够就换大一号的模型，别急着换硬件

**示例：三条命令把模型跑在自己电脑上**

```
# 在本地机器上跑一个模型（需要先装好 Ollama）
# 1. 下载并安装 Ollama，装完它会在后台跑着

# 2. 拉一个中等大小的模型下来（第一次会下载几个 G）
ollama pull qwen2.5:7b

# 3. 直接开聊
ollama run qwen2.5:7b

# 4. 也可以用接口方式调用，方便接进自己的脚本
curl http://localhost:11434/api/generate -d '{
"model": "qwen2.5:7b",
"prompt": "把这段话改得更正式：这个方案我觉得不太行",
"stream": false
}'
```

## 能直接抄的模板

拿这段去问 IT 或者老板，把该问的一次问清

> 我们想在公司内部跑一个 AI 模型，处理不外发的数据。请帮忙确认几件事：
> 1. 公司现在有没有闲置的、带独立显卡的机器？显卡型号和显存多大？
> 2. 这台机器能不能装新软件、能不能下载模型文件（有些公司网络会拦）？
> 3. 谁负责这台机器的日常维护和重启？
> 4. 如果用的人不止我一个，账号和权限怎么分？
> 5. 有没有规定说哪些数据绝对不能进这个系统？
> 先给我一台机器试两周，我拿真实任务跑一遍，再报需要多少资源。

## 常见坑

× **以为本地部署就是免费**

机器要钱，电费要钱，维护的人也要钱。一台能跑中等模型的机器，加上一年的维护，比买在线服务贵得多。数据不能外发才值得这么做，不然纯亏。

× **买了机器才发现模型装不上**

公司网络经常拦下载，显卡驱动也可能跟系统打架。先在一台机器上把整个流程跑通，再谈采购。

× **拿最小的模型试，试完说不行**

模型太小，效果确实差，但这不能说明本地部署不行。试的时候至少用中等大小的模型，否则结论是错的。

× **以为本地部署就绝对安全**

模型在你机器上，但聊天记录、导出的文件、截的图还在你手上。该管的权限、该删的临时文件，一样都不能少。

## 做完自查

- ☐ 我确认过手上这份数据属于哪一档，能不能外发
- ☐ 我知道这台机器的显卡型号和显存大小
- ☐ 我用至少 20 条真实任务试过效果，不是随便聊两句
- ☐ 我知道这台机器谁维护、出问题找谁

# 36
# 让模型跑得更快

同一个模型，有人等十秒有人等两秒。这篇讲清楚慢在哪、哪些地方能自己调、哪些只能花钱换机器。

### 这篇讲什么

☐ 能看出一次回答慢在排队、慢在生成、还是慢在你自己这边

☐ 会用几个不改代码的办法把等待时间砍掉一半

☐ 知道什么情况下该换模型、什么情况下该换机器

## 01 先搞清楚，到底慢在哪一段

等一次回答，时间花在三个地方。等它开口、它一个字一个字往外蹦、以及你自己这边处理结果。

等它开口这段时间叫「首字延迟」（从按下回车到屏幕上出现第一个字，中间那段空白）。这段时间跟你的问题长短关系很大。

它开始蹦字之后，速度基本固定了。所以想快，主要靠两件事：让它少说废话，让它早点开口。

- 问一句「你好」也要等很久：问题在排队或者网络
- 开头快、后面越写越慢：正常，它就是这么工作的
- 复制粘贴结果时卡：跟模型无关，是你电脑或者网页的问题

## 02 不改代码也能快一倍

最常见的浪费，是让它写一堆你根本不要的东西。「请详细说明」「请分十个方面展开」，这些字全是它现场生成的，每个字都要时间。

第二个浪费，是背景信息给太多。你把三千字的文档整篇贴进去，它每次都要从头读一遍。

第三个浪费，是让它先想再答。有些模型默认会先输出一大段思考过程，你不看那段，但时间照样花掉了。

- 明确写「控制在 100 字以内」，比事后删快得多
- 长文档先自己摘出关键三段，别整篇丢进去
- 能关掉「深度思考」就关掉，简单任务不需要

**示例：一段通用的「省字」尾巴**

```
# 加在提示词末尾，直接砍掉大半生成时间

输出要求：
- 只要结论，不要铺垫，不要客套话
- 总长度不超过 120 字
- 不要重复我的问题
- 不要解释你为什么这么回答
- 不要列「首先、其次、最后」这种结构

如果你需要更多信息才能回答，只回一句「缺：xxx」，不要猜。
```

## 03 该换模型还是换机器

如果你用的是在线服务，慢通常不是你的问题。换个时间段、换个更小的模型，往往立刻见效。

如果你在自己机器上跑，慢基本就是显卡不够。这时候先降模型大小，效果掉一点但速度快很多，多数日常活完全够。

顺序别搞反：先改提示词，再换小模型，最后才换硬件。前面两步不花钱，第三步最贵。还有个笨办法很管用：把常用的几套提示词存成文档，用的时候直接粘。省下现编的时间，比等它快几秒实在。

- 日常改文案、写邮件：用小模型，快而且够用
- 写复杂代码、做长文分析：用大模型，值得等
- 同一个任务固定用同一个模型，别来回换，方便你判断快慢

## 能直接抄的模板

把这段贴在提问前面，逼它少说话

> 回答前请遵守：
> 1. 先说结论，一句话说完。
> 2. 最多再补三条理由，每条不超过 20 字。
> 3. 不要复述我的问题，不要开场白，不要总结段。
> 4. 总字数控制在 150 字以内。
> 5. 我追问细节时再展开，第一次回答不要展开。
>
> 我的问题是：

## 常见坑

× **一次问十个问题**

它得一口气全答，输出越长等得越久，而且答到后面容易乱。拆成十次问，每次都快，还能随时改方向。

× **反复让它「再详细一点」**

每追问一轮，前面所有内容它都要重新读一遍。要细节就一次说清，别挤牙膏。

× **在高峰期硬等**

在线服务在大家上班的时间段最慢。能挪到早上或者晚上做的事，就别挤在下午三点。

× **以为换个大模型就一定更快**

大模型通常更慢。快慢和聪明是两回事，看你的任务要哪一个。

## 做完自查

☐ 我知道自己这次等的时间，是等开口还是等写完

☐ 我在提示词里写清了字数上限

☐ 我没有把整篇长文档原样贴进去

☐ 我先试过改提示词和换小模型，才考虑换机器

# 37
# 微调：教它你自己的东西

不用重新训练一个大模型。给它几十条你自己的例子，让它学会你的写法、你的行话、你的格式。这篇讲什么活该微调、什么活不该。

### 这篇讲什么

☐ 能分清「该微调」和「该写提示词」这两种情况

☐ 知道要准备多少条例子、怎么挑例子

☐ 会写一份能直接交给技术同事或平台的数据格式

## 01 大多数时候，你根本不需要微调

微调的意思是，拿你自己的例子，再去训练一下现成的模型，让它更贴合你的场景。你不用从头训一个大模型，成本比很多人想的低。

但先想清楚：你要它学的是「知识」还是「风格」。要它知道公司有哪些产品、报价多少，这些写进提示词就行，不用微调。

写提示词当场就能改，微调要重新跑一遍，改起来慢。但如果是这几件事：按公司固定格式写周报、按审核员的口气回客户、把行话翻成标准说法。这种「手感」类的活，微调才值得。

- 事实类知识：写提示词，或者接一个文档库，别微调
- 格式和口气：微调很有效
- 只改几个字的任务：写提示词更快，改起来也方便

## 02 例子怎么挑，比训练本身更重要

微调吃的就是例子。你要准备一批「输入」和「理想输出」成对的样本。

数量上，一般来说几十条就能看出效果，两三百条算比较稳。关键是质量，不是数量。

挑例子有个原则：挑你希望它学会的那种，别把平时的烂例子也塞进去。你喂它什么，它就学什么，包括你的坏习惯。

- 例子要覆盖你真实会遇到的各种情况，别全挑简单的
- 输出要由人来定稿，别拿上一版模型的输出当标准答案
- 留 20 条不参与训练，用来验收效果

**示例：一条样本长这样，一行一个 JSON**

```
{
"messages": [
{"role": "system", "content":
"你是某公司的客服，回复要简短、专业、不带感叹号。"},
{"role": "user", "content": "你们这个多少钱"},
{"role": "assistant", "content": "您好，标准版每年 XXX 元，含 5
个账号。需要我发一份详细报价吗？"}
]
}
{
"messages": [
{"role": "system", "content":
"你是某公司的客服，回复要简短、专业、不带感叹号。"},
{"role": "user", "content": "能不能便宜点"},
{"role": "assistant", "content":
"价格是统一的，不过年付可以少一个月。要按年付算一下吗？"}
]
}
```

## 03 做完怎么验收，别看感觉

微调最容易出的问题是「学过头」。格式学得很像，但别的能力变差了，问它别的事就开始胡说。

所以验收要两头看。一头看你留下的那 20 条，它答得对不对。另一头拿几个完全无关的问题试，看它有没有变笨。

还有一件事：微调不会让它记住新知识。你上周改的价格，它照样不知道。该走提示词的部分还得走提示词。

- 验收用固定的一套题，每次微调都跑同一套，方便对比

- 两个版本一起测，一个原版一个微调版，别只测新的
- 微调后提示词里照样要交代背景，两者不冲突。效果只好了两成就先别上

## 能直接抄的模板

准备数据前，先用这段把需求说清楚

> 我要微调一个模型，任务是：（一句话写清，比如「把客户的抱怨改写成内部工单标题」）
>
> 输入长这样：（贴 2 条真实输入）
> 理想输出长这样：（贴 2 条你亲手写的输出）
>
> 请帮我：
> 1. 判断这个任务该用微调还是写提示词，给理由。
> 2. 如果要微调，列出我需要准备多少条样本，以及样本要覆盖哪几类情况。
> 3. 给我一个样本的格式模板，我照着填。
> 4. 告诉我验收时该测哪些题，包括几道跟任务无关的题。

## 常见坑

× **想让它记住公司的新规定**

微调学的是说话方式，不是记忆。规定、价格、产品清单这类会变的东西，写进提示词或者接文档库，改了立刻生效。

× **只准备十几条例子**

太少了，模型学不到规律，只会背下来。一般来说几十条起步，两三百条更稳，而且要覆盖不同情况。

× **拿模型自己的输出当标准答案**

等于让它学自己的毛病，越练越歪。训练用的输出必须是人定稿的。

× **微调完不测别的能力**

可能格式变好了，但正常问答变差了。验收一定要加几道无关的题，跟原版对比着看。

## 做完自查

☐ 我想清楚了这是「学风格」还是「学知识」，后者我不微调

☐ 我的样本输出是人写的，不是模型自己生成的

☐ 我留了一批样本没参与训练，专门用来验收

☐ 我拿无关的问题测过，确认它没变笨

# 38
# 微调前先准备好数据

微调（拿自己的例子再教它一遍）成不成，八成看数据。这篇讲数据从哪翻、要多少条、怎么洗干净。

### 这篇讲什么

☐ 知道该从哪些旧记录里翻出真能用的数据，而不是上网找

☐ 能凑出一份 100 条以上、干净、不重复的训练数据

☐ 知道怎么留考试卷、怎么存档，避开「自己考自己」的假高分

## 01 数据从哪来：翻你自己的旧记录

别上网找现成的问答集。风格不是你的，学出来说话四不像，用户一眼能看出不对。

好数据藏在你已有的东西里：客服聊天记录、工单、邮件往来、群里的问答。还有你自己手写过的方案。这些都是真人真事，说法也真实。

先攒 200 条原始素材，最后能用的可能只剩 100 条。清洗会掉量，心里要有数。

- 优先挑反复做过很多遍的活，那种记录量才够
- 对话两头都要存：用户怎么问，你当时怎么答
- 没有旧记录就从今天开始攒，让同事把好回复存到一个共享文档

## 02 一条数据长什么样，要多少条

一条数据就是一对：你发给它的话，和你想让它回的话。行话叫「样本」。格式通常是一问一答。

数量上，同一个活先准备 100 到 300 条。少于 100 条，它学不稳；超过 500 条，提升开始变慢。

覆盖比数量重要。常见的 20 种情况各来 5 条，比一种情况来 100 条强得多。

- 同一个意思要有好几种问法，长的短的都要有
- 答案的语气、长度、格式要统一。用 Markdown 就全都用
- 故意留几条「答不了」的样本，教它什么时候该说不清楚

## 03 洗数据：删、脱敏、留考试卷、存档

删重复。意思一样的留一条，重复太多它会被带偏，最后只会那一种答法。

删错的。你自己都不认可的答案别放进去，它会照学，包括你当时的敷衍和口误。

脱敏。真名换「张三」，电话换「138****0000」，金额按比例缩。然后留考试卷：随机抽 10% 到 20%，训练时一条都不许用。

- 存成 JSONL（一行一条记录的文本文件），工具能直接读
- 训练用的和考试用的存两个文件，文件名带日期，别互相覆盖
- 开头留一行说明：谁做的、什么时候、一共多少条、抽走多少条

**数据台账长这样，边整理边填（最后两行是分用途）**

```
序号,来源,场景,用户问题,我的回答,是否重复,是否脱敏,用途
1,客服聊天,退款,你们的课能退吗,可以退。开课后 7 天内、听课不超过 2
节,否,是,训练
2,客服聊天,退款,买了课不想学了咋办,可以退。开课后 7 天内、听课不超过 2
节,否,是,训练
3,客服聊天,发票,能开发票吗,可以，下单后在订单页点申请,否,是,考试
```

## 能直接抄的模板

把这段发给 AI，让它帮你把杂乱的问答整理成训练数据

```
我给你一批真实的问答记录。请帮我整理成训练数据，规则如下：
1. 一条一行，输出成 JSONL，字段叫 messages，里面是 user 和 assistant 两条。
2. 意思重复的只留一条，其余删掉，并告诉我删了几条。
3. 人名换成「张三」，手机号换成「138****0000」，公司名换成「A 公司」。
4. 答案里如果缺关键信息，标成「待补充」，不要自己编。
5. 按场景分类，告诉我每类各有多少条，哪类太少需要补。
```

## 常见坑

× **上网找问答集凑数**

风格不是你的，学出来四不像。宁可少一点，也要用你自己的聊天记录、邮件、工单，哪怕是手写的。

× **只准备一种问法**

上线后用户换个说法它就不认了。同一个意思至少写 5 种问法，长短都要有。

× **训练和考试用同一批数据**

分数会高得离谱，一上线就露馅。必须留 10% 到 20% 出来，训练时完全不用。

× **答案写得又长又漂亮**

你写 800 字，它就学 800 字。按真实场景该多长就写多长，客服回复别写成小作文。

## 做完自查

☐ 我的数据来自真实记录，不是网上扒的问答集

☐ 我能凑出至少 100 条「用户这么问 + 我想它这么答」，并且覆盖多种情况

☐ 重复的、答错的都删了，敏感信息也换成了假的

☐ 我留了 10% 到 20% 没参与训练，专门用来考试

# 39
# 怎么知道它到底行不行：评测

别凭感觉说「好像变好了」。评测（用一批固定的题反复打分）才能告诉你，改之前和改之后差多少，差在哪。

### 这篇讲什么

☐ 能自己出 30 道题，凑成一套可以反复用的考试卷

☐ 知道该记「通过率」，而不是记几个漂亮例子

☐ 换模型、改提示词、加检索，都能用同一把尺子比

## 01 先认一件事：感觉不算数

你改了一句提示词，试了三个例子，觉得「好多了」。这三个例子是你自己挑的，当然好。

上线以后，用户问的东西你根本想不到。要比，就得用一批固定的题，每次都跑同一批。光靠印象不行，印象永远偏向最近那几次。

这套题加上打分规则，就叫评测集（一套固定的考题，配上一份「什么算过关」的标准）。它像体检报告：同一个项目，前后对比才有意义。

- 题目从真实记录里翻，别自己编
- 起步 30 到 50 题就够，不用一上来搞几百题
- 每题都要写清楚「什么算过关」

## 02 题怎么出，分怎么打

去翻你的真实记录：客服聊天、群里问过的问题、同事发你的需求。挑 30 条最有代表性的，常见的和刁钻的都要有。别挑你答得最漂亮的那几条。

打分分两种。能自动判的，写脚本判。比如答案里必须有某个字段、必须是合法 JSON（一种固定写法的数据格式）。判不了的，让另一个模型当裁判，按 1 到 5 分打。

最后只记一个数：通过率。30 题过了 21 题，就是 70%。改完变成 24 题，就是 80%。这才叫变好了。

- 让模型当裁判，评分标准要写进提示词，别只说「打个分」
- 同一批题跑两遍，结果差太多说明题不稳，得改题
- 每次的日期、改了什么、通过率，记在同一张表里

## 03 总分好看，不代表没问题

70% 不代表还行。要看那 30% 错的是什么。

把错题分个类：没看懂问题、答得太啰嗦、编了不存在的信息、格式不对。哪类最多先修哪类，修完再跑一轮，看这一类有没有降下来。

还有一个容易漏的指标：该说「不知道」的时候，它说了没有。这类题至少放 5 条进去。

- 错题要一条条看，别只盯着那个百分比
- 改一处跑一轮。同时改三处，你不知道是哪处起了作用
- 换模型和改提示词别一起上，不然比不出结果

**算通过率的小脚本（results.jsonl 里每条记一题的得分和题型）**

```
# 跑完一轮，把结果算成一个数
import json
rows = [json.loads(x) for x in open('results.jsonl', encoding='utf-8')]
passed = sum(1 for r in rows if r['score'] == 1)
print(f"{passed}/{len(rows)} = {passed/len(rows):.0%}")
# 再按题型分开看，找出整类都不过的
for t in sorted({r['type'] for r in rows}):
sub = [r for r in rows if r['type'] == t]
print(t, sum(1 for r in sub if r['score'] == 1), '/', len(sub))
```

## 能直接抄的模板

让模型当裁判的打分提示词，复制就能用

```
你是一个严格的评分员。下面给你一道题、几个标准答案要点、一个待评答案。
请按这三点打分，每点 0 或 1 分：
1. 有没有正面回答用户的问题，答非所问记 0。
2. 有没有出现标准答案要点里没有的具体信息，比如数字、日期、链接，编了就记 0。
3. 格式对不对：要求 JSON 就必须是合法 JSON，要求 3 行以内就不能超。
只输出一行：分数|理由。理由不超过 30 字。

题目：{{question}}
标准答案要点：{{expected}}
待评答案：{{answer}}
```

## 常见坑

### × 只挑成功的例子看

这是自欺欺人。全部题都要跑完，难看的也得记下来，否则你永远不知道真实水平。

### × 一次改三样

换了模型又改了提示词，分数涨了也不知道是谁的功劳。一次只改一处，跑完再动下一处。

### × 把考题写进提示词

分数会高得离谱，上线立刻崩。考题要藏起来，改提示词的人不能对着题目改。

### × 只看平均分

有一类题全军覆没，平均分照样好看。要按题型分开看，找出「整类都不行」的地方。

## 做完自查

- ☐ 我有一套 30 题以上的固定题，都来自真实场景
- ☐ 每题都写清了什么算过关，不是凭感觉判
- ☐ 我记了改动前后的通过率，而且一次只改一样
- ☐ 错题我逐条看过，并按类型归了类

第六部分

# 安全、协作与上线

第 40 — 54 讲

# 40
# 安全底线：什么不能做

用 AI 有几条线不能踩：别人的隐私、公司的机密、要你签字的事。这篇讲哪些别交出去，以及怎么防。

---

### 这篇讲什么

☐ 知道哪三类信息绝对不能贴进对话框

☐ 能给团队定一条三句话、真能执行的规矩

☐ 知道 AI 给的东西哪些必须自己再核一遍

## 01 三类东西，别往对话框里贴

第一类，别人的隐私。客户真名、手机号、身份证号、住址、病历、聊天记录。这些贴进去，等于把别人的信息交出去了，而且你收不回来。

第二类，公司的机密。没公开的财报、合同条款、源代码、还没发布的产品方案、内部报价。这类东西漏出去，损失不是一句道歉能补的。

第三类，能直接换钱的东西。密钥（一串能让程序登录账号的密码）、数据库密码、后台地址、内部系统的截图。

- 拿不准算不算机密，就按机密处理
- 非用不可就先换成假的：真名换「张三」，公司名换「A 公司」，金额按比例缩
- 用完把这段对话删掉，别留在历史记录里

## 02 这些活它能帮，但字得你自己签

法律、医疗、财税、投资这几类，AI 能帮你理解概念、理清思路、列问题清单。但它不能替你做决定。

它不知道你的具体情况，也可能不知道最新的规定。它给的条文编号、药名剂量、税率，都可能是编的。

正确用法：让它把「你该问什么」列出来，你拿着清单去问真正有资质的人。它负责帮你想全问题，不负责给结论。

- 涉及签字、付款、报税、吃药的，AI 的输出只能当草稿
- 它引用的法条、论文、链接，一条条点开核实
- 对外发的正式文件，必须有人从头到尾读过一遍

## 03 给团队定一条真能执行的规矩

规矩要短，短到大家记得住。比如：客户信息一律用代号，公司文件一律不外传，AI
写的东西发出去前必须有人看。这三句写下来贴在群公告里，比开会强调管用。

光有规矩没用，得有人能问。指定一个「拿不准就找他」的人，比写十页制度管用。这个人得说得上话，不能是刚来的实习生。

再加一条兜底：万一已经贴出去了，别瞒着，第一时间说。早说还有补救的余地，瞒着只会让事情更糟。

- 把允许用的工具列成一张清单，清单外的先别用
- 新同事第一天就讲这三条，别塞进手册第 8 页
- 每季度抽查几条对话记录，看有没有贴敏感信息

## 能直接抄的模板

这段直接贴到团队群公告里，就是一份能用的规矩

> 用 AI 的三条规矩，请大家照做：
> 1. 客户和同事的真实姓名、电话、证件号、住址，一律换成代号再用。
> 2. 没公开的合同、财报、代码、报价，一律不发。
> 3. AI 写的东西对外发出前，必须有人从头到尾读一遍。
> 拿不准就问［负责人名字］。已经贴出去了别瞒着，第一时间说，还来得及处理。

## 常见坑

### × 「就贴一点点」

一小段聊天记录里就有客户手机号和订单号。要么整段脱敏，要么换个方式描述。

× **以为删了对话就没事**

输入可能已经被记录下来。贴之前想清楚，别指望事后删干净。

× **拿 AI 的法律意见当真**

它给的条文编号经常是编的，读起来还特别像。当参考，别当依据。

× **公司没规定就默认可以**

没规定不等于允许。先问一句，比自己猜安全得多。

## 做完自查

☐ 我贴进去的内容里，没有别人的姓名、电话、证件号、住址

☐ 没有未公开的合同、财报、代码和报价

☐ 没有密钥、密码、后台地址

☐ AI 写的东西对外发出前，有人从头读过一遍

# 41
# 别把敏感信息漏出去

你随手贴进去的代码、日志、客户名单，可能已经离开了公司。这篇讲清楚哪些东西不能给 AI 看，必须用时又该怎么换成假的。

### 这篇讲什么

☐ 能一眼认出哪些内容属于不能外发的敏感信息

☐ 学会把真实数据换成假数据再提问，拿回来的结果照样能用

☐ 知道公司没规定时，自己该怎么定一条能落地的规矩

## 01 先搞清楚，你贴的东西去哪儿了

你把一段代码粘进输入框，按下回车，这段字就发到了别人家的服务器上。它不是在你电脑里算的。

大部分工具默认会保存对话记录，有的还会拿去改进模型。留多久、给谁看，各家不一样，而且随时可能改。

所以判断标准很简单：这段话如果明天出现在公司群里，你会不会慌？会慌，就别贴。

- 能公开的随便贴：开源库的代码、网上的报错、自己写的练习题
- 不能公开的一律打码：客户名、手机号、身份证、订单号、内部域名、密钥
- 拿不准就当不能。多花两分钟改一遍，比事后解释便宜

## 02 最常见的三种漏法

第一种，整段日志直接粘。日志里通常带着服务器地址、数据库账号、用户编号，你以为只是让它看个报错。

第二种，把配置文件整个发过去。里面往往有密钥（就是一串能直接登录你账号的密码）和第三方服务的令牌。

第三种最隐蔽：截图。截图里带着聊天窗口的其他人名、浏览器标签页、右下角弹出的通知。你以为只截了代码。

- 贴日志前，先把 IP、域名、账号、用户编号换成 example.com 这种假值
- 要贴配置文件，就把 key、token 后面的值删掉，只留字段名
- 截图前先关掉其他窗口，或者干脆把文字复制出来，别用图

## 03 必须用时，就把数据换成假的

大多数时候你不需要真数据。AI 要的是结构，不是内容。把真名换成张三，把金额换成 100，它一样能帮你把逻辑捋顺。

换完自己心里留一张对照表。它给回来的方案是照着假数据写的，你套回真数据时，检查一遍字段有没有对错位。

如果确实要用真数据，就走公司批过的内部工具。没有内部工具就别用，这不是技术问题，是流程问题。

- 假数据要保留原来的形状：手机号还是 11 位，日期照原样写
- 别用「客户A、客户B」，容易串。用「客户_001」这种带编号的，回头好对
- 提问时加一句：下面的数据都是假的，你只分析结构，不用纠结具体值

**示例代码：把日志里的敏感信息批量替换掉**

```
import re

def mask(text):
text = re.sub(r'\b\d{11}\b', '[手机号]', text)
text = re.sub(r'\b[\w.-]+@[\w.-]+\.\w+\b', '[邮箱]', text)
text = re.sub(r'\b\d{1,3}(\.\d{1,3}){3}\b', '[IP]', text)
text = re.sub(r'(?i)(key|token|secret|password)\s*[=:]\s*\S+', r'\1=[已隐藏]',
text)
return text

with open('app.log', encoding='utf-8') as f:
print(mask(f.read()))
```

## 能直接抄的模板

让 AI 帮你写一个对得上你日志格式的打码脚本

> 我有一批日志要发给别人看，但不能带敏感信息。日志每行大概长这样：
> 2024-01-01 10:00:00 INFO user=13800000000 ip=10.0.0.1 msg=login ok
>
> 请帮我写一个 Python 脚本，做这几件事：
> 1. 把手机号、IP、邮箱、身份证号替换成 [已隐藏]。
> 2. 把 key=、token= 后面的值替换成 [已隐藏]，等号前面的字保留。
> 3. 其他内容一个字都不要改，包括空格和换行。
> 4. 脚本读一个输入文件，写出一个新文件，不要动原文件。
>
> 先只输出脚本，不要解释。

## 常见坑

### × 以为删了对话就没事

删掉的只是你这边看到的记录，服务端可能还留着备份，也可能早就进了训练数据。发出去那一刻就收不回来了，别把删除当补救。

### × 只改了一处就发

同一份代码里，客户名往往出现在注释、变量名、测试数据三四个地方。贴之前搜一遍关键词，别凭印象。

### × 用私人账号问公司项目

公司代码贴进个人账号，等于把内部资料搬到公司管不着的地方。项目的事就用项目给的账号和工具。

### × 把密钥贴进去让它帮你调

调不通十有八九是权限或者格式问题，贴密钥解决不了。真要看，就换成等长的假字符，保持格式一样。

## 做完自查

☐ 贴之前我问过自己：这段要是明天出现在公司群里，我会不会慌

☐ 日志、配置、报错里的手机号、IP、账号、密钥，我都换成了假值

☐ 我用的是假数据，但字段格式和真的一样，拿回来的方案能直接套

☐ 项目相关的活，我只用公司批过的账号和工具

# 42
# 怎么发现它在胡说

它说得越顺，你越容易信。这篇给你几个能立刻上手的动作，几分钟就能把一段回答里的假话挑出来。

### 这篇讲什么

☐ 学会用追问细节的办法，让编出来的内容自己露馅

☐ 知道哪几类内容必须核实，不用句句都查

☐ 能把一段回答拆成「能直接用的」和「要自己验证的」两部分

## 01 它胡说的时候，为什么特别像真的

因为它写东西的目标是读起来顺，不是说得对。这两件事在它那里是分开的。

一段编造的内容，只要句式通顺、用词专业，看上去和真的一模一样。你读的时候，判断的其实是顺不顺，不是对不对。

所以想抓它的错，光读没用，得靠问。读只能让你觉得舒服，问才能让它露出马脚。

- 越具体越可疑：人名、年份、编号、链接、法条、引用
- 越冷门越可疑：它熟的东西翻来覆去就那些，一进小众领域就开始编
- 越像总结越可疑：「研究表明」「业内普遍认为」却不说谁研究、谁认为的，多半是凑字数

## 02 三招，几分钟就能试出来

第一招，追问来源。它说「有研究表明」，你就问：哪篇研究，谁写的，发在哪，哪一年。真知道的答得上来，编的会开始绕。

第二招，反着问。把它刚说的结论倒过来问一遍：那如果情况正好反过来呢？真懂的会告诉你边界在哪，编的会顺着你改口。

第三招，让它拆步骤。让它把结论是怎么一步步推出来的写清楚，中间每一步用了什么数据。编出来的东西一拆就散。

- 追问来源时，注意它有没有换说法。从「研究表明」变成「一般来说」，就是心虚
- 反着问时，如果它两边都讲得很有道理，那这段结论基本没信息量
- 拆步骤时，盯着「所以」前面那一句。假话经常就藏在那一步

## 03 哪些必须查，哪些可以放过

不用每句都查，那样比自己写还慢。按后果分就够了：错了会让你返工的，查；错了只是读着别扭的，放过。

必须查的是：能直接抄进代码里的命令和参数。对外发的文案里的事实和数据，也得查。涉及钱和规矩的内容，一律自己确认。

可以放过的是：帮你理思路的框架、改语气、解释一个你本来就懂的概念。这些错了你一眼就能看出来。

- 代码先跑一遍再说。它给的函数名、参数名，一半的错跑一次就现形
- 数字和日期一律自己核对，这是它出错最集中的地方
- 涉及别人的话、承诺、报价，一个字都别信它，全部自己确认

**示例提示词：让它自己交代哪儿不靠谱**

> 下面这段是你刚写的回答。请按这个格式重新过一遍：
>
> 1. 事实清单：把里面所有具体的人名、机构名、年份、数字、链接单独列出来。
> 2. 确定程度：每条后面标「确定」「大概」「不确定」。
> 3. 出处：能说清出处的写出处；说不清的，直接写「我没有可靠出处」。
> 4. 改口：把「研究表明」「业内普遍认为」这类没有主语的句子，改成你实际能负责的说法。
>
> 不要重写正文，只输出这份清单。

## 能直接抄的模板

拿不准的时候，用这段让同一个问题答两遍再对照

> 请分两部分回答，两部分分开写，不要混在一起。
>
> 第一部分：用你自己的话给出结论和理由，能多具体就多具体。
>
> 第二部分：现在请你扮演一个专门挑错的人，针对第一部分逐句找漏洞。逐句看这四点：没依据、可能过时、把特例说成通例、没有主语的「研究表明」。
>
> 最后单独用一句话说明：第一部分里最不可靠的是哪一句。

## 常见坑

× **它说得越专业越信**

术语多、句式正式，恰恰是最容易糊弄人的组合。判断标准换成「它能不能说清出处和步骤」，别用「读起来像不像专家」。

× **问一次就完事**

只问一遍，你拿到的是它最有把握的那版。换个说法再问一遍，两次答案对不上的地方，就是它心里没底的地方。

× **拿它当搜索引擎用**

它不查网，靠的是训练时记下的东西，可能已经过时好几年。要最新信息就自己搜，或者用带联网的工具，并且点开链接看原文。

× **它认错了就以为全对了**

被指出来就道歉，是它的默认反应，不代表它真改了。让它重写之后，按同样的办法再查一遍，别因为态度好就放过。

## 做完自查

☐ 具体的数字、日期、链接、人名，我一个都没直接抄，全部自己核过

☐ 我问过它这个结论哪来的，它给的是能查的出处，不是换了个说法

☐ 它给的代码我跑过一遍，报错的地方改完再用

☐ 对外发的内容，事实部分我自己确认过，不是它说啥我写啥

# 43
# 一份清单：AI 编程最常见的坑

用 AI 写代码翻车，八成不是它笨，是开头没说清、中间没小步走、收尾没验。这篇按开工到收工的顺序给你一条条过。

### 这篇讲什么

- ☐ 知道每次开新对话前必须先交代哪几件事
- ☐ 学会把一个大需求切成能验收的小块，改坏了能退回去
- ☐ 拿到一套收尾检查动作，判断这段代码到底能不能用

## 01 开工前，先把话说清楚

AI 写代码最常见的失败，不是它笨，是你没说清。它不知道你用什么语言、跑在哪儿、跟什么库配合。

有个能省一半返工的动作：开头先给它一段项目背景，五行字就够。用什么语言、什么框架、什么数据库、代码跑在本地还是服务器、有哪些地方不能动。

这段背景存起来，每次开新对话第一句就贴上。比每次重新解释省事得多。

- 说清不许改哪些文件。不然它会顺手把你没让它碰的地方一起重写了
- 一次只提一个需求。三个需求挤在一段话里，它经常只做对前两个
- 给它看报错原文，别自己转述成「跑不起来」

## 02 写的时候，小步走，每步都能退回去

让它一口气写五百行，你基本没法验收。正确做法是切小块：先让它写一个能跑通的最小版本，你跑一下，再让它加下一个功能。

每次改动前先提交一次，也就是给当前代码存一个存档点。它改坏了，你一条命令就能退回去，不用一行行找。

它给的代码，先看它改了哪几个文件、动了哪几行。看不懂的地方就让它解释，别整段复制粘贴了事。

- 一次改一个地方。同时动三个文件，出了问题你不知道是谁的锅
- 让它把改动列成清单：改了哪个文件、加了删了哪些函数、为什么这么改
- 没跑通就别接着加功能。带着毛病往前写，后面全是补丁

**示例提示词：让它按小步走的方式干活**

> 我们一步一步来，每次只做一件事。
>
> 第一步：先告诉我你打算怎么改，改哪几个文件、每个文件动什么，先不要写代码。
> 我确认之后，你再写。
>
> 写的时候遵守这几条：
> 1. 只改我指定的文件，其他文件一个字都不要动。
> 2. 改动的地方用注释标出来「改动开始」「改动结束」。
> 3. 写完列一个清单：改了哪些文件、新增了什么、删掉了什么。
> 4. 有你觉得该改但我没提的地方，单独列出来问我，不要自己动手。

## 03 收尾，怎么判断这代码能不能用

能跑不等于对。跑通了只说明这一条路上没摔，别的路可能全是坑。

最低要求是让它写几个测试用例，你自己跑一遍。测试用例就是自动检查「输入这个、应该输出那个」的小程序。重点看边界：空值、零、特别大的数、重复提交。

还有一件常被跳过的事：让它说清这段代码的假设。比如假设 `id` 不会重复、假设时区都是北京时间。这些假设一旦不成立，代码就会悄悄出错。

- 让它列出这段代码在什么情况下会出错，比你自己想快得多
- 涉及钱、权限、删数据的代码，逐行读，不要跳
- 上线前问一句：这段代码最坏会把什么弄坏？它答不上来就说明你没看懂

## 能直接抄的模板

每次收工前的最后一问，直接复制

> 这段代码写完了，请你做三件事：
>
> 1. 列出你在这里做的所有假设：数据格式、取值范围、时区、并发情况。每条说明假设不成立时会发生什么。
> 2. 列出 5 个最可能出问题的输入，并说明每个输入下代码会怎么走。
> 3. 写 5 个测试用例，分别覆盖：正常情况、空值、超大值、重复操作、权限不足。用我能直接运行的格式写，不要伪代码。

## 常见坑

### × 一口气要一个完整项目

它给出来的东西看着齐全，其实各处对不上，你还得从头读一遍。改成先要骨架，跑通，再一个功能一个功能往上加。

### × 不看改动直接粘贴

它可能删掉你没注意的校验，也可能把配置写死。每次粘贴前看一眼改了哪些文件、哪几行，改动大的就让它先解释再给代码。

### × 报错只发一句跑不起来

它只能靠猜。把完整报错、你运行的那条命令、相关的那几行代码一起发过去，一次说清比来回五轮快。

### × 不存档就让它大改

改坏了没法退回，只能顺着它继续修，越修越乱。动手前先提交一次，或者把文件复制一份存着，这是一分钟的保险。

## 做完自查

- ☐ 每次开新对话，我第一句就贴上项目背景：语言、框架、跑在哪、哪些不能动
- ☐ 我一次只让它做一件事，做完跑通再加下一件
- ☐ 它改完我会看改了哪些文件、哪几行，看不懂就让它解释
- ☐ 收工前我拿到了测试用例和假设清单，并且自己跑过一遍

# 44
# 团队一起用 AI 的规矩

三个人以上一起用 AI，最容易出两种乱子：资料贴错地方，出错没人认账。这篇给你一套能直接抄的规矩，半天就能落地。

### 这篇讲什么

☐ 能跟团队定下三条谁都不会争的底线

☐ 知道提示词该存哪些、怎么起名才找得到

☐ 能说清 AI 写的东西出错时该谁负责

## 01 先把三条底线定下来

团队用 AI 出乱子，八成不是技术问题，是没规矩。先定三条，写进工作群置顶，新人第一天就看得到。

第一条：客户资料、合同、报价、员工信息，一个字都不许贴进对话框。要试就换成假的，名字写「张三」，金额改成 100。

第二条：对外发的东西必须有人从头看过。第三条：AI 写的初稿要标出来，别混进自己的成果里。三条加起来不到十行字，比开一次会管用。

- 底线一：真实数据不出门，举例一律用假数据
- 底线二：对外交付前，必须有一个真人读完一遍
- 底线三：文档里标一行「初稿由 AI 生成，已人工修改」

## 02 好用的提示词，得有个公共抽屉

一个人调好的问法，别人不知道，等于没有。建一个共享文档，谁调出好用的就往里放。

每条只写四样：干什么用、原话怎么问、要替换哪几个词、效果大概什么样。别写心得，写能直接复制的东西。

名字统一成「用途-场景」，比如「周报-销售团队」。三个月后你还搜得到，这就是全部意义。

- 十个人的团队，能长期用起来的通常不超过 20 条，别贪多
- 每条下面留一行「最后验证日期」，超过半年没人用的就删掉
- 新人第一天得能自己找到这个文档，找不到就等于没建

## 03 AI 是初稿，签字的是人

谁把东西发出去，谁负责。AI 不背锅，也背不了，责任只能落到具体的人头上。

具体做法：让它干完活，你自己按三样过一遍。数字对不对，有没有编出来的引用，口气像不像我们。

加起来花不了五分钟，能挡住绝大多数丢人的错。团队里最好固定成一句话：「这版你看过了吗？」

- 数字、人名、日期、链接，这四样必须自己核对一遍
- 口气不对就让它按你们以前的成稿重写，别自己硬改

## 能直接抄的模板

团队 AI 使用约定，复制到共享文档里就能用

```
# 我们团队的 AI 使用约定
一、不做的：不贴客户资料、合同、报价、员工信息；不让 AI
直接回客户；不把没看过的内容交出去。
二、要做的：对外交付前真人读完并改成人话；文档末尾标「初稿由 AI
生成，已由（姓名）修改」。
三、提示词存共享文档，每条写用途、原话、要替换的词、效果。
四、同一类错出现两次，就补一条新约定。
```

## 常见坑

### × 各调各的提示词

十个人十套问法，好用的没人知道，踩过的坑轮流再踩。至少有个共享文档，谁调出好的就贴进去，写清用途和要替换的词。

× **让 AI 直接回客户**

口气一飘、数字一错，损失的是信任。AI
只能出草稿，发出去之前必须有人读一遍，并改成像人说的话。

× **拿真客户名单试效果**

贴进去就可能被存下来。要试就造一份假数据，列名一样、内容全假，试完把真的填回去。

× **没有验收标准**

只说「帮我写个方案」，交回来谁也不知道算不算合格。开工前先写清交什么、多长、给谁看、什么算过关。

## 做完自查

☐ 我们团队有写下来的底线，新人也看得到

☐ 共享提示词库里每条都能直接复制，不是空话

☐ 对外交付的东西，我能说清最后是谁看过的

☐ 我试 AI 的时候用的是假数据，不是真实客户信息

# 45
# 一个能直接开工的项目模板

每开一个新任务都要重新跟 AI 解释背景，太费时间。这篇给你四个文件夹加一份说明文件，照着填，五分钟就能开工。

### 这篇讲什么

☐ 能用固定结构，把散在各处的资料收进同一个地方

☐ 能写出一份让 AI 一读就懂的说明文件

☐ 知道第一轮对话该说什么，少绕两三轮弯

## 01 先建四个文件夹，名字固定

不管做什么项目，都建这四个：00-素材、01-产出、02-参考、09-归档。名字带数字，排序永远不乱。

00-素材放原始文件：客户给的表格、会议纪要、聊天截图。只放不改，要改就另存一份进 01。

01-产出放你做出来的东西，文件名带日期，比如「20250310-周报-初稿」。02-参考放别人的东西，09-归档放做完的。归档不是删，是挪走，三个月后要找还在。

- 素材只读不改，是这套结构里最要紧的一条
- 文件名统一成「日期-内容-状态」，状态就三个：初稿、待审、终版
- 空文件夹也先建出来，位置定了，东西才收得进去，别等有文件了再想往哪放

## 02 写一份项目说明，让 AI 先读它

每开一个新对话，你都得把背景重新交代一遍，这是最浪费时间的环节。解决办法是把它写成文件，写一次，以后每次复制。

这份文件只用七行。做什么、给谁看、最终交什么、有什么硬性限制。再加上：不要什么、现在到哪一步、下一步要它帮什么。

写完存成「项目说明.md」，放在项目根目录。七行写完通常不到两百字，一屏就能看完。

- 写不出来，说明你还没想清，先花十分钟想清楚再动手
- 说明文件改了，就在群里说一句，别让人拿着旧版本干活

## 03 第一轮对话，先让它复述

贴完说明文件，别急着让它干活。加一句：先复述一遍你理解的任务，再问我三个不清楚的地方。

它复述错了，你一眼就能看出来，改一句话就行。等它写完三千字你才发现方向偏了，那才叫费时间。

确认没问题再让它动手。这一步多花两分钟，通常能省掉两轮返工。返工两轮，一下午就没了。

- 让它提问，比让它猜要准得多
- 一次只让它做一件事，你确认了再给下一件
- 它答不上来的时候，就是你说明文件没写清的地方

## 能直接抄的模板

开工第一句，连同项目说明一起贴给 AI

> 先读下面这份项目说明，不要马上开始写。
>
> 第一步：用三句话复述你理解的任务目标。
> 第二步：列出三个你还不清楚、需要我补充的地方。
> 第三步：等我确认之后，再按下面的要求开始。
>
> 【项目说明】
> - 要做什么：
> - 给谁看：
> - 最终交付：
> - 硬性限制：字数、截止时间、必须用的格式
> - 不要什么：不要口语、不要表情、不要超过 3 页
> - 现在到哪一步：
> - 已完成：
>
> 【这次要做的事】
> 只写一件具体的事，写清交付物的样子。

## 常见坑

### × 要求全在聊天记录里

上周定的规矩，这周就翻不到了。凡是定下来的事，一律写进项目说明文件，聊天记录不算存档。

### × 说明写得像作文

一大段形容词，AI
读完还是不知道要交什么。写成清单，每条一句，字数、格式、截稿日都写进去。

### × 一上来就让它写全文

方向错了，三千字全废。先要提纲，或者先要第一段，看着对了再往下铺。

### × 所有文件塞一个文件夹

素材、产出、别人的参考混在一起，找一次文件花十分钟。按四个文件夹分开，文件名带日期和状态。

## 做完自查

☐ 我知道这个项目的素材在哪、产出放哪

☐ 我有项目说明文件，新开对话第一件事是贴它

☐ 我会先让它复述任务，再让它动手

☐ 我的文件名带日期和状态，别人接手也看得懂

# 46
# 攒一个自己的代码片段库

同一件事查第三遍，就该存下来。这篇讲怎么攒一个三个月后还找得到、还看得懂的片段库，不用装任何软件。

### 这篇讲什么

☐ 能判断哪些东西值得存，哪些存了也是垃圾

☐ 能给每条片段写清三行说明，半年后自己还看得懂

☐ 知道怎么让 AI 直接读你的库，不用每次重新描述

## 01 只存会用第二次的东西

判断标准很简单：同一件事，你是不是第三次在查了？是，就存。第一次见就存，库很快变成垃圾场。

值得存的有：常用的表格公式、查日志的命令、一段正则（匹配文字用的规则）。还有每次都要翻文档才想起来的配置。

不值得存的：网上搜一下就有的大路货、只用一次的临时脚本、你自己都看不懂的老代码。存了也不会再看。

- 库超过 200 条就该清一次，真正常用的通常不超过 30 条
- 拿不准就问自己：三个月后我还用得上吗。答不上来就别存
- 先存够 20 条再谈分类，一开始就分类会把自己卡住

## 02 每条写三行，别写一篇

片段库最怕一堆没头没尾的代码。半年后打开，只看见一堆符号，想不起来当时在想什么。

每条固定写四行：干什么、什么时候用、哪里要改、拿什么词能搜到。第三行最要紧，写清哪几个词要换成你自己的。

第四行是触发词，就是你会拿什么词去找它。比如「去重」「导出 Excel」。触发词写两三个就够，写多了反而想不起来搜哪个。

- 说明写在片段上面，不要写在下面，扫一眼就先看到用途
- 把「哪里要改」写具体：列名、文件名、路径，别写「按需修改」

## 03 让 AI 直接读你的库

存下来只是第一步，用得上才算数。最简单的办法：把整个库导成一个文本文件，提问时一起贴进去。

开头加一句：这是我的常用片段，先看一遍，按我的需求挑合适的改，不要另写一套。

这样它给的东西跟你的习惯一致，你也不用每次从零描述。库越大，这一句越省事。

- 整个库存成一个文件，别拆成一堆，贴的时候省事
- 每季度把最常用的 10 条提到最前面，你和 AI 都更容易找到
- 它新写的好东西，当场问一句要不要加进库，别等以后

## 能直接抄的模板

提问时先贴这段，再贴你的片段库

> 下面是我的常用片段库，先读一遍，不要急着回答。
>
> 规则：
> 1. 能用我库里现成的，就优先用，并说明用的是哪一条。
> 2. 需要改的，只改必要的部分，其余保持我的写法。
> 3. 库里没有合适的再新写，并告诉我这条值不值得加进库里。
> 4. 不要引入我没用过的工具或写法。
>
> 【我的片段库】
> （把文件内容整段贴在这里）
>
> 【这次的需求】
> （写清要做什么、结果长什么样）

## 常见坑

× **看见什么存什么**

三个月后库里有五百条，一条都找不到。只存第三次还在查的东西，其余的随它去。

× **只存代码不写说明**

半年后打开只看见一堆符号，不敢用也不敢删。每条至少写清干什么、什么时候用、哪里要改。

× **分类分得太细**

建了二十个分类，存的时候不知道往哪放，最后全堆在桌面。分类别超过五个，靠触发词搜更靠谱，别建目录树。

× **存完就不管了**

环境一变、工具一升级，老片段可能直接报错。每季度把常用的那几条跑一遍，跑不通的就地修好或删掉。

## 做完自查

- ☐ 我存东西之前会问自己：这是第三次在查了吗
- ☐ 每条片段都写了干什么、什么时候用、哪里要改
- ☐ 我能靠一个词就搜到想找的那条片段
- ☐ 我每季度清一次库，删掉跑不通的和没用过的

# 47
# 用 AI 写脚本，把重复活干掉

每天点同样的按钮、改同样格式的表，让 AI 给你写个小脚本。一次写完，以后天天省事。

### 这篇讲什么

☐ 能看出自己手上的活哪些适合写成脚本

☐ 会让 AI 写出能直接跑、你也大概看得懂的脚本

☐ 知道脚本跑之前该试什么、该备份什么

## 01 先挑那种「每周都要干好几遍」的活

什么叫重复活？步骤一模一样，只是文件名或数字变了。比如把十个表格的列宽字号统一，把一堆截图改成同样尺寸，把几十份 Word 转成 PDF。

这种活最值得交给脚本。你手动干一次二十分钟，脚本跑一次两秒。写脚本花半小时，第二周就回本了，之后每次跑都是白赚。

反过来，一个月才做一次、每次做法还不一样的活，别写。写脚本花的时间比省下来的还多。拿不准就先手做两遍，看看它到底重不重复。

- 先数一遍：一周做三次以上、步骤固定，就值得写
- 步骤要能用嘴说明白。说不明白的，先自己走一遍，把每一步记下来
- 留一份真实文件当样本，让 AI 照着它写

## 02 跟 AI 要脚本，三件事得说全

只说「帮我写个批量改文件名的脚本」，它只能猜。猜出来的能跑，但不是你要的。

你得交代三件事：文件放在哪、现在长什么样、你要变成什么样。再加一句「别动原文件」。这三样说清楚，它给的代码八成能直接跑。

还得说清你的电脑情况，比如「我用 Windows，装了 Python，不想再装别的」。它才知道该给你哪种写法，不然给你的代码你连跑都跑不起来。

**示例提示词**

```
我有个文件夹，路径是 D:\报表\原始。
里面全是 Excel 文件，文件名像「2024年1月销售.xlsx」。
我要做三件事：
1. 文件名改成 2024-01-销售.xlsx 这种格式
2. 每个文件里，第一行字体加粗、背景改成浅灰
3. 结果另存到 D:\报表\输出，原文件一个都不要动

我的电脑是 Windows，已经装了 Python。
请给我一个能直接运行的脚本，每段代码前面加一句中文说明。
跑之前先告诉我它会动到哪些文件。
```

## 03 第一次跑，先拿三个文件试水

别一上来就对着整个文件夹跑。复制三个文件到新文件夹，在那儿先跑一遍，这一步能挡掉八成的坑。

跑完打开看：名字对不对、内容坏没坏、有没有多出奇怪的文件。三个都对，再上全量。别只看第一个就放心，第一个往往是最规矩的那个。

让脚本先备份。跟 AI 说「改之前把原文件复制到 backup 文件夹」，它顺手就写了。真改坏了，你还捞得回来。

- 先在样本文件夹跑通，再跑全部
- 重要文件夹，跑之前自己手动复制一份，别只信脚本
- 跑完抽查五个文件，别只看第一个

## 能直接抄的模板

让 AI 写脚本时，照这个格式把话说完

> 帮我写一个脚本，要求如下：
> 1. 输入：文件夹「填你的路径」，里面是「填文件类型」，大概「填数量」个
> 2. 每个文件要做：「填第一步」「填第二步」
> 3. 输出：另存到「填输出路径」，文件名规则是「填规则」
> 4. 原文件不要改动，跑之前先复制一份到 backup 文件夹
> 5. 碰到处理不了的文件，跳过并告诉我它的名字
> 6. 每段代码前面加一句中文说明，最后告诉我怎么运行

## 常见坑

### × 一句话就要一个脚本

信息太少，它只能猜你的文件长什么样。写出来的代码看着像样，一跑就报错。文件名、格式、要改成什么样，一条条写清楚，比来回改五遍都快。

### × 看不懂也直接跑

代码里可能藏着删除、移动的命令。让 AI 逐行讲一遍，遇到 del、rm、move 这些词，问清它到底动哪些文件、动完还能不能找回来。问不明白就先别跑。

### × 拿正式文件当试验品

第一次就对着几百个正式文件跑，出错找不回来。先复制三五个到临时文件夹，试对了再跑全部，多花三分钟省掉一整天。

### × 报错就自己瞎改

看不懂的报错，原样复制给 AI，说明你做了什么、想得到什么结果。让它改，比你自己猜快十倍，也不会把好的部分改坏。

## 做完自查

- ☐ 我能说清这个活一周做几次、步骤是不是固定
- ☐ 我的提示里写了文件在哪、什么格式、要变成什么样
- ☐ 第一次跑之前，我把样本复制到了单独的文件夹，没有直接动正式文件
- ☐ 涉及删除和移动的代码，我让 AI 逐行解释过

# 48
# 一次处理一大批：批处理

一百个文件挨个改太慢。把活拆成「同一套动作重复一百次」，AI 帮你把套路写成能跑的东西。

### 这篇讲什么

☐ 能把一件大活拆成「处理一个」加「重复很多次」

☐ 知道一次跑多少合适，跑崩了怎么接着往下跑

☐ 会给批处理加上「出错就跳过并记录」，不中途全停

## 01 把大活拆成：处理一个，重复很多次

一百份合同要换成新模板，一百张图要加水印，听着都是大事。拆开看只有两件小事：把一份改好，再把这件事重复一百遍。

第二件几乎不用你操心，电脑最擅长的就是这个。你只要把第一件说清楚，剩下九十九遍它自己会跑，而且跑得比你手快，也不会漏。

拆的笨办法：拿张纸，把「改一份」的每一步写下来，写到不用想为止。这张纸就是给 AI 的需求说明，比你在脑子里过一遍靠谱。

- 先跑通一个，再跑十个，最后跑全部
- 文件名要有规律，比如 2024-01、2024-02，脚本才好一个个抓
- 结果另起一个文件夹放，别覆盖原件

## 02 一次跑多少，心里得有数

别一次丢一千个进去。先跑十个，看花了多久、结果对不对。十个没问题，再上五十，再上一百。用十个的耗时，你还能估出全量要多久。

跑的时候人别走开。很多活跑到一半断掉，就是因为人离开了、电脑自动睡了。把电源设置里的睡眠关掉，或者干脆去泡杯茶再回来。

跑完对一下数量：进去多少个，出来多少个。两个数对不上，说明中间有文件被漏掉了，这时候失败清单就是你的线索。

**示例代码**

```
import os, time

src = r"D:\合同\原始"
dst = r"D:\合同\输出"
os.makedirs(dst, exist_ok=True)

files = [f for f in os.listdir(src) if f.endswith(".docx")]
total = len(files)
failed = []

for i, name in enumerate(files, 1):
try:
# 这里换成「处理一个文件」的那段代码
print(f"{i}/{total} 完成：{name}")
except Exception as e:
failed.append(f"{name} -> {e}")
print(f"{i}/{total} 跳过：{name}")
time.sleep(0.2)

with open(os.path.join(dst, "失败清单.txt"), "w", encoding="utf-8") as f:
f.write("\n".join(failed) or "全部成功")

print(f"结束：成功 {total - len(failed)} 个，失败 {len(failed)} 个")
```

## 03 半路出错，别让它整个停下

一百个文件里总有一两个怪的：加密的、空的、名字里带奇怪符号的。默认情况下，一个出错整批就停在那儿。

让 AI 加一层保护：碰到错的就跳过，把名字记进「失败清单」，剩下的接着跑。这类文件占比很小，却能卡住整批。

跑完只看清单。那一两个单独处理，比从头再来一遍省事得多。同一套活留一份跑成功的脚本，下次改两个路径就能用。

- 要求它输出失败清单，写清哪个文件、什么错

- 同一批别反复跑，越跑越乱
- 没报错不等于全好了，数量对得上才算

## 能直接抄的模板

把一大批活交给 AI 时，这样描述

> 我要处理一个文件夹里的文件，每个文件都要做同样的事：
> 1. 打开「填文件类型」
> 2. 做「填第一步」和「填第二步」
> 3. 另存到「输出」文件夹，文件名保持不变
>
> 请按这些要求写：
> - 每处理完一个，在屏幕上打印「3/100 完成：文件名」
> - 碰到打不开的，跳过并记到「失败清单.txt」，不要整个停下来
> - 不要改动原文件
> - 最后打印一行总结：成功几个、失败几个
> - 先只处理前 10 个给我看，我确认没问题再跑全部

## 常见坑

× **一上来就全量跑**

一千个文件跑一半崩了，你不知道哪些改过、哪些没改。先跑十个，确认对了再往上加。

× **直接覆盖原文件**

改错了原件就没了，连对照的样本都不剩。让脚本写到新文件夹，跑完核对过再决定要不要删原件。

× **只看有没有报错**

程序安静跑完，不代表每个文件都成功了。进去一百个、出来九十七个，那三个得一个个找出来。

× **文件名乱七八糟也硬批**

名字没规律，脚本容易漏掉，也容易把不该处理的临时文件一起改了。先花十分钟统一命名，比事后找漏的快得多。

## 做完自查

☐ 我先跑通了「一个」，才加上重复，没有直接全量开跑

☐ 输出写在新文件夹，原文件没动

☐ 跑完我核对过进去和出来的数量

☐ 我让脚本出错时跳过并记录，而不是整个停下

# 49
# 把 AI 接到前端页面里

想让网页上那个输入框真的会说话，得把页面接到模型上。这篇讲清三件事：发什么、密钥放哪、怎么边写边出。

### 这篇讲什么

- ☐ 知道一次对话请求里到底装了什么内容
- ☐ 知道密钥为什么绝对不能写在前端代码里
- ☐ 能做出「边生成边显示」的效果，而不是干等十几秒

## 01 页面发出去的那一下，到底发了什么

按下发送以后，页面发出去的其实不是一句话。它是一小段固定格式的文字，叫 JSON（电脑之间传数据常用的一种写法）。

里面装三样东西：用哪个模型、之前聊过什么、这次说了什么。新手常漏掉中间那两样，一漏就出现「它怎么不记得我刚说的」。

还有件事容易忘：模型自己不记事。所谓「它记得」，是你每次都把之前的对话重新发了一遍。这个动作得你自己做，模型那边没有存。

**示例代码**

```
// 前端只管收集输入、显示回答，密钥不放在这里
async function ask(question) {
const res = await fetch("/api/chat", {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({
history: recentTurns, // 最近几轮对话
message: question // 用户这次说的
})
});
if (!res.ok) throw new Error("请求失败");
const data = await res.json();
return data.text; // 把正文抠出来
}
```

## 02 密钥千万别写在前端

调模型需要一个密钥（一串字符，相当于你账号的密码）。有人图省事，直接写在页面的代码里，测试时确实方便，上线就是事故。

页面代码用户都看得到。按 F12 或者右键看源码，密钥就明摆在那儿。别人拿走就能刷你的额度，你第二天才发现账单不对。

正确做法是中间加一层自己的服务端（一台服务器或云函数，专门替页面去调模型）。页面只跟它说话，密钥留在它那里。

- 前端代码里出现密钥，等于把家门钥匙挂在门把手上
- 让服务端顺手做两件事：限制单次请求长度、限制同一人每分钟问几次
- 密钥放环境变量里，别写死在代码里，也别提交到代码仓库

## 03 别让用户干等，让它边写边出

模型是一段一段往外吐字的。默认设置下它全写完才一次性给你，用户就盯着转圈十几秒。

把「流式」打开（让字一段一段实时送过来），页面上能看到字一个个冒出来。等待时间一样，感受差很远。

再补个细节：点发送后立刻清空输入框、显示「正在想」。用户就不会以为按钮没反应，反复点好几次，白白多花几份钱。

- 先把「正在生成」这个状态做出来，比什么动画都管用
- 生成中把发送按钮禁掉，防止连点
- 出错时页面上说人话，别把一长串英文报错甩给用户

## 能直接抄的模板

让 AI 帮你搭这一层时，照这个说

> 我要在一个网页里接入大模型，请按下面几点给我代码：
> 1. 前端只负责收集输入、显示回答，不放任何密钥
> 2. 中间加一个服务端接口，密钥从环境变量读
> 3. 回答要流式返回，页面上一段一段显示出来
> 4. 请求失败时，页面显示一句人话提示，并给一个重试按钮
> 5. 每次请求只带最近几轮对话，不要无限堆
> 6. 最后给我一份本地怎么跑起来的步骤，一步一步写清楚

## 常见坑

× **密钥写在前端**

前端代码用户全看得见，密钥等于公开。放到服务端，页面只调你自己的接口，密钥永远不出你的机器。

× **每次把整段聊天都发一遍**

聊得越久，每次发的东西越多，越慢越贵。只带最近几轮，早前的对话压成一小段摘要再发。

× **不做超时也不做重试**

网络一抖，页面就卡在那儿一直转，用户只能刷新。设个十几秒的超时，失败了给一个「再试一次」的按钮。

× **把报错原文直接甩给用户**

用户看不懂英文报错，只会觉得你的东西坏了。翻成「刚才没连上，请再试一次」，详细信息记进自己的日志。

## 做完自查

- ☐ 我检查过前端代码里没有任何密钥
- ☐ 请求失败时用户看到的是人话，不是一串英文
- ☐ 生成过程中发送按钮是禁用的，用户不会连点
- ☐ 我知道每次请求带了哪些上下文，不会无限往里堆

# 50
# 把 AI 接到后端服务里

接 AI 不用懂模型。让服务器替你去发请求，把钥匙藏好，把超时和重试管住，就算接完了。

### 这篇讲什么

☐ 知道后端接 AI 只是多一次网络请求，不神秘

☐ 能把 key 藏在服务器上，不让它出现在前端

☐ 会设超时、重试和兜底，模型挂了页面也不白屏

## 01 说白了，就是服务器替你去发个请求

你在页面上点一下按钮，浏览器把话发给你自己的服务器。服务器再转手发给模型厂商，拿回答案，送回浏览器。中间多出来的这一跳，就是「后端接 AI」。

为什么不干脆让浏览器直接调？因为要带一把钥匙。这把钥匙叫 API key（调用模型时出示的一串密码），谁拿到它，谁就能花你的钱。

所以规矩只有一条：钥匙只放在服务器上。浏览器只知道你服务器的地址，永远看不到钥匙。

- 一次调用就是一次 HTTPS 请求，跟你调支付、发短信没本质区别
- 请求里就三样：用哪个模型、你说的话、最多回多长
- 返回的是一段文本，存哪儿、怎么展示，都得你自己管

## 02 钥匙怎么藏，才不会漏出去

最常见的事故，是把 key 写进了前端代码。前端代码是公开的，谁打开浏览器都能看见，等于把银行卡密码贴在门上。

正确做法是放进环境变量（服务器上单独存密码的地方，不写进代码文件）。代码运行时去读，代码里一个字都不出现。

再稳一点：你自己做一层接口，只允许登录用户、只允许固定几个动作。别做成「传什么都能问」的万能通道。

- key 写在 .env 文件里，并把 .env 加进 .gitignore
- 打日志时把 key 遮掉，只留前几位
- 给这把 key 设个额度上限，出事至少烧不穿

**示例代码：后端转发一次请求（Python）**

```
# key 从环境变量读，代码里不出现明文
import os, requests

def ask(prompt: str) -> str:
r = requests.post(
"https://你的服务商地址/v1/chat/completions",
headers={"Authorization": f"Bearer {os.environ['AI_KEY']}"},
json={
"model": "你的模型名",
"messages": [{"role": "user", "content": prompt}],
"max_tokens": 500,
},
timeout=30,
)
r.raise_for_status()
return r.json()["choices"][0]["message"]["content"]
```

## 03 它一定会挂，先把退路想好

模型服务会超时、会限流（短时间内请求太多，对方直接拒收）、会突然报错。这不是意外，是日常。

三件事必须做：设超时（比如 30 秒就放弃）、失败重试、兜底话术。重试最多 2 次，间隔一次比一次长。

再加一层限制：同一个用户一分钟最多问几次。不然有人刷你的接口，账单会很难看。

- 超时别设太长，用户等 30 秒就想关页面了
- 只对超时和服务端错误重试，参数写错了重试一百次也没用

- 每次调用的字数记进日志，月底对账全靠它

## 能直接抄的模板

复制这段：给调用套上超时和重试

```
import time

def ask_with_retry(prompt, tries=3):
for i in range(tries):
try:
return ask(prompt)
except Exception:
if i == tries - 1:
return "服务有点忙，稍后再试一次"
time.sleep(2 ** i) # 等 1 秒、2 秒、4 秒再试
```

## 常见坑

× **把 key 写进前端**

打开浏览器就能看到，等于公开。key
只放服务器环境变量，前端只跟你自己的接口说话。

× **做一个万能接口**

传什么都能问，等于把模型白送给别人。接口里写死角色和任务，只收必要的几个字段。

× **不设超时**

对方不回应，你的进程就一直挂着，几个人一卡服务就瘫。超时必须设，30 秒是常见值。

× **让用户干等长任务**

生成一份长报告要几十秒，别让页面转圈。先返回「已提交」，后台跑完再推送通知。

## 做完自查

- ☐ key 只在服务器上，前端代码和仓库里搜不到它
- ☐ 接口只做固定的几件事，不是谁传什么都能问
- ☐ 超时、重试、兜底话术都写了，断网时页面不白屏
- ☐ 每次调用的字数和耗时都进了日志，月底能对账

# 51
# 手机上怎么调 AI

手机上调 AI，图的是随手：走路时用嘴说，看到东西拍一张。关键是把常用提示词存成短语，别每次重打。

### 这篇讲什么

☐ 知道手机上哪三类活最值得交给 AI

☐ 会把常用提示词存成输入法短语，打几个字母调出来

☐ 分得清哪些事手机上做，哪些事回电脑做

## 01 手机上最值钱的是「随手」两个字

地铁上想到一句话，掏出手机说一遍，让它帮你理顺。开会时拍张白板照，让它整理成三条结论。这些场景电脑替代不了。

手机的优势不是算力，是「你人就在现场」。站着、走着、等着的时候，它都能用。灵感、见闻、临时要回的消息，都发生在手机上。

所以别在手机上做长活。写五千字方案、改一堆表格，还是回电脑，屏幕小、切来切去太累。

- 用嘴说比打字快好几倍，长一点的提问直接语音输入
- 拍照加一句「这是什么、怎么用」，比搜关键词快
- 一次只让它干一件小事，小屏幕上才看得清、改得动

## 02 把常用的话存成短语，别再手打

每次重打一大段提示词，最劝退。手机输入法一般都有「文本替换」或「快捷短语」（打几个字母就自动展开成一整段话）。

做法很简单：把提示词存进去，起个短名字。比如输入 gw，就展开成一段固定的改写要求，后面只补原文。

存五到十条就够用，覆盖你最常干的活：改文案、写周报、翻译、总结会议记录、润色邮件。别贪多，用不上的存了也记不住。

- 短语名字取你不常用的字母组合，免得平时打字被误展开
- 把「字数」「语气」「给谁看」写死在短语里，省得每次补
- 换手机前记得导出，输入法一般都能备份

**示例：一条可以存成短语的提示词（存进输入法，起名 gw）**

> 把下面这段改写成给客户看的版本：
> 1. 去掉内部术语，换成客户听得懂的说法
> 2. 控制在 200 字内
> 3. 只留结论和要客户做什么，别铺垫
>
> 原文：

## 03 手机负责收，电脑负责干

手机适合起头：随手记一句、拍张照、录段语音，先扔进一个地方存着。别急着整理，那时候你也整理不动。

回到电脑再统一处理。云笔记、自己的对话记录都行，关键是「随手记」和「认真改」分开。

长对话别在手机上硬撑。屏幕小，翻记录费劲，聊到后面你自己也记不清前面说了啥。

- 用云笔记当一个中转箱，手机往里扔，电脑里改
- 重要结果当天就整理，别攒到周末，细节全忘光了
- 一次性的问题问完就用，不用费劲存；要反复用的才值得存

## 能直接抄的模板

复制这段：手机语音提问的开场白，说完直接跟你的问题

> 我在路上，用语音说的，可能有错字，你先按意思理解，别纠结错别字。
> 下面是我的问题，请先用一句话给结论，再补三条理由，每条不超过 20 字：

## 常见坑

× **在手机上写长文**

屏幕小、光标难定位，改起来比写还累。手机上只出提纲和短句，成稿回电脑。

× **语音输入不说明白**

语音常出错字，AI 会照着错字理解。开头先说一句「可能有错字，按意思理解」。

× **拿聊天记录当笔记本**

对话一多就翻不到，还容易误删。定好一个地方存结果，比如云笔记里的一个文件夹。

× **在外面随便发敏感内容**

外面的公共 Wi-Fi 不一定安全。客户资料、合同这类内容，等回到可信网络再弄。

## 做完自查

☐ 我把最常用的 5 条提示词存成了输入法短语

☐ 我知道手机上只干短活，长文和表格回电脑

☐ 语音提问时我会先说一句「可能有错字」

☐ 重要结果当天就存进云笔记，不留在对话里，下次找得到

# 52
# 算清楚一个 AI 功能要花多少钱

AI 的钱按字数算，输入输出都要钱。这篇教你用一条公式估出一次请求几分钱，再找出最费钱的那一处。

### 这篇讲什么

☐ 会用一个公式算出一次请求大概多少钱

☐ 知道输入、输出、上下文各自怎么影响账单

☐ 能找到三个最省钱的地方，知道先改哪个

## 01 钱是按字算的，不是按次算的

模型厂商按 token 收钱。token（可以理解成「字块」，一个汉字大概算一个到两个）是计费的最小单位。

注意两笔都收费：你发过去的算输入，它回的算输出。而且输出一般比输入贵，常见是两三倍。

所以「让它多说点」是要花钱的。你只想要一句结论，就别让它写八百字。省钱的第一刀，往往是少让它写。预算该看「一个月多少字」，不是「一次多少钱」。

- 一千个汉字，大概一千到一千五百个 token，按上限估更保险
- 每次对话都要把前面的记录一起发过去，聊得越久，输入越贵
- 图片、长文档也算钱，一张图往往顶好几段文字

## 02 一次请求几厘钱，动手算一遍

公式就一条：费用 = 输入字数 × 输入单价 + 输出字数 × 输出单价。单价在服务商后台能查到，抄下来代进去就行。

举个例子，数字是假设的。输入每百万 token 收 1 元，输出每百万 token 收 4 元。

一次请求按输入 2000 字、输出 500 字算。代入就是 0.002 元加 0.002 元，等于 0.004 元。

一次四厘。一天一万次，就是四十块。一个月一千二。

- 先估「一天多少次」，再乘单价，比盯着单价更有用
- 输出字数最容易被忽略，限死 max_tokens 能省一大块
- 两个数字填进下面这段代码，跑一下就出结果

**示例代码：把单价代进去，算出月账单**

```
INPUT_PRICE = 1.0 # 每百万 token 的输入价，换成你后台的
OUTPUT_PRICE = 4.0 # 每百万 token 的输出价

in_tok, out_tok = 2000, 500 # 一次请求大概用多少
times_per_day = 10000 # 一天调用多少次

per_call = in_tok / 1e6 * INPUT_PRICE + out_tok / 1e6 * OUTPUT_PRICE
print(f"一次 {per_call:.4f} 元，一个月约 {per_call * times_per_day * 30:.0f}
元")
```

## 03 三个地方最省钱，从最费钱的改

第一，砍上下文。别每次把整份文档塞进去，先切出相关的那几段。这一项通常省得最多。

第二，换小模型。改错别字、分类、抽取字段这种活，便宜的那档就够用，贵的那档留给真难的问题。

第三，缓存（把算过的结果存起来，下次同样的问题直接拿）。固定的问法、公共资料，存一份就行。命中一次就省一次钱。

- 先看日志里哪类请求最多，从最多的那类下手
- 把输出长度限死，模型就不会自由发挥
- 一次只改一项，改完再看一遍账单，才知道省了多少

## 能直接抄的模板

复制这段：让 AI 帮你核对成本估算有没有漏项

> 我在估算一个 AI 功能的成本，请帮我检查有没有漏算。
> 已知条件：
> - 一次请求输入约 ___ token，输出约 ___ token
> - 一天调用 ___ 次
> - 输入单价 ___ 元/百万 token
> - 输出单价 ___ 元/百万 token
> 请做三件事：
> 1. 算出一次请求和一个月（按 30 天）的费用。
> 2. 指出我可能漏掉的花费项。
> 3. 给出三个能立刻降成本的具体改法，每条写明大概能省几成。

## 常见坑

× **只算输出不算输入**

输入往往比输出多得多，长对话里更是。两头都要算，别只盯回答那几个字。

× **把整份文档塞进去**

每次都塞几万字，钱全花在重复内容上。先切出相关段落，或者用检索帮你挑。

× **什么都用最贵的模型**

简单活用贵模型，等于开卡车买菜。分类、抽取、改错别字，小模型够用。

× **上线后才发现账单炸了**

上线前用真实请求跑一百次，按实际字数估。上线后按天看用量，设一条告警线。

## 做完自查

- ☐ 我能用一条公式算出一次请求和一个月大概多少钱
- ☐ 输入和输出我都算了，输出还限了长度
- ☐ 我知道自己最费钱的是哪一类请求
- ☐ 上线前跑过真实请求，上线后每天看用量

# 53
# 上线前必须过一遍的检查表

上线前一小时，照这张表走一遍。把最贵的那几种错挡在门外，比上线之后再救火省事十倍。

### 这篇讲什么

☐ 能给自己手头的项目列出一份真正用得上的上线前清单

☐ 知道哪些错一旦发生就收不回来，必须排在最前面挡住

☐ 会让 AI 一条一条陪你查，而不是听它说一句「看起来没问题」

## 01 先想清楚：哪些错一上线就收不回

有些错改一下就好，有些错一发生就收不回来。先把后者挑出来，检查才有重点。

比如删掉用户的数据、把测试用的配置发到线上、把密钥写进代码里。这三类错，事后道歉也补不回来。

拿张纸写三行：最坏会丢什么、最坏会被谁看到、最坏要花多少时间补。写完你就知道先查哪儿了，这三行不用给任何人看。

- 数据能不能恢复，比界面好不好看重要得多
- 凡是「点一下就发出去」的动作，都多加一道确认
- 先查收不回来的，再查改得回来的

## 02 让 AI 当检查员，别让它当盖章的

很多人上线前会问 AI 一句「你看还有问题吗」。它多半回你「整体没问题」。这不是检查，这是求安慰。

正确用法是给它一份具体清单，让它一条一条答。答不上来就写「这条我查不了」，不许糊过去。

问题要问得死板。比如「有没有写死的密码」「出错时用户会看到什么」。问得越具体，它越没法敷衍你。

- 一次只问一条，问完再问下一条，别一次塞二十条
- 要求它说出在哪个文件、哪个页面，指不出位置就是没查清
- 它说「没问题」时，追问一句「你凭什么说没问题」

## 03 自己动手走一遍，别只看代码

AI 说好不算好，你自己当一次新用户。用手机、用别人的账号、把网断掉，各试一遍。

走三条路。最顺的那条：从注册到真正用上。最容易翻车的那条：输错密码、传个超大文件、半路断网。最不该走通的那条：没登录直接打开后台。

每步记三件事：点了什么、等了几秒、看到什么。超过三秒没动静，用户已经走了。

- 在别人的手机上试一遍，你手机里全是登录状态，看不出问题
- 传一个很大的文件、把网断掉，这两下能试出大半毛病
- 留出半小时慢慢查，别在上线前一分钟才想起来

**示例命令：上线前搜一遍写死的密钥**

```
# 在项目文件夹里跑，看看有没有把密码、密钥直接写在代码里
grep -rn "sk-\|api_key\|password" ./src
```

## 能直接抄的模板

上线前把这段发给 AI，让它逐条查，查不了的必须说实话

> 你是我的上线检查员。我的项目是：（一句话说清做什么的、给谁用）。
> 请按下面五条逐条检查。每条只回答三件事：有没有问题、问题在哪（文件或页面）、我该怎么改。
> 1. 代码里有没有写死的密码、密钥、后台地址？
> 2. 数据库里有没有测试时留下的假数据？
> 3. 用户输错内容、网络断掉、上传超大文件时，页面会显示什么？
> 4. 没登录的人能不能直接打开后台页面？
> 5. 出事了怎么退回上一个版本，把步骤写清楚。
> 查不了的那条，直接写「这条我查不了，需要你手动确认」，不要猜。

## 常见坑

### × 只在自己电脑上试过

你的电脑有缓存、有登录状态、依赖也都装好了。换台机器、换个浏览器、用手机流量再走一遍，问题基本都在这一步冒出来，而且一冒就是一串。

### × 听 AI 说一句「没问题」就放心了

它看不到你的运行环境，也看不到真实数据，只能判断代码写得像不像对的。让它指出具体位置，你再自己去点一遍，别省这一步。

### × 测试数据和真数据混在一起

测试时留下的假订单、假用户，上线后可能被当成真的，还会发给真人。上线前清一遍，或者干脆用一个单独的测试库。

### × 没给自己留退路

上线前先确认：出事了怎么退回去。旧版本还在不在、数据能不能恢复、谁能一键关掉。没想好退路就先别上，晚一天不亏。

## 做完自查

- ☐ 我能说出最坏情况是什么，也知道怎么恢复
- ☐ 代码里没有明文密码和密钥，配置和代码分开放
- ☐ 我自己用手机、新账号、断网各走了一遍主流程
- ☐ 出了事我知道怎么退回上一个版本，而且亲手试过一次

# 54
# 上线之后怎么持续改进

上线不是终点。定三个数字、每周复盘一次、每次只改一处，你的东西会越用越顺手，人也留得住。

### 这篇讲什么

☐ 知道上线后该盯哪三个数字，不被一堆报表淹没

☐ 能从一堆抱怨里挑出这周真该改的那一条

☐ 会用「小步改 + 前后对比」确认改动到底有没有用

## 01 先定三个数字，别什么都看

上线后最容易犯的错是：什么都想看，最后什么都没看。先挑三个数字，多一个都别加。

一个看有没有人来（比如每天多少人打开）；一个看有没有人用成（比如走完主流程的比例）；一个看有没有人骂（比如报错次数、退款数）。

数字不用精确到小数点。每天同一时间记一笔，一周下来方向就出来了。表格不用漂亮，手机备忘录里记也行。

- 三个数字要能一天看一次，一次不超过五分钟
- 只记数字不记感受：你觉得「今天挺顺」不算数
- 数字变差先别急着改，先看这周有没有特殊情况，比如放了假、做了推广

## 02 从抱怨里挑一条，别一次改十处

用户说什么都改，等于什么都没改。把抱怨按出现次数排个序，只动最上面那一条，剩下的先记着。

一次只改一处，改完观察一周。同时改五处，出了效果你也说不清是哪处起的作用，白改了你都不知道。

改之前先把当前数字抄下来。没有前后对比，你永远分不清是改好了，还是这周本来就运气好。

- 同一个问题被三个人提到，就值得排进这周的活
- 只被一个人提到、又要大改的，先记下来放着
- 改动越小越好：能改文案就别动流程，能改流程就别动结构

## 03 每周留半小时，跟 AI 复盘一次

复盘不用长。把这周的数字、用户的原话、你改了什么，三样东西一起丢给 AI，让它帮你找规律。半小时够用。

关键是给真实材料。你只说「最近不太好」，它只能回你一堆正确的废话，看着像建议，其实一句都用不上。

让它给三个可能的原因，你自己判断哪个最像。别让它直接出方案，方案得你定，因为只有你知道手上有什么牌。

- 用户的抱怨原话照抄，别自己先总结一遍，总结会把关键细节磨掉
- 复盘结论落成一句话：下周只改这一处

**示例：每周一张五行小表，比什么报表都好用**

```
日期 | 打开人数 | 走完主流程 | 报错次数 | 这周改了什么
周一 | 120 | 43 | 7 | 改了注册页按钮文案
周二 | 96 | 38 | 9 | 没动
周三 | 141 | 61 | 4 | 没动
周四 | 133 | 58 | 3 | 没动
周五 | 150 | 66 | 3 | 没动
```

## 能直接抄的模板

每周复盘用的提示词，把材料填进去就行

> 这是我这一周的情况，请帮我复盘。
> 项目：（一句话说清做什么的、给谁用）
> 三个数字：打开人数___，走完主流程___，报错次数___（上周分别是___、___、___）。
> 这周我改了什么：（写具体，比如改了注册页按钮文案）
> 用户的原话（照抄，别总结）：
> 1.
> 2.
> 3.
> 请你做三件事：
> 1. 从数字变化里指出最反常的一处，并说明可能是哪一步出了问题。
> 2. 把用户原话归成最多三类问题，按被提到的次数排序。
> 3. 给出三个可能的原因，每个都写清楚「如果真是这样，下周该看哪个数字来验证」。
> 不要给一堆建议，只给下周最值得动的那一处。

## 常见坑

### × 上线就不管了

上线只是开始。头两周问题最密，建议每天花五分钟看数字、看报错。等用户已经走了才发现，代价比每天看五分钟大得多。

### × 一次改太多地方

改五处，效果好了你不知道好在哪，效果差了也不知道差在哪，下次只能靠猜。一次一处、一周一看，改动才攒得下来。

### × 只看平均数

平均每天 100 人，可能是周一 600 人、其余几天几乎没人。平均数会把最糟的那天藏起来，每天的数字要分开看。

### × 把用户的客气话当真

用户说「挺好的」不代表好用，可能只是不好意思开口。真正有用的是他卡在哪一步、点了什么没反应。看行为，别只听评价。

## 做完自查

- ☐ 我只盯三个数字，每天五分钟能看完，不贪多
- ☐ 每次只改一处，改之前把当前数字抄了下来
- ☐ 我能说出这周被提到最多的一个问题是什么
- ☐ 我留的是用户的原话，不是自己总结过的版本

第七部分

# 场景实战

第 55 — 66 讲

# 55
# 用 AI 写周报和办公文档

每周都要写的东西最耗时间，因为它不难但很烦。这篇教你把周报、会议纪要、汇报材料变成填空题：先把素材堆进去，再让它按你的格式拼装。

---

### 这篇讲什么

☐ 能在十分钟内写出一份主管看得下去的周报

☐ 会把零散的会议记录整理成能直接发出去的纪要

☐ 知道哪些活 AI 写得比你好，哪些必须自己补

## 01 周报难写，难在你不知道主管想知道什么

大部分人写周报是从「我这周干了什么」开始的，于是写成了一份流水账。看的人其实只想知道三件事：进展到哪了、有什么风险、下周干什么。

所以别一上来就让它「帮我写周报」。先把你这周干的事列成流水账，一条一行，不用整理，写全就行。这是素材。

然后告诉它读者是谁、关心什么，让它按进展、风险、下周计划三段重排。

- 流水账要写够。你给它七条它才好挑，只给三条它只能编
- 把「做完了什么」写成带结果的句子，「调研了方案」不如「定了用方案 B，下周开始接」
- 风险那一段一定要有。没风险的周报，读的人会以为你没在推进

## 02 会议纪要：先记该记的，再整理

开会时你只要记三样：谁说了什么结论、定了什么、谁负责做什么。中间的讨论过程不用记，那是过程不是结果。

会开完把这些零散的句子丢给它，让它整理成「结论、待办、责任人、时间点」四栏。

有录音也行。先用工具转成文字再丢进去。转写的错字不影响理解，不用逐字校对。

- 待办必须带人和时间。只写「跟进一下」，下次开会还是没人做
- 让它单独列一节「没定下来的事」。这些才是最该追问的
- 发出去之前自己过一遍：有没有把谁的话记成别人的

## 03 汇报材料：先定结论，再铺证据

给上级的材料，最忌讳铺垫三页才说结论。正确顺序是：开头一页说清结论和建议，后面才是依据。

让 AI 先帮你把结论压成一句话。这句话写不出来，说明你自己还没想清楚，先别写材料。

结论定了，再让它按「结论、依据、风险、需要什么支持」四段铺开。

- 每页只讲一件事。一页塞三个观点，一个都记不住
- 数字要标时间和出处。「上个月涨了三成」比「涨了很多」有用得多
- 让它用主管的原话复述一遍你的结论，你看走没走样

**周报提示词，直接复制**

```
下面是我这周干的事，流水账，没整理：
<把你的流水账粘在这里>

请按下面要求写一份周报：

读者：我的部门主管，他只关心进度、风险和下周安排，不看过程细节。
篇幅：三段，每段不超过四行。

第一段 进展：把这周做完的事合并成两到三条，每条写成「做了什么，结果是什么」。
第二段 风险：列出目前卡住或可能延期的事，每条写清卡在哪、需要谁配合。如果确实没有，就写「本周无风险」。
第三段 下周：列出下周要推进的三件事，每件一句。

要求：
- 不要写「积极推进」「高度重视」这类空话。
- 我流水账里没提到的事，不要编。
- 有你觉得该补充但信息不够的地方，单独列出来问我。
```

## 能直接抄的模板

会议纪要整理，直接复制

> 下面是这次会议的零散记录，格式乱，不用管：
> 〈把记录粘在这里〉
>
> 请整理成四部分：
>
> 1. 结论：会上定了什么，一条一句。
> 2. 待办：谁、做什么、什么时候之前完成。
> 3. 没定下来的：讨论了但没有结论的事，写清卡在哪个分歧上。
> 4. 需要我确认的：记录里意思不明确的地方，列出来问我。
>
> 要求：
> - 只整理我记下来的内容，不要补充你没看到的信息。
> - 人名按原样保留，不要替换成「某同事」。
> - 待办如果缺时间或责任人，标成「待明确」并在第 4 部分提醒我。

## 常见坑

### × 让它凭空写周报

它只能编。看起来像模像样，一条都对不上你真干的事，改起来比自己写还费时间。永远先喂素材。

### × 流水账只给三条

素材太少它就开始补细节，补出来的你得一条条核实。宁可多写几条，让它帮你挑。

### × 不写风险段

没有风险的周报，读的人会认为这事没难度或者你没在推。卡住了要说，需要人配合更要说。

### × 整理完直接发

它可能把两件事合并成一件，也可能把别人的结论安到你头上。发之前逐句过一遍，两分钟的事。

## 做完自查

☐ 我先把这周干的事列成流水账，再让它整理，没有让它凭空写

☐ 我告诉了它读者是谁、关心什么，而不是只说「帮我写周报」

☐ 风险段我写了实话，卡住的地方和需要谁配合都写清了

☐ 发出去之前我逐句看过，确认没有编造和安错人

# 56
# 用 AI 做数据分析：从表格里问出答案

AI 不擅长算数，但它擅长写算数的代码。这篇讲怎么把一张表格交给它，问到你要的答案，而且数字不会错。

### 这篇讲什么

☐ 会把表格结构讲清楚，让它一次写对分析代码

☐ 学会用「让它写代码你去跑」来避免算错

☐ 分得清哪些结论能用，哪些只是看起来像结论

## 01 先说清表长什么样

人看一眼表头就懂。AI 看不见你的表，它只知道你打字告诉它的东西。

所以第一步是把结构讲清楚：有几列、每列叫什么、是什么类型、有没有空格子。三五行字就够。

最省事的办法是直接把前几行数据粘给它。带上表头，去掉敏感内容。

- 数据类型要说：这列是日期、这列是金额、这列是分类。它猜错类型，后面全错
- 空格子要说。空的它可能当成 0，平均值一算就偏了
- 数据量大就只给结构和前二十行，别把整张表粘进去

## 02 别让它心算，让它写代码算

大模型算数是弱项。三个数的加减还行，几百行求平均它经常算错。

正确做法是让它写代码，Python 或者 SQL 都行，你拿去跑一遍，让机器算。AI 负责写计算步骤，机器负责算。

跑出来的结果再丢回去让它解释。这样数字是准的，解释是人话的。

- 让它给能算出结果的代码，而不是直接给数字。张口就来一个数，多半是编的

- 让它把中间步骤也输出：总共多少行、去掉多少空格子、每类各多少条。对不上就是有问题
- 同一份数据问两遍，两次数字不一样，说明它在猜

**示例提示词：让它写代码而不是心算**

```
我有一份表格，结构如下：
<粘贴列名和类型，或者前 3 行数据>

我想知道：<你的问题>

请这样回答：
1. 先写出能算出答案的 Python 代码（用 pandas），我要自己跑。
2. 代码里加上这几步检查，并把结果打印出来：总行数、去掉空值后行数、关键分类各有多少条。
3. 代码跑完后，再解释结果是什么意思。

注意：
- 不要直接告诉我答案数字，我没有你的运行结果，你算的我不信。
- 如果我的问题用这份数据回答不了，直接说回答不了，并说明还缺什么数据。
```

## 03 结论要带证据

「转化率下降了」这句话没用。有用的是「转化率从上月的 3.1% 降到 2.4%，主要掉在新用户那一段」。

让它每个结论后面都跟一个数、一个对比对象、一个时间范围。凑不齐这三样的结论，别往报告里放。

还有一件事：让它列出「这份数据回答不了的问题」。比它硬答有用得多。

- 相关不等于因果。「买 A 的人也常买 B」不代表推 A 就能卖 B
- 样本太少的细分结论别信。一百人里掉三个，那是噪声不是趋势
- 让它说出结论成立的边界：什么情况下这个结论会不成立

## 能直接抄的模板

结论审查，直接复制

> 下面是我根据数据得出的结论，请你逐条审：
> <把结论粘在这里>
>
> 对每条结论，按这四点检查并输出：
>
> 1. 有没有具体数字？没有就标「缺数字」。
> 2. 有没有对比对象？没有就标「缺对比」。
> 3. 有没有时间范围？没有就标「缺时间」。
> 4.
> 这个结论是「相关」还是「因果」？如果原文声称因果但数据只能支持相关，标出来。
>
> 最后列出：这份数据回答不了、但读者可能会问的问题。
>
> 要求：严格一点，宁可说结论不成立，也不要说「有一定道理」。

## 常见坑

× **整张表粘进去**

又慢又容易看漏，敏感数据还有泄露风险。给它列名、类型和前几行就够了。

× **直接问「平均多少」**

它在心算，几百行必错。换成让它写代码，你跑一遍，让机器算。

× **信它张口就来的数字**

它给出的任何数字，都要能从代码或原始数据里复现出来。复现不了就当它没说。

× **把相关当成因果**

两件事一起变不等于一个导致另一个。报告里要写清这是相关还是因果，别含糊过去。

## 做完自查

- ☐ 我把表格的列名、类型、空格子情况说清楚了
- ☐ 我要数字的时候让它写代码，自己跑过一遍再采用
- ☐ 每个结论后面都有数、有对比对象、有时间范围
- ☐ 我知道这份数据回答不了什么，也把这个写进了报告

# 57
# 把图片、语音、文件交给 AI

现在的模型能看图、听声音、读 PDF，但它们看的方式和人不一样。这篇讲怎么给、怎么问，以及它们最常看错什么。

### 这篇讲什么

☐ 知道哪些内容适合直接给图，哪些要先转成文字

☐ 会写「看图回答」的提示词，而不是只说「看看这个」

☐ 知道它在哪类图上最容易看错，提前避开

## 01 能看，但看得很粗

模型看图不是像人一样盯着细节看。它先把整张图压成一段大概的描述，再基于这段描述回答你。大块的东西它看得准，细小的它经常漏。

经验是：图里东西少、字大、对比清楚，它可靠。图里密密麻麻、手写、反光、倾斜，它就开始编。

所以给图之前先问自己一句：这张图里最关键的三个信息是什么？如果藏在小字里，先裁出来单独给。

- 一张图只问一件事。同时问五个问题，准确率会掉下来
- 让它先描述图里有什么，再回答你的问题。描述错了你立刻能发现
- 表格类的图，先转成表格数据再给，比让它读图准得多

## 02 语音先转文字

多数模型能直接听音频，但日常用法还是先转文字更可靠。转写现在很准，而且转成文字你自己能检查。

转完再让它总结、提取待办。语气词和重复的话不用管，让它自己略过。

只有在一种情况值得直接给音频：你想让它听语气，比如判断对方是不是不满。这种场景很少。

- 转写时加一句「不确定的词用问号标出来」，你就能挑出该核对的地方
- 长会议录音先切段，每段十分钟左右。整段太长它容易漏掉中间
- 说话人分不清的时候，先说明「这是三个人在聊」，不然它会把谁说的搞混

## 03 文件别整个扔，先看结构

PDF、Word、Excel 都能直接给。但页数多、表格密的时候，它读得又慢又不准，还可能只看了前半部分就回答。

先看结构：多少页、几章、有没有大量表格和图。再决定是整份给，还是只给相关章节。

扫描件（图片转出来的 PDF）要先确认能不能选中文字。选不中就说明它是图片，得先做文字识别。

- 关键要求写在前三行。文件一长，后面的要求它容易当没看见
- 让它先给目录或章节清单。能列对才说明它真的读到了
- 涉及合同、金额、日期的文件，重要条目自己再核一遍

**示例提示词：看图提问**

> 这张图我要问的是：<你的问题>
>
> 请按这个顺序回答：
> 1. 先用一句话描述这张图整体是什么。
> 2. 然后只回答我上面的问题。
> 3. 如果你在图上看到但不确定的内容，单独列出来，标上「不确定」。
>
> 注意：
> - 图上没有的信息，不要根据常识推测。
> - 如果图片太模糊或者关键信息看不清，直接说看不清，不要猜。

## 能直接抄的模板

长文档提问，直接复制

> 这份文件我先说清楚：
> - 大概〈页数〉页，共〈章数〉章
> - 我关心的是第〈章号〉到第〈章号〉章
> - 我的问题是：〈你的问题〉
>
> 请这样做：
>
> 1. 先列一遍你能看到的章节标题。列不出来的部分直接说没读到。
> 2. 只在我指定的章节范围内回答。范围外的内容不要混进来。
> 3.
> 每个回答后面标出这个信息在第几页或第几节。标不出来的，说明你没有实际定位到。
>
> 要求：文件里没有的信息，不要用常识补。

## 常见坑

### × 一张图塞满问题

问题越多它越糊，往往前两个答对，后面全是编的。一张图一问，问完再给下一张。

### × 用截图代替复制文字

能复制的文字就直接复制，又快又准。截图只在排版本身有意义时才用，比如看设计稿、看报错格式。

### × 让它读手写和密表

这两样是它最容易看错的。手写先自己打一遍，密表先导出成数据，比事后核对省时间。

### × 假设它读完了整份文件

长文件它可能只处理了开头。让它先列章节清单、回答里标页码，你才知道它到底看了哪。

## 做完自查

☐ 一次只给一张图或一个文件，问一个问题

☐ 我让它先描述再回答，而不是直接丢问题

☐ 关键信息藏在细节里的时候，我先裁出来或转成文字

☐ 我知道它在手写、密排表格、模糊图上不可靠，这些地方自己核

# 58
# 用 AI 做翻译：怎么翻得又准又像人话

机器翻译一眼就能看出来，问题通常不在词，在句子结构和语气。这篇讲怎么让它翻得像人写的，以及翻完怎么查。

### 这篇讲什么

- ☐ 会写让翻译更自然的提示词，而不是逐字转换
- ☐ 学会用术语表和读者要求把风格定下来
- ☐ 知道翻完先查哪几样，哪些错最致命

## 01 先说清给谁看

同一段中文，「给老板看的报告」和「给客户看的介绍」翻出来完全不同。你不说，它只能翻成最保险的书面语。

所以第一句就要交代：翻成什么语言、给谁看、什么场合用。三件事说清，风格就定了一半。

再加一句文体要求：口语还是正式、要不要保留原术语、称呼用不用敬语。

- 说明读者水平。给专家看和给普通人看，用词完全不一样
- 指定哪些地方不要直译：成语、口号、产品名、职位名
- 告诉它哪些术语有固定译法，给一张对照表最省事

## 02 让它先理解，再动手翻

中文长句是翻译翻车的主要来源。一句话套三四个意思，照直译过去就成了谁也读不懂的东西。

让它先用自己的话复述一遍这段中文的意思，你确认没理解偏，再让它翻。多一步，错得少一半。

或者直接要求：先拆成短句再翻，每句不超过二十个词。

- 遇到有歧义的地方，让它列出两种理解让你选，别自己闷头选一个

- 人名、公司名、产品名先说清是保留原文还是音译
- 数字和日期格式要指定。中文的年月日顺序和英文不一样，混着写会出岔子

**示例提示词：带上读者和文体要求**

> 把下面这段中文翻译成英文。
>
> 读者：正在评估我们产品的海外客户，懂业务但不懂中文。
> 场合：官网上的产品介绍，语气专业但不生硬。
>
> 翻译要求：
> 1. 先拆句再翻，每句不超过 20 个词。
> 2. 不要逐字直译。中文的短句和排比，英文读起来会像口号，改成正常陈述句。
> 3. 下面这些词按固定译法：〈术语对照表〉。
> 4. 中文里省略的主语要补上，不要让读者猜是谁。
>
> 原文：
> 〈粘贴中文原文〉

## 03 翻完怎么查

最有效的一招不是让它重翻，是让它只挑错。你让它重写，它会改得面目全非；你让它挑错，它能指出哪几句不像母语。

具体做法：把译文丢回去，问「哪些句子读起来像翻译腔，原文什么意思，怎么改」。让它给出修改建议，你来定要不要采纳。

最后自己过一遍四样东西：数字、否定词、条件、时间。翻译最容易在这四处出错，而且错了很致命。

- 让它标出「没有把握的句子」，那些地方自己重点核
- 中译英要检查主语。中文常省略主语，英文不补上读者会读不懂
- 否定和条件最容易翻反。not、unless、only if 这些地方逐句对照原文

## 能直接抄的模板

译文校对，直接复制

> 下面是一段中文原文和它的英文译文。请你只做校对，不要重写全文。
>
> 请输出三部分：
>
> 1. 读起来像翻译腔的句子：列出原句和你的改写建议，并说明为什么原句不自然。
> 2. 意思可能走样的地方：包括数字、否定、条件、时间、主语的遗漏或重复。逐条对照中文原文。
> 3. 你拿不准的句子：列出来，说明你不确定什么。
>
> 要求：
> - 不要为了改得漂亮而改动原意。
> - 如果译文没有大问题，直接说没有，不要硬凑修改。
>
> 中文原文：
> <粘贴原文>
>
> 英文译文：
> <粘贴译文>

## 常见坑

### × 只说「帮我翻译」

它只能给你最平淡的书面语，读起来一眼假。补上读者、场合、文体三句话，质量差一个档。

### × 让它直接重写译文

它会把你的译文改成另一篇，问题和风格全变了。要改成「只挑错，提建议」，决定权在你手上。

### × 不核数字和否定

「不低于」翻成「低于」、「三成」翻成「三倍」，这类错误最要命。这四样必须逐句对照原文。

### × 长句照直译

中文一句套三件事，直译成英文就是一串从句，读者读到一半就丢了。先拆句，再翻。

## 做完自查

☐ 我交代了翻译成什么语言、给谁看、什么场合用

☐ 我给了术语对照表，或者说明哪些词保留原文

☐ 我让它先复述理解再翻，或者要求先拆句

☐ 翻完我逐句核过数字、否定、条件、时间这四样

# 59
# 用 AI 读懂你的整个项目

把整个代码仓库丢给 AI 是浪费，它读不完也读不准。这篇讲怎么用「先看地图、再问细节」的方式，让它真的读懂你的项目。

### 这篇讲什么

☐ 会先让它画出项目结构，确认它看懂了大局

☐ 学会只喂相关的几个文件，而不是整个仓库

☐ 知道怎么让它老实承认「这部分我没看到」

## 01 先让它画一张地图

新接手一个项目，你也得先看目录结构。AI 一样，你不给结构，它就只能猜。

第一步给它三样东西：目录树、说明文档、配置文件。然后问一句「这个项目是干什么的、怎么跑起来的」。

它答得对，说明读懂了大概。答得含糊，要么你给的信息不够，要么它压根没读完。

- 目录树两层到三层就够。太深它反而抓不住重点
- 直接问「这个项目从哪个文件开始执行」。这个问题的答案最能暴露它有没有真看懂
- 让它指出「哪些目录是自动生成的、可以不用看」，你就知道它是不是真读了

## 02 只喂相关的几个文件

一个中等项目几十万行，塞不进去。就算塞得进，相关的不相关的混在一起，它也抓不住。

正确顺序是先定位：你要改的功能涉及哪几个文件。搜关键词，找到三五个文件就够了。

把这几份给它，加一句「这个功能只跟这几个文件有关」。范围一收窄，回答质量立刻不一样。

- 让它说出「要改这个功能你还需要看哪些文件」，它列出来的你自己去确认
- 调用链一起给：谁调用了这个函数、这个函数又调用了谁
- 给测试文件比给注释更有用。测试写明了这个功能本来该有什么行为

## 03 让它承认自己不知道

AI 有个坏习惯：没看到的部分它也会顺着编。你说「这个项目里有缓存层」，它就假设有，然后开始编缓存的代码。

所以每次都要加一句：只基于我给的文件回答，没看到的地方直接标出来。

它标出来的地方，就是你下次要补的文件。这个习惯能省掉大量「它改了个根本不存在的函数」的麻烦。

- 让它每个结论后面标「依据：哪个文件、哪一段」。标不出来的就是它猜的
- 直接问「你没看到哪些部分，会让你的回答不可靠」，它列得出来
- 涉及配置和部署的问题，一定要它看到实际配置文件再答，不能靠猜

**示例提示词：先让它读懂项目**

```
下面是我的项目信息，请先只做一件事：理解它。

目录结构：
<粘贴两层目录树>

说明文档和配置文件：
<粘贴 README 和主要配置>

请回答：
1. 这个项目是干什么的？用三句话。
2. 从哪个文件开始执行？请求进来之后的处理顺序是什么？
3. 哪些目录是自动生成的、不用看？
4. 上面这些信息里，你有哪些地方是猜的？直接标出来。

要求：只基于我给的内容回答。没看到的部分明确说「未提供」，不要用通用经验补。
```

## 能直接抄的模板

改功能前的定位提问，直接复制

```
我要改这个功能：〈描述功能〉

我提供这几个文件：
〈文件1、文件2、文件3〉

请按顺序回答：

1. 这个功能现在的实现流程是什么？按调用顺序说，每步标出文件和函数名。
2. 要改这个功能，还需要看哪些文件？列出来，并说明为什么。
3. 这次改动可能影响哪些地方？包括调用方、下游、测试。
4. 你没看到的部分里，哪些会让你的回答不可靠？

要求：
- 只用我提供的文件作为依据。
- 涉及你没看到的文件，说「未提供」，不要推测里面的实现。
```

## 常见坑

### × 整个仓库塞进去

塞不进，塞进去也抓不住重点。先自己定位到几个文件，再给它，效率高一个数量级。

### × 不给目录树直接问

它不知道文件在哪，只能根据文件名猜，猜错的概率很高。花一分钟贴个目录树，省半小时来回。

### × 相信它说的「我理解了」

它随时都会说理解。要用具体问题验：从哪开始执行、这个函数被谁调用。答不上来就是没读懂。

### × 让它改它没看过的文件

它会按想象中的实现来改，改出来的东西和你项目对不上。改哪个文件，就先把那个文件给它。

## 做完自查

- ☐ 我先给了目录树和配置文件，让它画出项目地图
- ☐ 我自己定位了相关文件，只喂这几个，没有整仓塞进去
- ☐ 我要求它标注依据来源，并说明哪些是猜的
- ☐ 改哪个文件之前，我确认它真的看过那个文件

# 60
# 用 AI 做代码迁移和升级

把老代码搬到新框架、把旧版本升到新版本，是 AI
最能帮上忙也最容易帮倒忙的活。这篇讲怎么搬得稳，出了事怎么退回。

### 这篇讲什么

☐ 会先建一张对照表再动手，而不是让它一把梭

☐ 学会先搬一个文件来校准风格

☐ 知道迁移最怕什么，怎么提前拦住

## 01 先建对照表

迁移本质上是把旧写法换成新写法。所以在写代码之前，先让它列一张对照表：旧的写法对应新的哪个、参数变了什么、哪些直接删掉了。

这张表你逐条看一遍。看懂了再动手，看不懂就先补课。它改出来的东西你没法验，等于白改。

对照表还有个用处：它是你验收的依据。改完对着表一条条勾。

- 让它把「没有直接对应的」单独列出来。这些才是最费时间的部分
- 升到哪个版本要说清，相邻版本和跨大版本差别很大
- 让它标出每个改动会影响哪些下游调用

## 02 先搬一个文件

别让它一次搬完。挑一个中等复杂、有测试覆盖的文件，让它只改这一个。

改完你跑测试、看代码风格、看它怎么处理那几条难对应的地方。满意了，再让它按同样方式搬剩下的。

这一步不是为了省时间，是为了校准。你摸清了它的改法，后面才敢放手。

- 让它把这次迁移的规则总结成几条，后面每次搬都贴上去，保证风格一致
- 每搬完一个文件就提交一次。出问题能精确退回，不用全盘重来

- 搬完一个就立刻跑测试，别攒到最后一起跑

## 03 最怕的是悄悄改了行为

代码能编译、测试能过，不代表行为没变。老代码里可能有奇怪但必要的边界处理，新写法默认帮你「修」掉了。

防这个的办法是等价测试：同一批输入，新旧两版各跑一遍，输出必须一模一样。

让它帮你写这个对比测试。涉及金额、日期、排序的地方尤其要测，一处不同就是事故。

- 让它列出「这次迁移可能改变行为的地方」，逐条确认是不是你想要的
- 默认值变化特别危险。旧版默认 A、新版默认 B，静默就错了
- 收尾前保留旧代码路径，用开关切换。出问题能一键切回去

**示例提示词：迁移前先要对照表**

> 我要把这段代码从〈旧版本/旧框架〉迁移到〈新版本/新框架〉。
>
> 请先不要写代码，先给我三样东西：
> 1. 对照表：旧写法 → 新写法，一行一条。参数有变化的标出来。
> 2. 没有直接对应的部分：单独列出，并说明通常怎么处理。
> 3. 可能改变行为的地方：逐条说明旧的行为是什么、新的会变成什么。
>
> 我确认之后你再改代码。改的时候：
> - 只改这一个文件，其他文件不要动。
> - 保留原逻辑，不要顺手「优化」。
> - 改动处加注释标出「迁移改动」。

## 能直接抄的模板

等价测试，直接复制

> 我要验证迁移前后的行为是否完全一致。
>
> 旧代码：
> <粘贴旧代码>
>
> 新代码：
> <粘贴新代码>
>
> 请写一个测试脚本，做这几件事：
>
> 1. 构造一批输入，覆盖：正常值、空值、边界值（0、负数、最大值）、格式异常的值、重复值。
> 2. 把同一批输入分别喂给旧实现和新实现。
> 3. 逐个对比输出，只要有任何一处不同就打印出来：输入是什么、旧输出是什么、新输出是什么。
> 4. 最后打印一行总结：共测多少条，有几条不一致。
>
> 要求：
> - 两个实现放在同一个脚本里，我要能直接运行。
> - 不要修改两个实现本身。
> - 涉及浮点数比较请用容差，不要直接比相等。

## 常见坑

× **让它一次搬完整个项目**

改动量太大，出了问题你根本找不到是哪一步坏的。按文件走，一个一个来。

× **不看对照表就动手**

你不知道它把什么换成了什么，改完也没法验收。对照表是第一道关，必须过。

× **只看测试通过就认为成功**

测试覆盖不到的边界最容易出事。补等价测试，用同一批输入比新旧输出。

× **不留退路**

老代码直接删掉，出问题只能往前修。保留旧路径加开关，切换成本接近零。

## 做完自查

☐ 我先拿到了对照表，并且逐条看懂了

☐ 我先只搬一个文件试水，确认风格和改法没问题才铺开

☐ 每搬完一个文件我提交了一次，测试也跑过了

☐ 我写了等价测试，确认新旧行为一致，关键边界都覆盖到了

# 61
# 提示词也要版本管理

提示词改一次，效果可能好一点也可能坏一点，但你说不清改了什么、从哪一版开始变差。把它当代码管起来，这些问题就没了。

### 这篇讲什么

☐ 会把提示词存成文件，写清版本和用途

☐ 学会用一批固定问题检验改动是不是真变好了

☐ 知道团队里怎么共享和更新提示词

## 01 提示词就是代码

你在对话框里反复调试出来的那段话，是你的资产。窗口一关就没了，下次还得从头来一遍。

把它存成文件。文件名写清用途，文件开头写清：做什么、给谁用、改过几版、每版改了什么。

改之前先复制一份存旧的。提示词改坏了不会报错，只会悄悄变差，所以必须有退路。

- 一版一个文件，或者用版本管理工具存。别在原文件上直接覆盖
- 每版开头记一行「这版解决什么问题」。半年后你会感谢自己
- 把提示词和它的输出样例存在一起。光看提示词看不出好坏

## 02 改完怎么知道变好了

准备一批固定问题，二十到五十条，覆盖常见情况和边界情况。每次改完提示词，把这批问题全跑一遍。

然后对比新旧两版的输出。重点看坏例：原来答对的现在答错了没。

这批问题就是你的回归测试。它不保证新版本更好，但能保证你没把已经对的弄坏。

- 问题库里要放你真实遇到过的难例，不要只放容易的
- 每次改动至少看十条输出，只看一两条容易看走眼
- 记下每次改动的效果：哪类问题变好、哪类变差

**示例：一个提示词文件长这样**

```
# 提示词：客服邮件回复
# 用途：把用户投诉邮件转成可发送的回复草稿
# 版本：v3（2026-10-20）
# 本版改动：加了「不确定就说不确定」，减少编造补偿方案
# 上一版：v2 语气太软，用户以为我们认错了全部责任
# 验证问题库：prompts/tests/service-mail-questions.jsonl

你是客服。下面这封用户投诉邮件，请写一封回复草稿。

要求：
1. 先说明我们确认了什么事实，再说明我们会怎么处理。
2. 处理方案里不确定的部分，写「需要核实后回复」，不要直接给出承诺。
3. 不承认我们没有确认的责任，但也不要推卸。
4. 全文不超过 200 字。
5. 不要用「非常抱歉给您带来不便」这类套话开头。

用户邮件：
<粘贴邮件>
```

## 03 团队里怎么共享

一个人调好的提示词，别人直接用往往不好使，因为缺了背景。共享的时候要把背景一起给：这个提示词假设能读到什么、输出会被谁用。

建一个目录，按场景分类：写代码、写文档、做分析、客服回复。新人进来直接从里面挑。

谁改的谁说明改了什么。没有说明的改动，下次出问题没人查得出来。

- 提示词库里写清「什么场景用它」，比放提示词本身更重要
- 定期清掉没人用的。留着会误导新人以为这是标准做法
- 涉及公司敏感信息的提示词单独放，限制访问

## 能直接抄的模板

提示词改写请求，直接复制

```
这是我现在的提示词：
<粘贴旧提示词>

请帮我改一版，目标是：<具体问题，比如「减少编造」「输出更短」「更稳定地按格式输出」>

改完请给我三样东西：

1. 新版提示词全文。
2. 逐条说明你改了什么、为什么这么改。
3. 列出这版可能带来的副作用：会不会让某类回答变差？

要求：
- 保留原版的所有约束条件，不要为了改一处而删掉别的规则。
- 如果我的目标之间互相冲突，直接指出来，不要硬凑。
```

## 常见坑

× **只在对话框里调**

关掉就没了。你花两小时调出来的东西，下次得重来一遍。存文件只要十秒。

× **在原文件上直接改**

改完发现不如从前，已经找不回原来那版了。改前复制一份，成本接近零。

× **没有固定问题集**

每次改完只是随手试两三个问题，越改越偏还不知道。二十条固定问题就能兜住。

× **共享只给提示词**

别人不知道它假设了什么、输出要干什么用，套上去效果很差。背景和提示词一样重要。

## 做完自查

☐ 我的提示词都存成了文件，开头写清了用途和版本

☐ 每次改之前我复制了一份旧的，能退回去

☐ 我有一批固定问题，改完会跑一遍对比坏例

☐ 共享出去的提示词带了使用场景和背景说明

# 62
# A/B 测试：怎么知道改版真的变好了

你觉得新版更好，用户未必这么想。这篇讲怎么用最简单的办法测出真实效果，而不是靠感觉拍板。

### 这篇讲什么

☐ 会设计一个能得出结论的对比测试

☐ 知道样本要多少、看什么指标

☐ 分得清「真的变好了」和「这次运气好」

## 01 先定一个指标

「感觉更好了」不是指标。指标是一个能算出来的数：用户采纳率、平均修改次数、完成率、投诉数。

一次只盯一个主指标。同时看五个数，你永远得不出结论。

主指标要跟你要解决的问题有关。用户点得更多但问题没解决，那不叫变好。

- 指标要在开始之前定。不能等看到数据再挑一个好看的
- 除了主指标，再留两三个「护栏指标」，看有没有把别的地方搞坏
- 定性反馈也要收。用户说哪里别扭，比数字更能提示下一步怎么改

## 02 把人分成两组

最简单的做法：同一批用户随机分成两组，一组用旧版一组用新版，跑一两周。

样本太少得不出结论。几十个人的差异大概率是噪声，至少要有几百次使用。

没条件做真随机就按时间交替：单日旧版、双日新版。这不如随机严谨，但比拍脑袋强得多。

- 两组的人要像。一组全是老用户、一组全是新用户，比出来的差异不说明问题

- 测试期间别同时改别的东西，不然分不清是谁的功劳
- 跑够时间之前不要看结果，看早了容易被中间波动带偏

## 03 怎么读结果

新版好一点，不代表真的好。要追问一句：这个差距会不会只是运气？

经验判断：两组都只有几十次使用，差个百分之几，基本可以当没差别。差一倍以上，那多半是真的。

更该看的是坏例：新版在哪类情况下明显更差。整体变好但某一类变差，往往是最值得修的地方。

- 让 AI 帮你算显著性（这个差距纯属偶然出现的概率），别自己目测
- 分场景看，不要只看总体。总体变好可能掩盖了某个场景的崩坏
- 结论里写清边界：在什么用户、什么场景下，新版本更好

**示例提示词：让它算显著性**

```
我做了一次 A/B 测试，数据如下：

旧版：使用 <N1> 次，达标 <A1> 次
新版：使用 <N2> 次，达标 <A2> 次

请做三件事：
1. 算出两组的比率，以及绝对差距。
2. 算出这个差距纯属偶然的概率（做显著性检验），说明用的是什么方法。
3.
给出结论：这个差距是真差异还是噪声？如果样本不够，告诉我至少还需要多少数据。

要求：
- 不要直接下「新版更好」的结论，先给判断依据。
- 如果我给的数据量不足以下结论，直接说不足。
```

## 能直接抄的模板

测试方案设计，直接复制

> 我想验证这个改动有没有效果：<描述改动>
>
> 背景：<用户规模、使用频次、能改的范围>
>
> 请帮我设计一个能得出可信结论的测试方案，包含：
>
> 1. 主指标：用什么数衡量效果？怎么算出来？
> 2. 护栏指标：哪几个数能说明我没把别的地方搞坏？
> 3. 分组方式：怎么分？我现在能做的是 <随机分组 / 按时间交替 / 其他>。
> 4. 样本量和时长：我至少需要多少数据、跑多久才能下结论？给出估算过程。
> 5. 可能的干扰因素：测试期间还有哪些事会影响结果？怎么避免？
>
> 要求：按我的实际条件给方案，不要给一个我做不到的完美方案。

## 常见坑

× **没定指标就开始测**

测完再看数据挑一个好看的数，等于没有结论。指标必须在开始前写好。

× **样本太少就下结论**

几十次使用里的百分之几差异基本是噪声。让它估一下需要多少数据，按那个量跑。

× **同时改了好几个地方**

效果变好了你也不知道是哪个改动起的作用。一次只改一个变量，这是底线。

× **只看总体不看分场景**

总体提升可能掩盖了某个重要场景的崩坏。分层看，坏例比好例更有信息量。

## 做完自查

- ☐ 测试开始前我就定好了主指标和护栏指标
- ☐ 我算过需要多少样本，没有拿几十次使用就下结论
- ☐ 测试期间我只改了这一处，没有同时动别的
- ☐ 我分场景看过结果，也检查了坏例

# 63
# 长任务：让 AI 跑几个小时也不跑偏

让 AI 连着干几个小时，最容易出的问题不是能力不够，是跑着跑着忘了目标、越改越远。这篇讲怎么把长任务切成能收拢的形状。

### 这篇讲什么

☐ 会把长任务切成有产物的阶段

☐ 学会设检查点和中断后接着跑的办法

☐ 知道怎么给长任务上成本上限

## 01 长任务为什么容易跑偏

任务一长，中间会累积大量内容。它越往后越容易丢掉开头定的目标，抓着手边最近的细节不放。

所以长任务的关键不是让它更聪明，是把目标钉在前面，把进度记在外面。

记在外面就是：每完成一步写一个状态文件。文件里只记四样：目标、已完成、下一步、已知问题。断了能接着跑。

- 目标写在最前面，每开一段新对话就重新贴一次
- 每阶段结束让它总结「现在到哪了、离目标还差什么」，你核对
- 不相关的讨论及时清掉，别混在同一条线里越滚越大

## 02 切成有产物的阶段

好的切法，每一段都有一个能检查的产物：一个能跑的文件、一份清单、一张表。检查不了的分段没有意义。

每段开始前先让它说清「这段要做完什么、做完是什么样」。说不清就是切错了，重切。

产物出来你验一下，通过了再开下一段。这比最后一起验省力得多。

- 每段控制在一两个小时的量。太长中间出问题你发现不了

- 让它每段结束时列出「做完了什么、没做什么、有什么遗留」
- 遗留问题单独记。别让它顺手在里面加新需求

## 03 设卡点和上限

长任务要有几个必须停下来问你一声的点。比如涉及删数据、改配置、花钱的操作之前。

成本也要设上限。跑几个小时的花费可能远超你预期，提前说清「到多少就停」。

还要设止损条件：连续几次尝试都失败，就停下来商量，别在同一个坑里反复试。

- 明确告诉它哪些操作必须先问过你，不能自己执行
- 让它每次汇报带上已消耗的量，你才知道花了多少
- 连续三次没进展就换思路，或者自己接手

**示例提示词：给长任务立规矩**

```
这是一个长任务，我们先定规矩再开始。

目标：〈一句话说清做完是什么样〉
验收标准：〈怎么判断做完了〉

请分阶段做，每阶段遵守：
1. 开始前先说这个阶段要做什么、做完产出什么，等我确认。
2. 阶段结束时给我：做完了什么、没做什么、遗留问题。
3. 写一份状态文件，内容只有四项：目标、已完成、下一步、已知问题。

硬性要求：
- 涉及删除数据、修改配置、花钱的操作，必须先问我，不许自己执行。
- 同一个问题连续三次没解决，停下来告诉我，不要继续试。
- 每阶段开始前把上面的目标重贴一遍。
```

## 能直接抄的模板

状态文件模板，直接复制

```
# 任务状态

## 目标
<一句话，做完是什么样>

## 验收标准
<怎么判断做完了，越具体越好>

## 已完成
- <做完了什么，带产物位置>

## 下一步
- <紧接着要做的一件事，只写一件>

## 已知问题
- <卡在哪、试过什么、为什么没成>

## 不能动的地方
- <明确不许碰的文件、配置、数据>

## 消耗
<已用多少轮、多少费用>

---
新一段对话开始时，先把这份文件整个贴进去，再说「继续」。
```

## 常见坑

× **一口气说个巨大的目标**

它会在中途丢目标，你也不知道跑到哪了。切成有产物的阶段，每段验一次。

× **不写状态文件**

对话一长它就忘，一断线就得从头解释。状态文件是长任务的存档点，五分鐘就能写好。

× **不设停机条件**

它会在同一个错误上反复试十几次，烧时间也烧钱。连续三次没进展就该叫停。

× **让它在同一条对话里跑一整天**

越往后质量越差。一个阶段一收，重开一段对话，贴状态文件接着跑，反而更稳。

## 做完自查

☐ 任务切成了几个阶段，每个阶段有能检查的产物

☐ 我写了状态文件，随时能中断和恢复

☐ 我设了必须停下来问我的操作清单

☐ 我设了止损条件和成本上限

# 64
# AI 的失败模式清单：什么时候别信它

它不是随机出错，它有固定的几种出错方式。认识这几种，你就能在它胡说的时候第一时间发现。

### 这篇讲什么

☐ 能认出六种最常见的失败模式

☐ 每种都有一套当场验证的办法

☐ 知道什么情况下根本不该用它

## 01 六种最常见的翻车

第一种，编来源。它给你一个看起来很像真的书名、链接、法条编号，实际上不存在。要引用的场合，这一种最致命。

第二种，算错数。三个数还行，多了就错。凡是要精确结果，都让它写代码算，你跑一遍。

第三种，知识过期。它的知识停在训练那天。新版本、新政策、新价格，它可能给你去年甚至更早的答案。

- 第四种，顺着你说。你先表达了倾向，它就跟着附和，把反对意见说得越来越轻
- 第五种，假装读过。你给的文件太长，它没读完却按读完的口气回答
- 第六种，过度自信。它说「一定」「肯定」的时候，往往正是它最没把握的时候

## 02 怎么当场验

引用类的东西，一律要能点开。点不开就是编的。这条没有例外。

数字类的东西，让它写代码或者列算式，你自己核一遍。它不肯给过程，就当它没算。

时效类的东西，直接问它「这条信息的时间截止到什么时候」。它说不清，你就当是旧的。

- 想判断它是不是在顺着你说：换个说法问一遍相反的观点，看它会不会立刻倒戈
- 想判断它有没有读完：让它说出文件里的具体细节，比如第几节、某个具体数字
- 它说得很绝对的时候，追问一句「什么情况下这个结论会不成立」

**示例提示词：让它自查刚才的回答**

> 请你对自己刚才的回答做一次自查，逐条回答：
>
> 1. 你这句话里有引用吗？如果有，逐条列出出处，并说明你有多确定它真实存在。
> 2. 有数字吗？数字是怎么得出的？请给出算式或代码。
> 3. 这里面有多少信息是来自我的输入，多少来自你的训练知识？分不清的标出来。
> 4. 你有没有因为我的问法而倾向于附和？如果我的前提是错的，现在指出来。
> 5. 你对哪一句最没把握？那句话应该怎么核实？
>
> 要求：诚实一点。不确定就说「不确定」，不要为了显得可靠而给确定答案。

## 03 什么时候干脆别用

涉及安全、法律、医疗的最终判断，不能只靠它。它可以帮你整理信息、列出选项，但决定必须人做。

要担责任的事：签合同、报税、发公告。它写草稿可以，你逐字看完才能用。

它看不到的东西：你公司的内部情况、你团队里的人际关系、昨天刚发生的事。这些它不知道，问了也白问。

- 涉及别人的隐私和公司机密，不要往里贴
- 涉及钱的计算，一律自己算或者用代码算
- 它给你的任何结论，你都要能说出「我是怎么验证的」。说不出就是没验证

## 能直接抄的模板

高风险回答的复核清单，直接复制

> 这段回答我准备用在正式场合，请你帮我按下面的清单复核，逐条给出结论：
>
> <粘贴你要用的内容>
>
> 1.
> 事实类：每一条事实性陈述，标注「可核实」或「需核实」。可核实的说明去哪里核。
> 2.
> 引用类：所有引用的出处，标出真实存在的把握程度。把握低于高，就要我另外去查。
> 3. 数字类：所有数字的来源和计算方式。算出来的要能复现。
> 4. 时效类：哪几条信息可能已经过期？大概是什么时候的？
> 5. 缺口：要判断这件事，还缺哪些信息？
>
> 最后用一句话回答：这段内容能不能直接用在正式场合？如果不能，最需要补的是什么？

## 常见坑

× **相信它给的引用**

编造出处是它最稳定的一个毛病。任何引用都要能点开或者查到实物，查不到就当它没说过。

× **用它算钱**

它在多位数运算上不可靠。金额一律自己算，或者让它写代码你去跑。

× **把它的自信当准确**

语气跟准确率没有关系。说得越绝对，越要追问一句「什么情况下不成立」。

× **拿它做要担责的决定**

它能整理信息但不能替你承担责任。合同、税务、公告这类，草稿可以它写，结论必须人定。

## 做完自查

☐ 我要引用的每一处，都实际查到了出处

☐ 所有数字要么我自己算过，要么用代码跑过

☐ 我对时效性内容确认过截止时间

☐ 定稿前我让它自查了一遍，自己也复核了一遍

# 65
# 在团队里推广 AI：怎么带不写代码的同事上手

一个人用得再好，团队不跟上也没用。这篇讲怎么推、先推什么最见效，以及哪些做法会让人反感。

---

### 这篇讲什么

☐ 会挑一个能马上省时间的场景做示范

☐ 知道怎么建一个别人愿意用的共享库

☐ 避开推广里最容易踩的几个坑

## 01 别从培训开始

一上来搞两小时培训，讲一堆原理，大部分人听完还是不会用。这是最常见的失败。

有效的路子反过来：先找一个具体的人、一件具体的事，帮他用 AI 省下两小时，然后让他在群里说一句。

一个人真实的省时间故事，比十页幻灯片有用。人会信同事，不信讲师。

- 挑那些每周重复、又不需要判断力的活：整表格、写固定格式的文档、改错别字
- 第一次一定陪着他做完，让他自己看到结果，而不是你演示给他看
- 让受益的人自己讲。你去讲，别人会以为你在推销

## 02 建一个别人愿意用的库

库里最该放的不是提示词本身，是场景：「我要写周报」对应哪段话，直接复制能出结果。

每个条目写清三样：什么场景用、复制到哪里、输出大概长什么样。三样缺一个，别人就不敢用。

条目宁少宁精。放五十条没人看的，不如放十条真能用的。

- 用他们的原话写场景描述，不要用你的专业术语
- 定期问「哪条你用了、哪条没用」，把没用的删掉
- 允许别人改，但改完要留一句说明改了什么

## 03 定边界，别搞成恐惧

必须说清什么不能往上传：客户名单、身份证号、未公开的财务数据、合同原文。这是硬线。

但别把边界讲成恐吓。讲清「这几样不能传」和「其余随便用」，比笼统说「注意安全」有用得多。

也别定「必须用AI」这种指标。逼出来的使用量全是形式主义，反而让人抵触。

- 边界写成三条以内，贴在最显眼的地方。太多了记不住
- 给出安全替代做法：不能传原文，就传处理过的版本
- 允许失败。有人用出了错，讲出来比藏着有价值

**示例提示词：设计一次小范围试推行**

> 帮我设计一次小范围的试推行，不要理论，要能落地。
>
> 背景：我所在的团队〈人数〉人，其中会写代码的〈人数〉人。
> 团队每周最耗时间的重复活是：〈举两三件〉
>
> 请给我：
> 1. 最该先试的那件事是哪个？为什么选它，判断依据是什么。
> 2. 找一个愿意试的人，我该怎么开口？给我一段可以直接说的话。
> 3. 第一次陪他做的时候，具体分几步？每步做什么。
> 4. 如果成功了，怎么让成果被更多人看见，而不显得像我在推销。
> 5. 如果失败了，怎么收场才不打击其他人。
>
> 要求：所有建议都要在我这个人数规模下做得到，不要给需要专门立项的方案。

## 能直接抄的模板

共享条目格式，直接复制

```
## 场景
<用同事的原话说：我每周要做什么事>

## 怎么做
复制下面这段，把尖括号里的内容换成你的：

<提示词全文>

## 输出大概长什么样
<贴一个脱敏后的真实输出片段，让人知道能拿到什么>

## 注意
- <这个条目什么时候不适用>
- <哪些内容不能粘进去>

## 谁在用
<名字，或者写「已验证可用」>

## 最近改动
<日期 谁 改了什么>
```

## 常见坑

× **先开大会讲原理**

听的时候觉得有道理，散会后一个都不做。先帮一个人省下时间，效果传播得比培训快。

× **建个大而全的库**

条目太多没人翻，最后变成没人维护的文档。十条真能用的，好过五十条摆设。

× **只讲风险不讲用法**

讲完大家只记住「不能用」，于是干脆不用。先给能用的场景，再划三条硬线。

× **定使用率指标**

逼出来的使用量全是应付，还会让人反感。要看的是有没有真省下时间，不是用了多少次。

## 做完自查

☐ 我先找了一件具体的活做示范，而不是先组织培训

☐ 共享库里每个条目都有场景、用法和输出样例

☐ 我划清了不能上传的几类信息，并给了替代做法

☐ 我没有定使用率这类形式指标，只看实际省下的时间

# 66
# 一年之后：这套东西会怎么变

这本书里的具体工具会过时，但底下那几件事不会。这篇说清哪些会变、哪些不变，以及你该怎么跟着走。

### 这篇讲什么

☐ 知道哪些能力大概率会变强、变得便宜

☐ 知道哪些做法不随工具变化

☐ 有一套自己的跟进方式，不用追着每条新消息跑

## 01 会变的部分

能看懂的东西会更多：从只能读文字，到能看图、听声音、看视频、直接操作电脑。

价格会继续降，速度会继续快。今天算起来贵得不敢用的场景，一年后可能随便用。

模型会越来越擅长自己动手：自己调工具、自己检查结果、自己重试。那套「一步一步指挥」的方法会变得没那么必要。

- 具体版本号、具体价格、具体参数都会过时，别去背
- 别把流程绑死在某个产品上。接口换来换去是常态
- 每半年重新试一次当初放弃的场景，可能现在能做了

## 02 不会变的部分

能把需求说清楚的人，永远占优势。工具越强，这个优势越明显，因为工具是放大器，放大的是你自己的清晰度。

小步验证、留退路、事后核对，这三件事跟模型多聪明无关。事情做错了，代价永远由你来担。

知道什么不该交给 AI，这种判断力只会越来越值钱。

- 说清目标、给够背景、定好验收标准，这三样不会过时
- 关键结论自己核一遍，这条不会因为模型变强而取消
- 留一份自己的判断，别把决策权整体交出去

## 03 怎么跟

别追每一条新消息。这个领域每天都有新东西，追不过来，而且大部分跟你手上的活无关。

有效的节奏是每季度做一次：拿你现在最费时间的一件事，看看有没有新的做法能省掉它。有问题就查，没问题就继续用你手上的。

遇到明显的能力跃迁，比如它突然能可靠地做某类以前做不成的活，再花时间系统学一遍。

- 一年重读一遍基础。基础没变，但你的理解会变
- 关注你所在行业的具体进展，别只看通用的热闹
- 手上留两三个不同厂商的备选，别把全部押在一个上

**示例提示词：季度复查**

> 帮我做一次「这套流程还成不成立」的复查。
>
> 我现在的工作方式是：
> <描述你现在用 AI 做的两三件事，以及用什么方式做>
>
> 请帮我判断：
> 1. 这里面哪些步骤现在可能已经有更省事的做法？说明理由，不要只说「可以试试新工具」。
> 2. 哪一步是我自己在做但本来能交出去的？
> 3. 哪一步看起来能省，但其实不该省？为什么？
> 4. 如果我只有一小时来验证，最该先试哪一件？
>
> 要求：不要推荐具体产品。我关心的是做法，不是工具。

## 能直接抄的模板

季度自查四项，直接复制到备忘录

> 每季度花二十分钟回答这四个问题：
>
> 1. 我这季度重复做得最多的三件事是什么？
> 2. 这三件里，有哪件现在可能能省掉一半时间？
> 3. 我上季度放弃过什么，现在能不能做？
> 4. 我有没有哪件事是「一直在用某个做法，但没想过为什么」？
>
> 答完只做一件事：挑上面最省时间的那一条，这季度把它改掉。不要一次改四个。

## 常见坑

× **追每个新模型**

每天都有新消息，追不过来，而且大部分跟你手上的活无关。按季度看一次就够。

× **背具体参数**

版本号、上下文长度、价格，这些几个月就变。背下来只会让你显得过时。

× **把流程绑死在一个产品**

接口变了、价格涨了、服务停了，你就得重做一遍。留两三个备选，切换成本会低很多。

× **以为基础不用复习**

工具在变，说清需求、小步验证、留退路这些没变。但每过一年，你对它们的理解会深一层。

## 做完自查

- ☐ 我知道这本书里哪些内容会过时，不会去死记那些
- ☐ 我记住了三件不随工具变化的事：说清目标、小步验证、留退路
- ☐ 我有固定的复查节奏，不是靠刷消息跟进
- ☐ 我手上不只有一个厂商的选项

Part Two

# English Translation

Introduction • 66 Chapters • Conclusion

# Introduction: Why This Book Exists

This book is about one thing: how to get AI to actually finish the work for you.

It is not about how powerful AI is, and it is not about who it will replace. You have heard enough of that. This book cares about a much more concrete question — you have a task in front of you today; how do you get AI to complete it, and complete it correctly?

## Who it is for

It is for people who already use AI but do not use it well.

Maybe you have tried. You asked it to write some code and got a pile of things that would not run. You asked it to improve a piece of copy and the result was worse than what you started with. You asked it to summarise a document and it missed every important point. And you concluded that this thing is overrated.

The problem is usually not the AI. It is how you talk to it. It works the same way as asking a colleague for help: if you say "just sort it out for me", they have to guess. If you say what you want, what you do not want, and what counts as done, they get it right the first time.

## Why it is written in plain language

Because most AI tutorials are not written as if a person were going to read them.

They like words such as "prompt engineering", "context window" and "retrieval-augmented generation", and then they assume you already know

what those words mean. You finish the article and still have no idea what to do at work tomorrow.

In this book, every technical term is explained in one sentence the first time it appears. Any term that is not explained is a term that does not need explaining. If you read a paragraph twice and still do not understand it, that is my failure, not yours.

## How the book is arranged

Sixty-six chapters in seven parts, from the most basic mental model all the way through hands-on scenarios to how to keep up over time.

Each chapter stands alone as a single question, so you do not have to read front to back. The structure is fixed: three opening lines telling you what you will be able to do afterwards, then three sections of body text, then a prompt or a piece of code you can copy and use directly, then the four mistakes people make most often, and finally four things to check before you call it done.

That copy-and-use template in the middle is the most valuable part of every chapter. Even if you skip all the body text and only take the templates, this book has paid for itself.

## Three ways to read it

If you are new to AI, read chapters 1 through 8. Those eight are about saying what you mean, and everything else is built on top of them.

If you already write code, jump to chapter 9. Keep the first eight as a reference you come back to.

If you are building a product, start at chapter 15, read through to the launch checklist in chapter 53, then continue into the hands-on scenarios.

If you just want to solve the problem in front of you, look up the closest chapter in the contents and go straight to the copy-and-use template.

## What this book does not do

It does not teach you how to train a model. It does not explain the internals of a transformer. It does not compare benchmark scores. There are more specialised books for all of that.

It also does not promise you a salary or a product. That part is yours.

It does one thing: it breaks down "how to work with AI" into sixty-six concrete moves you can use today.

The rest is you, doing the work.

Part One

# Foundations and Asking

Chapters 01 — 08

# 01
# What a Large Language Model Actually Is

It's not a search engine, and it's not a database. It's a machine that guesses the next word. Get that one idea, and you can work out every other technique in this book on your own.

---

## What this chapter covers

☐ You'll know why it talks so well, and why it can also lie with a completely straight face

☐ You'll be able to tell the difference between 'it actually knows this' and 'it made this up'

☐ You'll know what work you can safely hand over, and what work you never should

## 01 It really only does one thing: guess the next word

You give it a piece of text. It works out the most likely next word, writes that down, then guesses the one after it. Word by word, out comes a complete answer.

That's it. It isn't thinking, and it isn't understanding you. It learned, from an enormous amount of text, which words usually follow which other words.

That's why it writes so smoothly. Fluent text is the most common kind of text there is, and fluent sentences are all it ever saw.

- It's a probability machine, not a knowledge base. What you get is 'the sentence that most looks like an answer', not the answer
- By default it has no memory. Close the conversation and it forgets everything; next time you have to explain the background again

- It doesn't know when it's wrong. Getting it right and getting it wrong feel exactly the same to it

## 02 So why does it make things up so confidently

Because what it's guessing is what the next word looks like, not whether the thing is true.

A book title that doesn't exist, an invented paper number — they still read smoothly, so it can still write them, and write them with total confidence.

This has a name: hallucination. It isn't trying to fool you. It simply has no dial in its head marked true or false.

- The more specific something is, the more likely it is to be wrong: names, dates, numbers, links, the numbers of legal statutes
- The more common the knowledge, the more reliable it is: writing code, editing copy, explaining a concept — it's steady on those
- The newer something is, the less it knows: its knowledge stops on the day it was trained, and anything after that you have to tell it

## 03 What to hand over, and what never to hand over

Safe to hand over: rewriting, summarizing, translating, coming up with names, writing boilerplate code, explaining concepts, turning a pile of messy ideas into a structure. If it gets these wrong, you'll spot it at a glance, and a quick fix is all it takes.

Don't hand over: looking up the latest facts, doing precise arithmetic, making the final decision for you, anything touching private data or keys. If it gets these wrong you may not notice, and by the time you do it's too late.

The test is really one sentence: if this goes wrong, does somebody get hurt? If yes, don't let it do the job alone.

## A template you can copy

Add this at the end of your prompt and make it own up to what's shaky

```
After you answer, go back over it yourself:
1. Are there any specific names, dates, numbers or links in it? If so, list them separately and say how sure you are of each one.
2. Is there any part you're actually unsure about but wrote as if you were certain? Point it out directly.
3. Where you're unsure, write 'I'm not sure about this part'. Don't invent something that looks reasonable and paper over it.
```

## Common mistakes

### × Thinking it's looking things up for you

It isn't online (unless you've specifically given it a web tool). Its answers come entirely from what it memorized during training, which may be old and may be wrong. If you need facts, look them up yourself, or hook it up to search.

### × Letting one conversation run for dozens of turns

Late in the conversation it 'forgets' what was said earlier. Restate the important information every few turns, or just start a new conversation and lay the background out again.

### × Asking it to do arithmetic

It guesses words; it isn't a calculator. Multiplication and division past three digits goes wrong often. If you need a number, have it write a bit of code and run that, or work it out with a tool yourself.

### × Pasting keys and customer data straight in

What you paste in may be stored, and may be used for training. If you really need to, swap the sensitive parts for fakes first (a real name becomes 'Zhang San', a key becomes 'XXX'), then swap them back once you have the result.

## Checklist

☐ I know it's guessing words rather than looking things up, so I verify factual content myself

☐ I don't paste in keys, customer privacy or unreleased plans

☐ For important work, I have it flag which parts it's unsure about

☐ I don't expect it to do precise arithmetic, and I don't expect it to know what happened yesterday

# 02
# How to Say It Clearly: Getting Started with Prompts

The same request, worded differently, gets very different results. This chapter shows you how to state what you want in three sentences, so the AI hands you something you can use on the first try.

## What this chapter covers

☐ You'll know the three things any decent prompt has to spell out

☐ You'll know how to lock in tone and format with one role plus one sample

☐ You'll be able to say what you don't want up front and save yourself two or three rounds of rework

## 01 Start with three things: what you're doing, who reads it, what format

A lot of people type one line: 'write me a weekly report'. All it can do is guess, and you'll probably have to rewrite whatever it guesses.

Add the three things and the result changes immediately. What: turn this week's progress on three projects into a weekly report. Who reads it: the department head, who only cares about risks and next week's plan. What format: three paragraphs, no more than four lines each.

You'll notice that being able to state those three things means you had already thought them through. Writing a prompt is really forcing yourself to break the work down.

- What: start with a verb, one job per sentence. Don't stuff 'write the weekly report' and 'plan next week' into the same sentence

- Who reads it: your reader decides how deep the vocabulary goes. Writing for the boss and writing for a new hire are completely different jobs
- What format: how many paragraphs, how many words, whether you want a table or subheadings — just say it

## 02 Give it a role, then give it a sample

Telling it who to play works better than ten adjectives.

For example, 'you're an HR person who's been recruiting for ten years', and the job ad it writes will drift toward that kind of wording.

The stronger move is a sample. Paste in a piece of writing you think is good and say 'write in this tone'. A sample is far more precise than an adjective: tell it 'be more professional' and it has no idea how professional you mean; give it a paragraph and it gets it in one read.

- Make the role specific in years and setting. 'A middle-school Chinese teacher with eight years' experience' works better than 'an education expert'
- One paragraph of sample is enough. Paste in too much and it loses the thread
- If the sample contains things you don't want (it loves exclamation marks, say), mention it in passing: 'don't copy that part'

## 03 Once you've said what you want, say what you don't want

It has a common habit: ask for three hundred words and it writes eight hundred; tell it no jargon and it still crams a pile in.

Because 'don't' isn't there unless you say it. Blocking the usual problems up front saves you half the rework.

The usual lines: no exclamation marks, no industry jargon, don't make up data. Where it's unsure, have it flag that, and don't write a summary paragraph.

- List three to five prohibitions at a time. List too many and it can't keep track
- Give word counts as numbers. 'Around three hundred words' beats 'a bit shorter'
- If it adds something you don't want, don't delete it yourself. Just say which part you don't want and have it rewrite

**Example prompt**

```
Please write this week's work report for me.

What: turn this week's progress on three projects into a weekly report.
Who reads it: the department head, who only cares about risks and next week's plan.
What format: three paragraphs, no more than four lines each, no summary paragraph.

Your role: a veteran of eight years in project management who talks straight.
Write in this tone: "Project A slipped two days this week because the test environment went down. We'll make it up on Monday."

Don't do these: no exclamation marks, no industry jargon, don't make up numbers.
```

## A template you can copy

Save this empty shell and just swap the content each time

```
What: (one sentence, starting with a verb)
Who reads it: (who they are, what they care about most)
What format: (how many paragraphs, how many words, list or not)

Your role: (specific in years and setting)
Write in this tone: (paste a short piece of writing you like)

Don't do these: (three to five items, e.g. no exclamation marks, don't make up data)
```

## Common mistakes

### × Just saying 'make it better'

'Better' is empty. All it can do is write the most common version, which is the most mediocre one. Swap 'better' for concrete actions: open with a number, no paragraph longer than three lines, close with one piece of advice.

### × Asking five things at once

It'll do all five half-heartedly, and it may drop one or two entirely. Hand over one job at a time, then the next. If you really need them together, number them and have it answer one by one.

### × Being roundabout and polite, as if it were a person

'Could you possibly take a look at this proposal and see if there's anything that could be improved' — it can't find the point. Say instead: 'pick the three points the boss is most likely to reject, and give one fix for each.'

### × Fixing it yourself when you're not happy

Fixing it yourself means you never used it. Say what's wrong directly: 'the second paragraph is too vague, swap in concrete numbers'. It'll rewrite the whole paragraph, which is faster than picking at the words yourself.

## Checklist

☐ My prompt states what I'm doing, who reads it, and what format I want

☐ I gave it a specific role, or a sample it can copy

☐ I listed three to five 'don'ts', and I gave an explicit word count

☐ When its first attempt was bad, I changed how I asked instead of editing it myself

# 03
# Turn Your Requirements into a Checklist: Structured Prompts

Same model: for some people it works like an assistant, for others it's a lottery. The difference is whether you wrote your requirements as a checklist. The clearer the checklist, the less it drifts.

---

## What this chapter covers

☐ You'll be able to break a vague request into five blocks and write it in a fixed format

☐ You'll know which blocks you have to fill in and which you can skip

☐ You'll get it to answer item by item against your numbers, so you can check them off one at a time

## 01 Why a checklist keeps it from drifting

When you write one big block of text, it reads with a bias. Things at the end stick more easily; things buried mid-sentence get skipped. The same requirements written as a paragraph and written as three lines don't get carried out with the same accuracy.

A checklist pulls each item out on its own line, so it can't drop one. Checking is easier too: go item by item and you see at a glance what's missing.

There's nothing advanced about this. It's like handing work to an intern: say it out loud once and they won't remember all of it; write it as one, two, three, four and they won't go wrong. A checklist has another benefit: it's harder for you to miss something too.

- One item, one thing. Put two requirements in one sentence and it often does only the first
- Number them 1. 2. 3. Don't chain them together with commas
- Order them by importance, most important first

## 02 Five building blocks are usually enough

Background: it doesn't know your situation, so you have to tell it. One sentence on who you are and who the work is for.

Then task, requirements, output format, boundaries. Task in one sentence, starting with a verb; three to six requirements, as specific as you can make them; format spelled out as what it should look like; boundaries spelled out as what must not happen. Leave a blank line between blocks, and it reads more clearly.

You don't need all five every time. For simple work, task plus format is enough. For complex work, write all five. If you're not sure, start with task and format, and add more if the result isn't good enough.

- One sentence of background is enough: 'we're a six-person design studio, mostly local restaurant clients'
- Requirements have to be checkable: 'use colloquial wording' can't be checked; 'no technical terms appear' can be checked
- For the format block, it's best to give an empty shell and have it fill in the blanks

## 03 Make it answer against the checklist

One last move: have it number its answers and go through your requirements one at a time, saying how it met each one. The longer the output, the more this saves you.

That way you don't have to read the whole thing hunting for something. Glance at the numbers and you know what was done and what wasn't.

If something wasn't done, just say 'item 4 wasn't done, rewrite it'. That works far better than a vague 'have another go'.

- You can have it restate the requirements first, to confirm it understood them, before it starts writing
- Split long jobs in two: have it give an outline first, and write the full text only once you nod
- Save the same checklist and reuse it next time with different content

**Example prompt**

```
[Background]
We're a six-person design studio, mostly local restaurant clients.

[Task]
Write a short service introduction for new clients to read.

[Requirements]
1. Under three hundred words total.
2. No technical terms. Talk like a person.
3. Mention three specific things: turnaround time, number of revisions, how you charge.
4. End with a line that makes people want to get in touch.

[Output format]
Three paragraphs, each with a small heading in front of it.

[Boundaries]
No exclamation marks, don't promise anything you can't deliver, don't make up prices.

[How to answer]
When you're done, go through items 1-4 above in order and explain how you met each one.
```

## A template you can copy

The empty shell of the five blocks: copy it and fill it in

```
[Background]
(One sentence: who you are, who the work is for)

[Task]
(One sentence, starting with a verb)

[Requirements]
1. (A specific, checkable requirement)
2. (A specific, checkable requirement)
3. (A specific, checkable requirement)

[Output format]
(How many paragraphs, how many words, subheadings or a table or not)

[Boundaries]
(What must not happen, three to five items)

[How to answer]
When you're done, go through 1, 2, 3 in order and explain how you met each one.
```

## Common mistakes

### × Writing the checklist as an essay

Write a long explanation for every item and it loses the point. One item per line, no more than twenty words a line, and put anything that needs explaining in brackets.

### × Requirements that can't be checked

With a requirement like 'make it compelling', all it can do is guess. Swap in 'open with one concrete scene' or 'no paragraph longer than three lines', and you can see at a glance whether it worked.

### × Handing over a checklist of ten-plus items

With too many items it will definitely drop a few, and you won't know which ones. Past six items, split it into two rounds: do the body first, then refine the details.

### × Giving a checklist but no example

> The checklist says 'friendly tone' and it still doesn't know how friendly you mean. Attach a short piece of writing you're happy with and say 'write like this'. That does more than three extra requirements would.

## Checklist

☐ Every item in this checklist of mine can be checked, not just a pile of adjectives

☐ I broke the task into one thing per item, with no sentence carrying two requirements

☐ I gave an output format, ideally an empty shell it can fill in directly

☐ I had it answer by number, so I can check each item off

# 04
# Making the AI Remember What You Already Told It

It has no memory. Close the window and everything is gone. This chapter teaches two moves: how to repeat yourself less in a long conversation, and how to save the things you always have to explain into one file you carry around.

---

## What this chapter covers

☐ You know why it forgets, and you have a feel for the pattern of how it forgets

☐ You can write a background summary that picks the work right back up when you paste it into a new conversation

☐ You can turn long-term information into a file you paste in whenever you need it

## 01 It Didn't Forget — It Never Stored It

A lot of people assume it is quietly keeping notes. It isn't. Every time it answers, it can only see what is inside this conversation window. Close the window and all of it is gone.

The window has a ceiling. Talk long enough and the early part gets pushed out. The further back something is, the sooner it goes. How much fits depends on the model, but there is always a limit.

So its "forgetting" usually shows up after you have been chatting for a long while. The setup you gave at the start stops counting later on.

- One topic, one conversation. Don't go from polishing a resume to fixing code in the same window

- Repeat an important setting a few turns after you first said it, or put it at the top of each question
- When it starts answering the wrong question, the window is nearly full and it is time to open a new one

## 02 Before You Open a New Conversation, Write a Handover Note

Switching windows doesn't mean starting over. Before you close one, you can have it write a summary itself. It is good at this, and it takes a couple of minutes.

That summary is your ticket in next time. Open a new window, paste the summary first, then add the new request, and it picks right up.

Keep the summary short. Say who you are, what you are working on, what is settled, what is still missing. Past one screen it stops working, because it still skips things when it reads. Better to leave out a line or two than to write a running log.

- Cap the length when you ask for it: "Under two hundred words. Conclusions and settled decisions only"
- Save the summary in a notes file. Don't count on scrolling back up through the chat history
- If you do the same kind of work over and over, keep adding to the same summary. It gets more accurate the more you use it

## 03 Turn What You Always Have to Explain Into One File You Carry Around

Some things you have to say every single time: your role, your readers, the way your company words things. You shouldn't be retyping those. Retype them often enough and you will stop bothering.

Write them into a plain text file and keep it on your desktop. Every time you open a new conversation, paste the contents in first.

Ten seconds to paste, and it saves far more than explaining from scratch each time. Copywriting, fixing emails, drafting reports — the same file works for all of them. The longer you use it, the more it is worth.

- Keep only stable information in the file. Don't stuff one-off requests into it
- Before adding a line, ask yourself: will this still be true in three months? If not, leave it out
- Replace sensitive details (real names, client names, keys) with codes before writing them in

**Example: a carry-around background file**

```
[Who I am]
I do content in the marketing department, mostly WeChat posts and product copy.

[My readers]
Office workers aged 25 to 35. Impatient. They decide in the first three lines whether to keep reading.

[How we word things]
Talk like a person. No industry jargon. Address the reader as "you."

[What I don't like]
Long warm-ups at the start, too many exclamation marks, a forced uplift at the end.

[Standing requirements]
No paragraph longer than three lines. Keep the whole piece under 800 words.
```

## A template you can copy

Before you switch windows, have it write a handover note

> This conversation is almost over, and I am going to open a new window to keep going.
> Write a handover note under two hundred words that covers:
> 1. Who I am and what I am working on.
> 2. Conclusions already settled (only what is settled, not the process).
> 3. Problems still open.
> 4. Information I need to add before we start next time.
> Use short sentences, write it as a list, no pleasantries.

## Common mistakes

### × Expecting it to remember what you said last week

Across conversations it remembers nothing. The setup you gave last week has to be pasted again this week. Treat it like a new colleague who needs the background explained again every morning.

### × Chatting in one window all day long

The longer it runs, the muddier it gets, and the tail end is all noise. Finish one thing, open a new window, reconnect with a handover note. That beats grinding along in one long window. One window, one job — that is the cheapest way to work.

### × Writing the handover note as a long essay

A summary that is too long gets skipped anyway when it reads. Stay under two hundred words, keep only conclusions and settled decisions, and cut the whole discussion.

### × Putting real client names into the background file

That file gets pasted into conversations again and again, which means uploaded again and again. Use codes like "Client A" and "Project B," and keep the mapping in your own head or on a sheet only you have.

## Checklist

☐ I know it only has memory inside the current conversation, and nothing carries over to a new window on its own

☐ I restate the important setup at the top of my question

☐ Before switching windows, I have it write a handover note under two hundred words

☐ I have a carry-around background file, and every sensitive detail in it is replaced with a code

05

# Give the AI a Role and It Comes Out Much Better

Same request, plus one line about who it is, and the output is completely different. This chapter covers how to write a role, how much detail to give, and why most roles people write do nothing.

## What this chapter covers

☐ You know how a role shapes the way it talks, and you stop treating it as a polite formality

☐ You can write a specific role that actually constrains the output, instead of one line of "you are an expert"

☐ You know a role cannot fix factual accuracy, and you know what you still have to check yourself

## 01 A Role Is Not a Courtesy — It Switches Which Voice It Uses

Giving it a role is telling it whose voice to speak in. Its word choice, its examples, its sentence length all lean that way.

Ask the same question — "how is this copy?" — and you get a pile of polite filler. Say "you are the client who nitpicks everything" and it starts going down the list, problem by problem.

The reason is plain. It has seen countless ways of speaking across an enormous amount of text. The role you give it just helps it pick one of those to use.

- A role decides who is talking, not whether the facts are right. A wrong fact stays wrong

- The more specific, the better it works: name the industry, how long they have done it, who they are facing right now
- One role covers one kind of job. Don't use the same role for a weekly report and an apology letter

## 02 A Useful Role Needs Three Things

Don't just write "you are an expert." There are too many kinds of expert. It doesn't know which direction to push, so in the end it is the same as saying nothing.

A useful role has three parts: who this person is in the field, how many years they have done it, and who they are helping with what right now. Put all three together and it knows which words to reach for.

Add one line about what you don't want and the effect is even clearer. "No jargon," "no pleasantries" — it tightens up fast.

- Role, audience, goal — that fits in one sentence. Don't spread it over a paragraph
- "You are a teacher who grades essays for elementary school kids" is far more useful than "you are an education expert"
- If you are worried it will guess, add "if you don't have enough information, ask me three questions first"

**Example prompt**

```
You are a property management customer service supervisor with eight years on the job. Right now you are helping me write a water shutoff notice to residents.
Most of the residents are office workers, and they only glance at a notice.
Requirements:
1. Lead with the shutoff time and who is affected. Don't circle around it.
2. Plain language. No stock phrases like "we kindly ask for your understanding."
3. Under 150 words, in three lines.
4. If key information is missing, ask me first. Don't make up a time.
```

## 03 The More Titles You Stack On, the Worse the Work Gets

Some people like to pile on a long string: world-class, twenty years of experience, has served Fortune 500 clients, has won countless awards. This almost never helps.

The more you stack, the more it starts performing. The page fills with pretty adjectives and the substance that actually does the job shrinks. You want the work done well, not a beautiful self-introduction.

Two or three sentences is enough. Spend the rest of the space on the task itself: what you want, who reads it, how long it should be, what format. Those four matter more than the role.

- The role is seasoning, the task is the meal. If the meal isn't explained, no amount of seasoning saves it
- Once a role works well for you, save it as a template and reuse it next time
- The role and the style request have to point the same way. Don't ask for rigorous and funny at the same time

## A template you can copy

Fill in the brackets and you have a role prompt you can use as it is

```
You are a [job title] with [number of years] in [industry]. Right now you are helping [who] get [what] done.

How to write:
1. Plain language, no industry jargon. If you truly need a term, explain it in parentheses right after.
2. Give the conclusion first, then the reasons. No more than three reasons.
3. No pleasantries. No closing lines like "hope this helps."
4. If you don't have enough information, ask me three questions first. Don't guess.

The task this time is: [write your request here]
```

## Common mistakes

### × Writing only "you are an expert"

The word "expert" barely constrains it at all. It still answers the default way. Change it to "you write menus for a restaurant chain, and your readers are office workers ordering delivery" and the output shifts immediately.

### × The role and the task pull against each other

Say "you are a meticulous lawyer" and then ask it to "keep it lively, throw in some jokes" and it can't hold both. It ends up looking like neither. The role and the style request have to point the same way.

### × Assuming a role means it won't get things wrong

A role changes how it talks, not how accurate it is. It will still invent statistics, invent legal clauses, invent case numbers. Anything that has to be right, you check yourself.

### × Repeating the role before every message

Once per conversation is enough. It remembers. Repeating it just eats space and squeezes out the actual request. Add it again when you start a new conversation.

## Checklist

☐ My role spells out the person in the field, how long they have done it, and who they are helping with what

☐ I added a line on what I don't want, such as no jargon and no pleasantries

☐ The role stays at two or three sentences, and the rest of the space goes to the specific task

☐ I know a role doesn't improve factual accuracy, so I checked the key numbers and quotes myself

# 06
# A Few Examples Beat a Long Explanation

Saying "make it livelier" ten times does less than handing over three old pieces you are happy with. This chapter covers where to find examples, how many to give, where to put them, and why examples sometimes make the result worse.

---

## What this chapter covers

☐ You can pin down the style you want with two or three samples instead of leaning on adjectives

☐ You know how to pick examples and how many to give, so it doesn't pick up a pile of bad habits

☐ You can get it to learn the style without copying the specific details out of your samples

## 01 Adjectives Don't Land — Examples Do

Phrases like "more professional" or "more down to earth" mean something different to everyone. It can only guess at your yardstick, and guessing wrong is normal.

Hand it three samples you already have and the yardstick becomes clear at once. It follows the sentence length, the word habits, the paragraph structure, and hits the mark far more often.

This approach has a name: few-shot prompting — you give it a few samples first and let it follow their format. You don't need to remember the name. Just remember "give examples."

- One decent example beats five sentences describing the style

- For anything format-driven — weekly reports, tidying a table, fixing an email — giving examples is about the most reliable method there is
- A verbal description only sets a direction. An example sets the standard

## 02 One Rule for Picking Examples: You Have to Be Happy With It

Don't pad the list with things that are "close enough" or "still usable." The examples are the standard. Lower the standard and the output drops with it.

Two or three is the sweet spot. Give only one and it may not catch the pattern. Give more than five and it takes up space and throws it off course.

Three examples should ideally cover different cases. For product titles, say: one short, one with the price in it, one leading with the selling point. Then it understands what is allowed to vary.

- Keep the examples in one format. Don't mix a table with a long paragraph
- Label them "Example 1," "Example 2," and end with "now handle the following in this style"
- Counter-examples work too, but you have to say why: "don't write this one like that, because…"

**Example: three samples plus one new task**

> Below are three WeChat Moments posts I am happy with. First look at what they have in common:
>
> Example 1: Friday, three in the afternoon. The air conditioning in the office finally stopped rattling.
> Example 2: The meeting moved to tomorrow. I got an extra hour of sleep. Worth it.
> Example 3: The bakery downstairs has a sale. I bought three and called it dinner.
>
> What they share: all short, all have a specific time or number, no slogans, no pitch at the end.
>
> Now, in this style, write three posts about "the new coffee shop downstairs from the office." Learn the style only. Don't use any of the names or places that appear above.

## 03 Bad Examples Are Worse Than No Examples

The most common problem is examples that contradict each other. One serious, one jokey — whichever it learns, the result looks half-finished. You end up with something that is neither.

The second problem is examples that are too long. Paste three 2,000-word articles and it can't find the point. It also tends to lift whole paragraphs, and editing that is more work than rewriting.

The fix is unglamorous: point at the one thing you actually want. Add "note that every sentence in these three is under twenty characters" and it has something to aim at.

- After the examples, have it restate the pattern. Confirm it understood, then let it work
- Tell it explicitly not to carry over the names, places and numbers that appear in the samples
- If you can't find good examples, first have it pull three patterns out of your old copy for you

## A template you can copy

When you have no samples handy, use this to have it work out the pattern first

```
I am giving you 3 samples below that I am happy with. Don't rush to write anything new yet.

Step one: list what these three have in common — sentence length, tone, structure, whether they use numbers and specific times.
Step two: after I confirm, write 3 new pieces following those common points.
Step three: the new pieces must not contain any of the names, places or numbers that appear in the samples.

The samples:
[Paste the three samples here]

The new task: [Write it here, and say who reads it and how long it should be]
```

## Common mistakes

### × Padding the list with "okay" examples

The examples are the standard. Lower the standard and the output drops, and you will end up thinking the AI is no good. Better to give two you are really happy with than to stuff in five mediocre ones.

### × Examples in inconsistent formats

One table, one long paragraph — it can't learn a pattern, it just blends the two styles together. Get them into one format before you hand them over.

### × Giving examples without stating the task

It may just keep writing, or hand the samples back with two words changed. After the examples, always add "now handle the following in this style."

### × Samples so long they get copied whole

Long samples are the easiest thing for it to lift. Cut them down to three or five sentences, or write it plainly: "learn the style and structure

only, all the content has to be rewritten."

## Checklist

☐ The samples I give are ones I am genuinely happy with, not whatever was within reach

☐ The examples share one format, there are two or three of them, and they are labeled "Example 1," "Example 2"

☐ I said clearly: learn the style only, don't copy the specific details inside

☐ I stated exactly what the new task is instead of letting it guess what to do

# 07
# Let It Think One Step at a Time: Chain-of-Thought

When a job involves calculating, comparing, or weighing several conditions at once, asking straight for the answer is a good way to get it wrong. Make it write the intermediate steps out, and the accuracy looks noticeably different. You can also see exactly where it went wrong.

## What this chapter covers

☐ Know which jobs are worth making it write out steps for, and which are just a waste of time

☐ Can pin the steps down yourself, instead of tossing in a vague 'think carefully'

☐ Learn to spot-check the middle steps, so a smooth-sounding conclusion doesn't fool you

## 01 It Thinks While It Writes, and If It Doesn't Write, It Hasn't Thought

It thinks as it writes. The words that come out are its thinking process. Whatever it doesn't write down is, for practical purposes, not thought.

Handing a complex problem to it in one go is really asking it to guess. Let it list the conditions first, then work forward step by step, and every step can be written while it looks at the one before.

This approach is called chain-of-thought (that is, having it write out the intermediate reasoning steps one by one, instead of just throwing a conclusion at you).

- Money math, comparing options, decisions with several conditions — these are the jobs most worth making it show its work
- Once the process is written out, you can check it. If it's wrong, you know which step went wrong, and you don't have to start the whole thing over
- Don't use it for simple jobs. It just drags things out, and the answer gets wordier

## 02 You Set the Steps, and That Holds Up Better Than Letting It Freestyle

The laziest fix is to add one line: 'Reason step by step first, then give the conclusion separately at the end.' Adding it already beats not adding it.

A steadier way is to pin the steps down. For example: 'Step one, list the known conditions. Step two, say which formula to use. Step three, plug in and calculate. Step four, check the units.'

Then have it put the conclusion alone on the last line. That makes your review fast, and you don't have to hunt for the answer inside several hundred words of reasoning.

- 'Step by step' works better than 'think carefully' — with the latter, it often just agrees and carries on as before
- Ask it to write out the intermediate result of every step. Don't accept 'and then we get the result'
- Four to six steps is the sweet spot. Beyond that it starts padding

**Example prompt**

```
Don't give the answer yet. Follow these four steps:
Step 1: List the conditions given in the problem one by one. Include both the numbers and the units.
Step 2: Say which formula or rule to use, and why that one.
Step 3: Plug in and calculate. Write out the intermediate result of every step.
Step 4: On its own line, write 'Conclusion:' and put only the final answer there.

If there isn't enough information, stop at step 1 and ask me. Don't make assumptions.

Problem: [paste the problem here]
```

## 03 Reading Smoothly Doesn't Mean It Computed Correctly

For jobs like writing copy, fixing typos or translating, making it write out reasoning only makes things slower and wordier. What one step can do doesn't need to be spread out.

One more thing to watch: steps that look reasonable are not necessarily right. If it invents a number in some middle step, everything after that follows the error, and it still reads smoothly.

So spot-check long reasoning. Pick two or three middle steps and verify them yourself. Look at numbers, units and dates first — those three are where things go wrong most often.

- Check the steps for hard errors first, then look at the conclusion. Do it the other way round and the conclusion drags you along with it
- Give it the same problem twice. If the two conclusions differ, it isn't stable on that problem
- For jobs with lots of numbers, just have it write a bit of code and run it. That's more reliable than reasoning

## A template you can copy

A general template for complex problems: split first, compute next, check at the end

```
Answer in this order. Don't skip steps:
1. Restate: say in one sentence what I'm asking.
2. Conditions: list the known information. Mark anything missing as 'missing'.
3. Approach: say how you plan to solve it, and why that way.
4. Compute: calculate step by step. Write the intermediate result for each step.
5. Check: go back over the numbers and units, and point out where an error is most likely.
6. Conclusion: on its own line, only the final answer.

If there isn't enough information, stop at step 2 and ask me. Don't make assumptions.

My question: [write it here]
```

## Common mistakes

### × Adding just one line: 'think carefully'

That instruction is too weak, and it often still gives the answer straight away. Change it to 'write it out in steps 1, 2, 3, then give the conclusion separately at the end', and it holds up.

### × The more detailed the steps, the better

Too many steps and it starts padding, inventing useless stages to fill the space. Four to six steps is enough. The extras are usually filler, and they can bury the step that actually matters.

### × Seeing smooth steps and believing them

A reasoning process can read smoothly and still be made up. Verify the numbers, units and dates yourself. Don't just look at the final answer.

### × Making it reason at length over simple jobs

Fixing typos, adjusting formatting, rephrasing — writing out reasoning on these just wastes time. If one sentence does the job, just say it. Don't add a process.

## Checklist

☐ When the job needed calculating, comparing or several conditions, I made it write the intermediate steps

☐ I pinned the steps down to a few concrete ones, instead of just saying 'think carefully'

☐ I had it put the conclusion alone on the last line, so it's easy to check

☐ I spot-checked the numbers and units in the middle, and didn't just copy the conclusion

# 08
# Make It Output the Format You Want

Same task, but whether you have it output a table, a list or JSON makes a big difference. This piece shows you how to spell out 'what it should look like', and skip ninety percent of the rework.

## What this chapter covers

☐ Can write instructions that get the format right the first time

☐ Can use two lines of example instead of a pile of adjectives to describe a format

☐ Know how to run batch jobs in chunks, and how to spot quickly when it dropped a few rows

## 01 Why It Keeps Handing You a Wall of Filler

If you don't specify a format, it writes in the most common shape. And what does the most common answer look like? A line of pleasantries at the start, a few paragraphs of explanation in the middle, and a summary paragraph at the end.

This isn't it being disobedient. It has no choice. All it can do is guess 'what is the most likely next word', and if you don't give it a frame, it takes the smoothest path.

So don't write things like 'be concise' or 'be professional'. There's no way for it to know how concise you mean. Say how many lines, how many fields, what order.

- The more your request looks like a 'shape', the easier it is to follow: how many lines, how many columns, what the fields are called, header or no header
- Words like 'concise', 'professional' and 'clear' can't be quantified, so they say nothing

- It's built to add an opening and a closing summary. Either say explicitly that you don't want them, or hard-code 'output only the table, no other text'

## 02 The Fastest Way: Show It an Example

Describing a format is hard. Demonstrating one is easy. Give it two lines of sample, and it's very good at copying what it sees.

It's best to put the edge cases into the sample too: what to fill in when there's no data, how to write 'not sure', how to truncate when it's too long. Once it sees it, it'll do the same.

Use the field names you'll actually use later. Don't write 'name' — write 'customer name'. Don't write 'amount' — write 'amount (CNY)'.

- Two lines of sample is enough. One line doesn't show the pattern, and three or more and it starts copying across the wrong rows
- Have it write fixed text for anything uncertain, 'unknown' for instance. That's better than leaving it blank — blanks are hard to filter out later

**Example: spell out the format with two lines of sample**

```
Output in the format below. Output only the table, no opening or closing text:

No. | Customer name | Request | Priority | Amount (CNY)
1 | Zhang San | Quotation | High | 12000
2 | Li Si | Website redesign | Unknown | Unknown

Rule: write 'Unknown' for anything the source doesn't mention. Don't speculate.
```

## 03 Run a Small Batch at a Time and Check as You Go

Don't dump two hundred rows on it right away. Give it five to test-run, see whether the format is right, then scale up.

Once the count goes up it gets lazy: it merges a few rows in the middle, drops a field, or skips the last few rows entirely. This is a normal thing to see. It's not your bad luck.

The safe approach is twenty to thirty rows at a time, then glance at the row count yourself. If the count is off, it dropped some.

- Have it number its output. Afterwards, check whether the numbering has gaps — much faster than reading row by row
- If it's going into spreadsheet software, have it produce CSV (plain text with columns separated by commas). Don't have it draw a table with vertical bars — those paste across the wrong columns

## A template you can copy

Use this when tidying up information in bulk. Replace the bracketed parts with your own content.

```
Turn the content below into a table. Output only the table, no explanation.

Fields (fixed order): No. | Customer name | Request | Priority
(high/medium/low) | Amount (CNY)

Rules:
1. Keep the numbering exactly as I gave it. Don't renumber.
2. Write 'Unknown' for anything the source doesn't mention. Don't speculate.
3. Amounts: digits only. No 'CNY' and no commas.
4. How to judge priority: result needed within three days = high; within a
month = medium; everything else = low.

Format example:
1 | Zhang San | Quotation | High | 12000
2 | Li Si | Website redesign | Unknown | Unknown

Here is the content:
[paste your source text]
```

## Common mistakes

### × Only saying 'make the format neat'

It doesn't know whether neat means four columns or six rows. State the field names, the order, the count. If you can't, show it two lines of sample — that works ten times better than adjectives.

### × Handing it several hundred rows at once

Past a certain count it slacks off: merging in the middle, skipping at the end. Do twenty to thirty at a time, count the rows when it finishes, and re-run that stretch if the count is off.

### × Asking for a pretty table layout

Tables with vertical bars often paste into Excel in the wrong columns, and then you're splitting them by hand. If it's going into a spreadsheet, ask for CSV, comma-separated.

### × Not saying what to write when there's no data

If you don't say, it makes something up and fills it in, and it looks quite plausible. Define a fixed marker like 'Unknown' up front, so you can filter out what still needs filling in.

## Checklist

☐ My instructions state the field names, the order and the count, and don't use words like 'concise' or 'professional'

☐ I gave at least two lines of sample, including what to fill in when there's no data

☐ For batch work I ran it in chunks, and counted the output rows to check

☐ For anything going into spreadsheet software, I asked for CSV instead of a drawn table

Part Two

# Writing Code

Chapters 09 — 14

# 09
# The Right Way to Have AI Write Code

It's very capable at writing code, but if you don't spell out the environment, you get back code that won't run. This piece is about stating what you need in a form it can work from.

---

## What this chapter covers

☐ Can turn a need into a spec that AI can write from directly

☐ Know how to paste an error back so it gets the fix right in one round

☐ Can ask for the smallest working version first, then add features one step at a time

## 01 State the Environment First, Don't Make It Guess

The same requirement may be written completely differently in Python 3 and in an old version. If you don't say, it can only go with the most common one.

So put the environment in your very first sentence: which language, which version, which libraries are installed, where it runs. For example, 'Python 3.10, standard library only, running on Windows'.

Also spell out input and output. Where does the data come from — reading a file, or typing it in? Does the result print to the screen, or get written to a file? It can't guess these reliably.

- Use as few libraries as you can. Specifying 'standard library only' can save you more than half the environment setup time
- Say how it will run: from the command line, by double-clicking, or in a web page. Those three are written very differently
- If someone else will use this code later, say up front how you want it to be called. That changes how it's written

## 02 Have It Do One Thing at a Time

The mistake beginners make most often: asking it for a complete system in one shot. Hundreds of lines of code, and when one spot is wrong you simply can't find where.

The right order is to get the smallest step working first. For example, 'read this file in, print the first five lines'. Once that runs, add filtering. Then add export.

With each addition, have it change only that one place. Don't let it rewrite the whole block. The less it changes, the more you can see what changed.

- Have it break the code into a few small blocks, each with a one-line comment above it saying what it does
- For complex logic, have it write one sentence of approach first, and write the code after you've confirmed it. If the approach is wrong, beautiful code is still useless
- Have it throw in a set of test data you can run directly, so you don't have to make your own

## 03 Paste the Error Back Verbatim, Don't Just Say 'It Won't Run'

'It errored' — those three words are useless to it. It needs to see the complete error text, especially the last line and the line number.

When you paste, give all three together: the full error, the relevant code, and what you did. Leave one out and it starts guessing, and a wrong guess costs you another round.

If it's still broken after the fix, paste the new error again, and say 'the last fix didn't solve it'. It doesn't remember what you didn't say.

- The most useful part of an error is the last line — that's where it says what went wrong and what type it is

- Don't paraphrase the error yourself. Paraphrasing loses key details. Copy and paste verbatim is fastest
- After a fix, have it write one line on 'what this fix assumes'. Then next time it breaks, you know where to start looking

**Example: how to phrase it when you paste an error back**

```
I ran the code you gave me and it still errors.

Error text:
[paste the whole thing, including the last line]

The command I ran: python clean.py
My code:
[paste the relevant code]

Please change only the lines that error. Don't rewrite the whole block. When
you're done, tell me what you changed and why.
```

## A template you can copy

Fill this in when you want code. The more specific you are, the more accurate it writes.

```
Write me a piece of code.

Environment: Python 3.10, Windows, standard library only, no pandas.
Input: orders.csv in the same directory, UTF-8 encoded, first row is the
header, columns are order number, customer, amount, date.
What to do:
1. Read the file in.
2. Keep only rows where the amount is greater than 1000.
3. Group by customer and compute each customer's total amount.
4. Sort by total amount, largest first, and print the top 10 rows.
Output: print straight to the screen. Don't write to a file.
Requirements: the amount column may have empty values. Skip those rows and
count them, then print 'skipped N rows' at the end. Add a one-line Chinese
comment to each key step.

Give me only the code first, no explanation. If anything I said is unclear, ask
me one question before writing.
```

## Common mistakes

### × Only saying 'write me a script'

It doesn't know the language, the version, or what the data looks like, so it can only give you a generic template you'll have to rewrite. State the environment and the input/output, and it's usable first time.

### × Asking for a whole system in one go

In hundreds of lines, one small mistake stops the whole thing and it's hard to locate. Ask for the smallest version that runs. Once it runs, add the next step.

### × Saying only 'it won't run' about an error

With no information it can only guess, and you burn several rounds. Paste the full error, the relevant code and what you did, all three together, and it usually gets it right in one round.

### × Having it rewrite the whole block to fix one bug

A full rewrite also changes the parts that already worked, and can drag in new problems. Have it change only the lines that error, then run it

yourself.

## Checklist

☐ My instructions state the language, the version, and whether third-party libraries are allowed

☐ I said where the data comes from and where the result should go

☐ I asked for the smallest working version first, and only added features after it ran

☐ When I pasted the error, I pasted the original text, not my own paraphrase

# 10
# Getting AI to Review Your Code

Throw code at it and ask "any problems?" and it will usually pick out a few small nits and call it done. This chapter is about getting it to check the things you actually care about, and to give you fixes you can apply directly.

---

## What this chapter covers

☐ You can get it to check the specific kinds of problems you care about, instead of just skimming

☐ You know how to have it rank the problems, so you fix the urgent ones first

☐ You can use it to understand what old code you've inherited actually does

## 01 First, Have It Explain What the Code Does

When you get hold of code someone else wrote and you're not even clear on what it does, don't rush into having it hunt for bugs. First have it explain the code in plain language.

Then ask: "Does it actually do all that? Where does it fall short?" That's when the bugs it finds become trustworthy.

Do it in the wrong order and things go sideways. Ask "are there bugs?" straight away and it hands you a pile of advice that looks professional but doesn't affect whether the program runs. You end up more confused than before.

- Have it cover input, processing and output in three sentences. If it can't, that usually means the code itself has a problem
- Have it mark the lines that matter most, then focus your reading there

## 02 Give It a Checklist

"Take a look at this for me" is too broad. All it can do is pick the most obvious things. Give it a few specific categories and it will search inside those.

The four usual categories: will it crash, will it compute the wrong answer, are there security holes, is it slow. Two at a time is enough. Ask for all four and every item comes back shallow.

Slowness usually comes down to one spot: whether a loop queries the database or re-reads a file over and over. That pattern chokes as soon as the data grows.

- Will it crash: null values, divide by zero, missing files, wrong types
- Will it compute the wrong answer: edge values, how it rounds, dates crossing months or years
- Are there security holes: passwords hardcoded in the source, queries built by gluing strings together

**Example: narrowing the scope of the review**

> Check only these two categories, ignore everything else for now:
> 1. Will it crash: null values, divide by zero, missing files, wrong types.
> 2. Will it compute the wrong answer: edge values, how it rounds, dates crossing months or years.
> For every item, state clearly: which line, and under what conditions it goes wrong.

## 03 Have It Rank the Findings, Not Dump Fifty on You

With no cap, it writes up everything you didn't ask about and your head spins. Say up front: five items max, ordered by severity.

You can let it choose its own severity criteria, as long as it explains the reasoning. Read the first two items and park the rest.

Every item needs a fix attached, ideally the corrected lines themselves. "Consider adding exception handling" says nothing. You still

have to work out how to write it.

- Ask for three tiers: must fix (it will break), should fix (hard to maintain later), optional (style preference)
- Skip the style tier every time. Changing a lot of it tends to introduce new problems
- State the usage: "runs once a day, around ten thousand rows." Only then does it know what counts as slow

## A template you can copy

Use this for code review — swap in your own context and priorities

```
Please review the code below.

Background: this is a script I maintain. It runs once a day, around ten
thousand rows of data, Python 3.10.

Work through it in this order:
1. First explain in three sentences what this code does (input, processing,
output).
2. Then check only these two categories: will it crash, will it compute the
wrong answer.
3. List 5 items at most, ordered by severity. For each one state clearly: which
line, what the problem is, under what conditions it goes wrong.
4. For each item give the corrected code. Paste only the lines that change,
don't rewrite the whole block.

Style, naming and comment issues — skip those for now.

Here is the code:
[paste]
```

## Common mistakes

### × Asking "are there any problems?"

Too broad. It picks a pile of small nits that don't affect whether the code runs, and the real trap drowns in the noise. Say which failure you're most afraid of, and have it check only those categories.

### × Pasting only a small fragment

Where a variable comes from, who calls this function — if it can't see that, it can only guess. At minimum paste the whole function together with the place that calls it.

### × Taking "this isn't safe here" at face value

It often labels perfectly normal code a vulnerability, and it can miss real problems too. For every item ask "under what conditions does this go wrong?" If it can't answer, set that item aside.

### × Applying every suggestion it gives you

Change dozens at once and you'll easily create new problems, and you won't be able to tell which change broke things. Change one at a time, run it, then touch the next.

## Checklist

☐ I had it explain what the code does before I let it hunt for bugs

☐ I told it how the code runs and how much data it handles, so it knew what counts as slow or broken

☐ I capped the number of items and required severity order plus line numbers

☐ I asked "under what conditions does this break?" for every item instead of accepting the whole list

# 11
# When Your Code Won't Run: Getting AI to Help You Debug

An error message isn't there to scare you. It's there for you to copy. This chapter covers handing AI the exact error text plus the lines that broke, and getting it to change one thing at a time.

---

## What this chapter covers

☐ You can package an error into material AI can actually work with, instead of guessing back and forth a dozen times

☐ You know which lines to paste and which to leave out

☐ You can judge whether the fix it hands you is sound

## 01 Copy the Error Text In First, Word for Word

When a program crashes it spits out a wall of English. That's the error message (the text that tells you which line it stopped on and why). Don't translate it, don't summarize it. Copy the whole thing.

A lot of people just write "my code threw an error, what do I do." There isn't one useful line in that sentence, so AI can only guess. It guesses, you try it, and a dozen rounds later half an hour is gone.

The most valuable parts of an error are the file name and the line number. With those two, it can point straight at the line that broke.

- The last line usually matters most. Don't paste only the first line
- Before pasting, replace keys and internal network addresses with XXX

## 02 Don't Paste the Whole Project — Paste the Small Piece That Reproduces It

A project runs to thousands of lines. Paste all of it and it can't read to the end, and it won't read accurately. Find the function (a small block of code you can call again and again) that's involved and paste those lines.

If you really can't tell where the problem is, delete code step by step. When the error disappears, the problem was in what you just removed. This dumb approach is faster than letting AI guess.

While you're at it, state three things: what you wanted it to do, what it actually did, and which language you're using.

- "I expected 5, it printed None" is worth more than ten lines of description
- A three-to-five-line example that reproduces the problem is the most valuable thing you can give it

**A minimal reproduction looks like this**

```
rows = [1, 2, 3]
print(rows[5]) # IndexError: list index out of range
```

## 03 One Change at a Time — and You Should Be Able to Explain It

The worst case is it rewrites eight lines in one go and says "this should do it." If it works, you don't know why. If it breaks, you know even less.

Just ask for it directly: one fix at a time, stating what was wrong and what changes after the fix. You change a line, run it once, look at the result, then ask the next question.

Once it runs, have it explain in three sentences how the code works now. If you can't explain it yourself, you'll fall into the same hole next time.

- Have it mark the lines it changed instead of rewriting the whole block

- If the same problem doesn't budge after two attempts, change approach. Don't burn half an hour

## A template you can copy

Use this for debugging — swap in your own details for the bracketed parts

```
My code won't run. Help me track it down using the information below:
[Language] Python, using pandas
[What I wanted it to do] Delete the blank rows in a table
[What it actually did] Exited immediately with an error, not one row deleted
[Full error]
(paste the original text, first line to last line, don't change it)
[The code that breaks]
(paste the lines that reproduce the problem, not the whole project)

Please answer like this:
1. First say in plain language what this error means. No jargon.
2. Point out which line most likely has the problem, and why.
3. Give one fix at a time, and say what changes after the fix.
4. If the information isn't enough, ask me first. Don't guess.
```

## Common mistakes

### × Just saying "it won't run"

No error, no code, no language — all AI can do is hand back generic advice. Give it at least three things: the full error, the lines that broke, and your language.

### × Dumping the entire project on it

Thousands of lines is more than it can read, so it samples — and it may sample the wrong place. Narrow it down first to the smallest set of lines that reproduces the problem. Three to five is ideal.

### × Asking it to "tidy things up while you're in there"

Eight changes at once: if it runs you don't know which one did the job, if it doesn't you don't know which one to revert. One change at a time, and

run it immediately.

### × Copying the fix without checking the line numbers

Its line numbers count from the snippet you pasted, not from your actual file. If they don't line up, count yourself. Don't delete the wrong line.

## Checklist

☐ I pasted the error verbatim, not my own summary of it

☐ The code I pasted runs on its own and reproduces the problem

☐ I can explain why for every fix it gave me

☐ After each change I ran it once to confirm the original behavior still works

# 12
# Getting AI to Clean Up Messy Code

Bad code doesn't always fail to run — but the moment you touch it, something breaks. This chapter is about having AI understand the code before it touches anything, cleaning up one function at a time, and keeping the changes auditable afterward.

## What this chapter covers

☐ You can understand code you don't follow before you touch it

☐ You know how to limit the scope, changing one function at a time

☐ You can read its changes and judge which to keep and which to roll back

## 01 Have It Read First — No Editing Allowed

When you're handed code you can't follow, don't open with "optimize this for me." Open with "explain what this does, line by line."

That step costs you two minutes and can save you half a day. As it explains, you'll notice the logic you assumed doesn't match what's actually there.

If it gets the explanation wrong, that's exactly the measure of how tangled the code is. If even AI has to guess, a human had better read it properly before touching it.

- Have it flag "I'm not sure about these lines" instead of forcing an explanation
- While you're there, ask where this is most likely to break

## 02 Clean Up One Function at a Time

Cleaning up (the jargon is refactoring — changing nothing about what the code does, just making it easier to read) is the last thing you want

to do as a whole-file rewrite. Touch one function, get it running, then move to the next.

Lay down three hard lines: behavior doesn't change, public names don't change, no sneaking in new features. If you don't write that down, it will absolutely help itself to extras.

If a function is too long, have it split the function first. Past fifty lines, with the indentation turning three corners — those basically all need to become several smaller functions.

- Start with the function you edit most often. Biggest payoff, least risk
- Let it rename things too: turn func1, tmp and the like into names you can read

## 03 Ask for a Before-and-After Comparison, Not a Fresh Block of Code

Have it answer in three parts: what it looked like, what it looks like now, why it changed. Read them side by side, line by line, and you'll see whether it touched your logic.

When you see it reorder conditions, swallow an exception, or squash a ten-line loop into one line, ask one more question. Under what conditions would the behavior differ?

After the change, run the original behavior yourself. If there are tests, run the tests. If not, manually click through the critical path three times.

- When it says "this is more elegant," push for specifics on how. If it can't answer, don't make the change
- Put only one function's changes in each commit, so a rollback is easy if something goes wrong

**Have it answer in this format**

```
Before:
def f(a, b):
if a == None:
return b

Now:
def add_or_fallback(a, b):
if a is None:
return b

Why this change: the name doesn't show what it's for, and the comparison isn't written strictly.
Behavior change: none.
```

## A template you can copy

When you want to clean up a piece of code, copy this whole block to it

```
I'm nervous about changing this code. Don't touch it yet — work through this in order:
[Step 1] Walk through it line by line and say what it does, in plain language. Mark any line you're unsure about directly. Don't force a guess.
[Step 2] Point out where it's most likely to break, three items at most.
[Step 3] Take only the first problem and give me one fix, in this format:
Before: ...
Now: ...
Why this change: ...
Any behavior change: ...
[Constraints]
1. Behavior must not change. Public names must not change.
2. Don't sneak in new features, and don't sneak in fixes for other bugs.

The code:
(paste here)
```

## Common mistakes

- × **A bare "rewrite this for me"**

It changes the behavior along with the code. It looks shorter and prettier, and it already does something different. Say it plainly: behavior must not change, only readability.

### × Overwriting without a backup

Copy the original file first, or make a commit. Turning working code into code that won't run, during a cleanup, is the most common accident there is.

### × Mixing bug fixes with style changes

Do them as two separate passes. Mix them into one and when something breaks you can't tell whether the logic is wrong or you fumbled something during the cleanup.

### × Names you don't recognize after it renames things

A pretty name you don't understand is the same as no cleanup at all. Ask what it means, then rename it to something you can read.

## Checklist

- ☐ Before changing anything I made a backup, or made a commit
- ☐ I touched one function at a time and ran it after each one
- ☐ I can explain the reason for every change it made
- ☐ Behavior is unchanged and public names are unchanged

# 13
# Let AI Write the Tests and Save Half the Time

Tests aren't for your boss to look at. They're the safety net for the next time you change the code. This chapter shows how to have it list what needs testing before it writes anything, and aim straight at the places that break.

---

## What this chapter covers

☐ You can get a list of what should be tested before you let it write a line of code

☐ You know bad data and edge cases are the real target, so you stop testing only the happy path

☐ When something goes red (fails), you can tell whether the code is wrong or the test is

## 01 Get the List First — Don't Rush to Code

A test is a few lines of code that check things automatically. Run them and you know whether a feature just got broken. How good your tests are depends entirely on what you chose to test. So step one isn't writing. It's listing.

Have it produce a table first: how this function gets called, what the normal cases are, where things tend to go wrong. Reading that table is worth more than reading a hundred lines of tests.

Whatever's missing from the list, you add. You're the one who knows where you tripped up last week. Have it write a test for exactly that spot.

- Thirty percent normal cases, seventy percent bad data and edge cases
- Edge values (the numbers sitting right at the limit, like an empty string or 0) get listed separately

## 02 Bad Data Is the Main Event

Most people write tests that only check "input 1 and 2, output 3". That kind of test is always green and never catches anything. It's as good as not written.

What actually blows up is this stuff: null values, negative numbers, very long text, badly formatted dates, duplicate submissions, a dropped connection. Have it write one test per line on that list.

Every test needs an assertion — the line that says "the result should equal this". If it doesn't match, the test fails. A test with no assertion is a test that doesn't exist.

- One test checks one thing, and the name should tell you what it's guarding against

**What the generated tests look like (Python example)**

```
import pytest

def test_normal_input():
assert parse_age("18") == 18

def test_empty_raises():
with pytest.raises(ValueError):
parse_age("")
```

## 03 Run Them, Then Make It Explain the Red Ones

Always run the tests once they're written. Red (failing) is the good news. It means the tests actually caught something.

Paste the whole failure output back and ask: is the code wrong, or is the test wrong? Those two call for opposite fixes, so make it pick one,

clearly.

What you have to watch for is it quietly changing the expected value just to get green. Before you run, check whether it touched the assert line.

- Keep test files separate. Don't mix them into the same file as the business code
- Run them after every change. It takes ten seconds

## A template you can copy

Use this as your opener when you want tests written

```
Write tests for this function. Don't write any code yet:
[Step 1] First give me a table: how this function gets called, what the normal cases are, what the bad data looks like, where the edges are.
[Step 2] After I confirm, write one test per line on the list, each checking one thing.
[Step 3] Every test needs an explicit assertion. No "close enough" judgments.
[Limits]
1. Don't modify the function under test.
2. When a test fails, say clearly whether the code is wrong or the test is. Don't change the expected value to force green.

Function:
(paste it here)
```

## Common mistakes

### × Testing only the happy path

A test like "input 1 and 2, output 3" is always green and never catches anything. Nulls, negatives, oversized input and bad formats are where the work is.

### × Letting it edit the test to force green

When a test fails, it will very likely nudge the expected value until it passes. The test goes green and the bug stays. Before you run, check whether the assertion line moved.

### × Generating a pile of tests and running none of them

A test you never ran is a test you never wrote, and it may be broken itself. Run them the moment they're written. Red is the good news.

### × Tests you can't read yourself

A test you can't read is one you won't dare change and won't trust. One test, one thing. If you can't follow it, have it rewritten.

## Checklist

☐ I asked for the list first and confirmed what needed testing before it wrote any code

☐ I listed bad data and edge values separately instead of only testing the normal case

☐ I ran the tests right after they were written instead of just accepting them

☐ It didn't touch the code under test, and it didn't quietly change the assertions

# 14
# Let AI Fill In the Comments and Docs

Old code with no comments. Three months later you can't read it either, and a new hire won't dare touch it. This chapter shows how to have AI fill in the comments and docs without letting it mess with your code.

## What this chapter covers

☐ You can have AI comment a block of uncommented code without moving a single line of logic

☐ You know what a comment should say and what it shouldn't, so you don't get buried in filler

☐ You can have AI draft a README and function docs, then touch up two things and hand them over

## 01 Have It Explain the Code First, Then Write

Don't open with "add comments for me". First have it walk through the code: what it's for, what goes in, what comes out, which lines are most likely to break.

If the explanation is wrong, it didn't understand the code. And if it didn't understand the code, the comments will be wrong too. Once the explanation is right, have it turn that explanation into comments and put them back into the code.

The order matters. Explaining forces it to walk the logic from start to finish, which is far more reliable than making it produce comments cold. As a bonus, you've just reviewed that code yourself.

- Explain first, comment second, and judge in between whether it actually understood
- If words like "probably" or "should be" show up in the explanation, it isn't sure about that part. Watch those lines closely

- Feed it one file or one function at a time. Don't dump the whole project in

## 02 Comments Explain Why, Not What

A good comment only says what the code itself can't show you. Why it's written this way. What's tricky about this parameter. Which part of the business this belongs to.

A bad comment reads the code back to you in plain language. The line increments i, and the comment says "increment i". It takes up space, helps nobody, and goes stale.

By default it loves restating the code, and it will do exactly that unless you stop it. So put a hard line in the prompt: no restating. Only reasons and things to watch out for.

- At the top of a function, cover three things: what it does, what the parameters mean, what it returns
- Flag tricky spots with their own line, e.g. "this must be null-checked — upstream can pass null"
- Assignments, loops and prints that read at a glance don't need comments

**Example prompt**

> Here's a piece of code. Do this in order:
> 1. First explain in three sentences what it does, what goes in, and what comes out.
> 2. Point out the one to three lines most likely to cause trouble, and say why.
> 3. After I confirm, add comments to each function.
> 4. Comments only cover "why it's written this way" and "what's tricky here". Do not translate the code.
> 5. Do not change a single line of code. Only add comments.

## 03 Docs Are Where It Makes Things Up — Feed It the Truth First

The place things go wrong most often is asking it to write a README (the project's manual — the first thing anyone sees when they open your repo). It has no real information, so it mostly guesses.

It has never seen how your project starts, which port it runs on, or how the dependencies get installed. So it writes the usual template. It looks right. It doesn't run.

The right way is to paste the real facts in first: the start command, the directory layout, which environment variables need setting. Give it the truth and it can write something accurate.

- Paste the real start command and directory layout before it writes anything
- Where it's unsure, have it write "to be filled in". It is not allowed to invent something
- Afterwards, run it against your actual environment. A command that doesn't work was made up

## A template you can copy

Use this when documenting a whole module — copy it, change two spots, and go

```
Here is my project's directory structure and start command. Write a README from them.
Directories:
(paste the directory tree)
Start command:
(paste the command you actually use)

Requirements:
1. Only write what I've given you. Don't add anything you haven't seen.
2. Anywhere you don't know, write "to be filled in". Never invent a plausible-looking answer.
3. Fixed structure: what this project does, how to install it, how to start it, what each folder is responsible for.
4. Short sentences throughout. No empty words like "powerful" or "efficient".
```

## Common mistakes

### × Letting it refactor while it's in there

You say "add some comments" and it may refactor along the way, quietly changing the logic. Put a hard line in the prompt — "only add comments, don't change code" — and run a diff on the result to see which lines moved.

### × Dumping the whole project in at once

With many files it loses track — it reads the first ones and forgets the rest. One file at a time, one module at most, then move on to the next.

### × Comments stuffed with filler

A comment on every line makes the code harder to read, not easier. Comment the top of functions, the complicated conditionals, the tricky spots. Leave plain assignments and loops alone.

### × Sending the README out as-is

The install commands, port numbers and config entries in it are probably invented. Run through it on the real machine, fix what doesn't work yourself, then send it.

## Checklist

☐ I had it explain the code first and only asked for comments once I was sure it understood

☐ I put "only add comments, don't change code" in the prompt, and I diffed the result

☐ The comments give reasons and warnings, not a translation of the code

☐ I actually ran every command in the README

Part Three

# Integration and Engineering

Chapters 15 — 21

# 15
# Your First API Call: Wiring a Model Into Your Program

Chatting on a web page gets you one exchange at a time. Wire the model into a program and you can work in batches. This chapter walks you through the first call: where the key goes, how to send the request, how to read the result.

---

## What this chapter covers

☐ You know where the key belongs and where it must never go, so a single commit can't leak it

☐ You can send your first request and print the model's answer on your own screen

☐ You can tell which part of the response is the answer and which part is the tokens this call burned

## 01 An API Is That Wrapper Taken Off

Chatting with a model on a web page is also just sending requests. The page has already wrapped the request for you; all you do is type.

An API (the interface programs use to talk to each other) is that wrapper taken off. You send a piece of text in a set format, and it sends back a piece of text in a set format.

So hooking up an API isn't hard. There are only three things to nail down: what address to use, where the key goes, and what the request looks like. Get those three straight and switching to another provider is the same routine.

- Address: the URL the provider gives you, usually called base_url

- Key: a long string of characters that works like an account password. Leak it and someone else spends your money
- Format: the big providers' request formats are close enough that learning one makes the next one quick

## 02 Version One Only Has to Run

Don't make version one complicated. There's one goal: send a sentence, print the answer. Get that running, then add things on top.

Failures usually come in three flavors: the key is wrong, the address has an extra or missing slash, the field name is misspelled. The error message generally tells you which one.

Once it runs, resist wrapping it in a function right away. Run the crudest version three times, confirm it's stable, then tidy it up. That saves a lot of back-and-forth. What this step really does is eliminate unknowns one at a time: one successful run tells you what the address, the key and the format are supposed to look like.

- Check the status code first: 200 is success, 401 usually means a bad key, 429 means you're sending requests too fast
- Pull the answer with choices[0].message.content — don't drop that [0]
- Record the usage, and estimate the total before you run a batch job

**Example code (Python — replace the three placeholders with your own values)**

```
import requests

resp = requests.post(
"your-provider-url/v1/chat/completions",
headers={"Authorization": "Bearer your-api-key"},
json={
"model": "the-model-name-your-provider-gave-you",
"messages": [{"role": "user", "content": "Explain what a cache is in one
sentence"}],
},
timeout=60,
)

print(resp.status_code)
print(resp.json()["choices"][0]["message"]["content"])
```

## 03 In the Response, You Only Read Two Places

The response is a chunk of JSON (a data format built out of curly braces that programs read easily). You don't need all of it. Two places matter.

choices holds the answer. That content field is what the model said, and you can pull it out and use it directly.

usage tells you how many tokens this call cost, counted separately for the input and the output. Look at that number before you run a batch so you know what you're in for.

- Read the status code before the content — when the status code is wrong, the content is nothing but error text
- Log the whole response, so you have something to check when things break
- Within a single request, the longer the history you send, the more tokens it costs

## A template you can copy

Run this prompt when you switch to a new provider — it gets you a minimal script

```
Write me a minimal Python example that calls this API. Requirements:
1. Use only requests. No third-party SDK.
2. Read the provider URL, the key and the model name from environment
variables. Don't hard-code them.
3. Send "hello" and print the status code and the returned content.
4. A timeout is mandatory.
5. Tell me what to check for each of these status codes: 401, 429, 500.
6. No retry logic, no wrapping it in a class. Just one script that runs as-is.
```

## Common mistakes

### × Hard-coding the key in your code

The moment you commit it to the repo, it's public. Put it in an environment variable (a value stored by the system that your program can read but that never appears in the code), or in a local config file — and then add that file to your ignore list.

### × Pasting your real key to the AI

When you have AI help you debug, swap the key for the words "your key" before pasting. It doesn't need the real one to get your code right.

### × Thinking one request is one conversation

The API has no memory. It doesn't know what you said last time. For a multi-turn conversation you have to bring every earlier message along and send them together.

### × No timeout

When the network stalls, your program sits there waiting and the whole job hangs. Put a timeout on every request. Sixty seconds is generally enough; very long text can go higher.

## Checklist

☐ The key isn't in the code and has never gone into the repo

☐ I got the minimal example running and saw the model's answer before writing anything else

☐ Every request I send has a timeout

☐ I know that multi-turn conversation means bringing the history along myself

# 16
# One Character at a Time: Streaming Output

Waiting for a whole answer to appear takes a dozen seconds. The screen just spins and people assume it has frozen. Streaming output makes the words pop out one by one, so you see something in the first second. This piece covers how to wire it up.

## What this chapter covers

☐ Know what streaming output is, and why the feeling of waiting disappears

☐ Can turn on streaming in the request and stitch the little fragments back into a complete answer

☐ Avoid three traps: empty chunks throwing errors, no end check, and losing everything when the connection drops

## 01 Why waiting for one whole block feels so bad

By default, the model writes the entire answer and returns it in one go. Three hundred words can take a dozen seconds, and for those dozen seconds your interface does nothing at all.

Streaming output (the server sends as it writes, you display as you receive) spreads that process out in the open. It writes one character, you get one character. What you see is the progress of it writing.

The total time doesn't get one bit shorter, but it feels completely different. There are words on screen in the first second, so people don't think it's stuck, and they stop hitting refresh. So this isn't a performance optimization. It's an experience optimization.

- Same total time, but the first screen goes from a dozen seconds to under one
- The typewriter effect isn't decoration. It tells the user "I'm working"
- Long answers, long documents and writing code — these three cases benefit most

## 02 Turning it on is really just adding a switch

Add a field to the request that tells the server "don't hold it all in, send a bit as you write". The field is usually called stream, and you set it to true.

Once it's on, what comes back is no longer one block of JSON but data line by line. Each line starts with the characters data:.

Each line holds one small piece of the answer, usually just a few characters. All you have to do is stitch those pieces together in order into one complete passage.

- You have to turn it on in the request, and set the client to streaming while it reads. Both places need changing
- The first few pieces are often empty — they carry only role information, no text
- An end marker arrives at the end. Stop when you see it, don't keep waiting

**Example code (Python, prints as it receives)**

```
import requests, json

resp = requests.post(
"your provider URL/v1/chat/completions",
headers={"Authorization": "Bearer YOUR_API_KEY"},
json={
"model": "the model name your provider gave you",
"messages": [{"role": "user", "content": "Write a 100-word
self-introduction"}],
"stream": True,
},
stream=True,
timeout=60,
)

full = ""
for line in resp.iter_lines():
if not line:
continue
text = line.decode("utf-8").removeprefix("data: ")
if text.strip() == "[DONE]":
break
chunk = json.loads(text)
piece = chunk["choices"][0]["delta"].get("content", "")
full += piece
print(piece, end="", flush=True)

print("\nTotal characters:", len(full))
```

## 03 Where the stitching goes wrong

A single piece might be half a word, or just one punctuation mark. Don't rush to judge the content of a piece. Stitch it into the full passage first, then talk.

Give the content lookup a default value. Some pieces have no text field at all. Read it directly and you get an error, the program stops on the spot, and everything received so far is wasted.

The closing line isn't the answer. It's a special marker. If you don't stop, the loop either spins forever or just hangs without returning. Hanging doesn't even raise an error, which makes it the hardest kind to track down.

- Don't run keyword checks, translation or format validation on a single piece
- If the network drops, the stream drops. Keep the part you already received on screen
- Save as you receive, and flush the output buffer after every piece

## A template you can copy

Have it write you a more robust streaming endpoint

```
Please write me an example that calls a streaming API, with these requirements:
1. Use requests to read the stream and print as you receive. Don't wait until everything has arrived.
2. Print each piece the moment it arrives and flush the buffer, to get the typewriter effect.
3. Skip empty pieces and pieces that only carry role information, and stop when you hit the end marker.
4. Accumulate the received text into a variable at the same time, and print the total character count at the end.
5. Add exception handling: when the network drops, keep what was already received and show "the answer was interrupted, you can retry".
6. Don't use a third-party SDK, and don't wrap it in a class.
```

## Common mistakes

### × Only half switched on

You set stream to true in the request, but the client isn't reading in streaming mode. The program still waits for all the data before returning — you may as well not have bothered. Both places need changing.

### × Processing each piece on its own

A piece may be half a word. Matching it against keywords, translating it or validating it on its own is all wrong. Stitch it into a complete

sentence first.

### × No end check

The stream doesn't stop by itself. You have to see the end marker and exit the loop. Leave that line out and the program can hang and never return, and it's hard to track down.

### × Everything lost when the connection drops

The user has already read three hundred characters, the network drops, and the interface goes blank. Terrible experience. Save as you receive, tell them they can retry, and keep what already arrived.

## Checklist

☐ I turned on streaming in the request, and set the client to streaming reads as well

☐ I stitch the fragments into a complete answer before doing anything else with them

☐ My code checks for the end marker, and handles empty pieces with no text

☐ When the network drops, what the user has already seen doesn't disappear

# 17
# When It Breaks: Retry and Fallback

AI making mistakes isn't an accident. It's normal. This piece shows you how to sort errors into three kinds, each with its own retry and fallback plan, so the work still gets done.

## What this chapter covers

☐ Can tell at a glance whether the network hiccuped, the format went off the rails, or the task is simply beyond it

☐ Can write retry logic: how many attempts, how long to wait, and when you have to give up

☐ Every call has a fallback, so one crash doesn't jam the whole pipeline

## 01 First, work out which kind of error it is

AI errors come in roughly three kinds. The first is the connection not going through: timeouts, rate limiting, a pile of unreadable text coming back. Retrying usually fixes this one.

The second is the wrong format: you ask for JSON and it gives you prose; you ask for three fields and it gives you five. Fixing the prompt fixes this one.

The third is that it genuinely can't do it: the information isn't in the material at all, or the task is beyond its ability. Retrying a hundred times won't help.

There's a nastier one as well. The answer looks perfectly reasonable, but the content is wrong. This one doesn't raise any error at all. Only your own eyes can catch it.

- For network errors, wait and retry. For format errors, nail the requirements down and retry. For ability errors, change approach — don't force it
- If the same error comes back identically three times in a row, the prompt or the task itself is the problem. Fix that first
- Save the original error text every time. It's a hundred times more useful than the words "it failed"

## 02 Retries need a number: three attempts, a few seconds apart

The least effort version is "try at most three times, with the gap growing each time". Wait 2 seconds after the first failure, 5 after the second, 10 after the third.

Why grow the gap? Because most errors mean the other side is busy. Slam into it again immediately and you'll mostly just hit the same wall.

If all three fail, stop. Don't keep trying. Unlimited retries only blow up your bill and your logs together.

- Only retry errors worth retrying. An expired key or a malformed parameter will still be wrong after ten thousand retries
- Retries have to be idempotent (sending the same request several times gives the same result). For sending notifications, placing orders or charging money, check whether it already succeeded before you retry
- Raise an alert once failures pass the count you set. Don't let it quietly keep failing
- Log each request before every retry, so after three failures you can follow the logs to which input it was

**Example code: three retries, with the gap growing each time**

```
import time

def ask(prompt, tries=3):
waits = [2, 5, 10]
for i in range(tries):
try:
return call_model(prompt)
except (TimeoutError, RateLimitError) as e:
print(f"Attempt {i + 1} failed: {e}")
if i < tries - 1:
time.sleep(waits[i])
return None # hand it to the fallback
```

## 03 Fallback: even on failure, give an answer

A fallback is "which way to go when the main road is blocked". The simplest fallback is a default value: if the summary can't be generated, return the first three hundred characters of the original text.

One step further is degradation: if the main model times out, switch to a faster model, or just skip this step and finish the rest first.

The most important thing is not to fail silently. Errors need a trace and a place where they can be seen. Otherwise the user just sees a blank page.

- For every call, ask yourself "what happens if this dies". The answer can't be "the whole pipeline stops"
- Mark fallback results, so people know this is a fallback and not the normal result
- Cache when you can. If the same input has been computed once, don't ask again every time
- Log at least three things: which step failed, what the original error text was, and which fallback path it finally took

## A template you can copy

Have the AI add retries and a fallback to a piece of code

```
The code below calls an AI API and currently has no error handling at all.
Please add retries and a fallback:
1. Retry only on timeouts and rate limiting, at most three times, with gaps of
2 seconds, 5 seconds and 10 seconds.
2. Don't retry parameter errors or key errors. Report them straight away.
3. When all three fail, take the fallback: return a hard-coded default value
and write a log line carrying the original error.
4. Record the retry count and the final outcome, so I can check afterwards.
5. Explain each change you made one by one. Don't rewrite the whole block.

Code:
(paste your code here)
```

## Common mistakes

### × Retrying the moment anything errors

Retrying without reading the reason is like hitting the wall with your eyes closed. Read the line that came back first. Most errors say what's wrong themselves: a timeout, quota used up, or a malformed parameter.

### × Retrying forever

Write a loop that keeps trying and it becomes an infinite loop when the other side has an ongoing outage, with the bill and the logs exploding together. Cap the count, say three, then take the fallback and raise an alert.

### × Retrying where money gets charged twice

For sending notifications, placing orders or charging money, a retry may do the thing twice. Check the status once before retrying, or give each request a unique reference number and let the other side deduplicate on it.

### × Swallowing errors silently

Wrap a layer of error handling around it but do nothing, and the program doesn't crash — the problem is hidden too. At the very least, write a log, count the occurrences, and let people see it in monitoring.

## Checklist

☐ I can say which category this error belongs to, instead of just saying "it broke"

☐ Retries have a cap and a growing gap, not frantic hammering in place

☐ Every call has a fallback, so the flow can keep going when something errors

☐ Fallback results get marked and errors leave a log. Nothing quietly disappears

# 18
# Don't Let the Bill Run Away: Tokens and Cost

AI charges by the character. The more you stuff in, the more you pay. This piece shows you how to work out what one call costs, then use a few small habits to push the bill back down.

## What this chapter covers

☐ Know where the money goes: what you send and what comes back both cost money

☐ Can cut cost with three moves: stuff in less background, rein in the output, pick the right model

☐ Can work out roughly what a feature costs per month, so the bill doesn't make you jump

## 01 First, understand how it charges

AI charges by token. A token is its unit of billing — you can think of it as "a small piece of text". One Chinese character is roughly one to two tokens.

The key point is that both what you send and what it returns cost money. Paste in a five-thousand-character document and it charges you for reading it once.

So the first rule of saving money isn't asking less. It's stuffing less. If the background material can be condensed to three hundred characters, don't paste three thousand.

Also, many platforms quote prices per thousand tokens. Unify the unit before you compare prices. Don't mistake a thousand for one.

- Output generally costs more than input. Making it waste fewer words saves more than making it read less
- Every turn of a conversation re-bills all the history before it. By turn twenty you're paying for the first nineteen rounds over and over
- Summarize a long document yourself first, then hand it over. That often saves more than half

## 02 Three money-saving moves that work immediately

First: rein in the output length. Just write "no more than 200 words" or "give only three items, one sentence each". Don't let it write eight hundred words when you're only going to read three lines.

Second: start a new conversation in time. When one thing is done, reopen. Don't cram ten unrelated things into one conversation.

Third: pick the right model. For rewriting copy or tidying up a table, a cheaper small or mid-size model is completely enough. Only genuinely hard work needs the strongest one.

- Write your fixed background as a template block and paste one block each time. Less hassle and fewer characters than explaining it again
- Ask it to "give only the answer, no explanation of the process" and the output can be cut in half
- For batch jobs, test-run ten items first to see what it costs, then decide whether to run ten thousand
- For the same job, compressing the prompt from three hundred characters to eighty cuts the cost, and the results are often steadier too

**Example: tighten the prompt and the output gets shorter by itself**

```
Task: turn the meeting notes below into a to-do list.
Output format: only three items, one per line, in the form "who - what - when".
Length: no more than 100 characters in total.
No preamble, no summary, no explanation of your reasoning.

Meeting notes:
(paste the notes here, keep it under 500 characters)
```

## 03 Work out the monthly bill for one feature

The arithmetic is simple: what one call costs, times how many times each person uses it per day, times the number of people, times 30 days.

Here's an example. Suppose one call is about 800 tokens in and about 400 tokens out, and it runs 500 times a day. That's fifteen thousand times a month. Multiply by the published unit price and you'll know whether the money is worth it.

Once you've done the math, the problem is usually not the number of users. It's the waste inside each call. Pushing the per-call cost down works better than cutting features.

Unit prices differ between platforms. Check the official site before you do the math. Don't estimate from memory.

- Set a monthly cap first. When you hit it, stop or degrade. Don't find out when the bill arrives
- Log the token count of every call. The most expensive step stands out immediately
- When the monthly cost beats your expectations, look at the most expensive step in the logs first. Waste usually clusters in one or two places — fix those

## A template you can copy

Work out the monthly cost of a feature — just use this

```
Help me work out the monthly cost of this feature, following these steps:
1. Estimate one call: roughly how many tokens in, roughly how many tokens out.
2. Using the published unit prices, work out the cost of the input and the
output separately, add them up for the per-call cost.
3. Estimate usage: how many people will use it, and how many times per person
per day.
4. Per-call cost × calls per day × number of people × 30 gives the monthly
cost.
5. Give three concrete changes that would lower the per-call cost, and for each
one say roughly what percentage it would save.
Where my numbers are missing, assume conservative values first, and clearly
mark them as assumptions.

Feature description:
(one sentence on what this feature does)
```

## Common mistakes

### × Pasting in the whole document

Stuffing in a file dozens of pages long means you're billed for the whole text after one read, while most of it has nothing to do with the question. Pull out the two or three relevant paragraphs yourself first, or summarize it section by section and then ask with the summaries.

### × One conversation all the way to the end

Every turn re-bills the history before it, so the longer you chat the more expensive each turn gets. One thing, one conversation. Reopen when it's done, and paste the necessary background again.

### × Using the strongest model for everything

For fixing typos or sorting things into categories, the strongest model and an ordinary one give similar results, at several times the price. Tier by difficulty: strong models for the hard work, cheap ones for the daily stuff.

### × Setting no cap at all

No budget, no alerts, and by the time you notice, a big pile has already been spent. Set a spending cap on the account, add a daily call cap in the

program, and degrade or stop when you hit the line.

## Checklist

☐ I know both input and output are billed, so I don't paste in whole long documents that aren't relevant

☐ I spell out the output length and format, and don't let it ramble on freely

☐ One thing, one conversation. I reopen when it's done instead of dragging the history out longer and longer

☐ Before launch I worked out the per-call and monthly cost, and set a spending cap

# 19
# So Many Models: Which One Do You Actually Pick

Expensive doesn't mean right, and cheap doesn't mean useless. This chapter gives you a table for picking a model by the job, plus a selection process you can finish in thirty seconds.

## What this chapter covers

☐ Know the three things models really differ on, so parameter counts and leaderboards stop leading you around

☐ Pick a tier straight off the job in front of you, instead of agonizing every single time

☐ Test with your own five real tasks and let the results decide, instead of listening to marketing

## 01 The Three Things Models Actually Differ On

First, can it think? Some models are good at multi-step reasoning — steady at writing code, working through logic, spotting contradictions. Others just write well. The moment a task gets complicated, the gap shows up immediately.

Second, can it read images? If you need it to read screenshots, scans or charts, you have to pick one that accepts image input.

Third, is it fast, and is it expensive? Within the same tier of capability, some respond quickly at a low price, and some are slower but steadier.

- Leaderboard rankings are only a reference. The questions they test are often not the same thing as the work in your hands

- A bigger parameter count doesn't mean a better fit for you. Enough is enough — the big ones are usually slower and more expensive
- Models get retired, renamed and upgraded. Don't hard-code your choice into the code; leave a switch so you can swap it whenever you want
- On the same model, turning on "deep thinking" makes it noticeably slower and more expensive. Don't turn it on for simple work — that's pure waste

## 02 Pick by the Job, Not by Reputation

Everyday text work — rewriting copy, writing emails, polishing, translating, coming up with titles: use a cheap small or mid-size model. It's fast and good enough, and you won't wince when the volume grows.

Work that needs real thinking — writing code, chasing down an error, multi-step analysis, checking a long document for contradictions. For this, go to the strongest model. The extra money you spend buys back your own time.

Work that needs image reading — a screenshot of an error, reading an invoice, reading a photo of a table: first confirm it accepts image input. If it doesn't, being the strongest model makes no difference.

There's one more category that gets overlooked most often: turning a big messy pile of text into a table. A cheap model is enough for this. Don't waste your strongest one on it.

- A whole long document: pick one with a large context window (the ceiling on how much text it can hold at once), or it gets cut off partway through
- Strict output format (JSON, a table): pick an obedient one — the kind that outputs only the format and adds no chatter
- Someone is waiting to see the result: favor a fast one. Three seconds slower and that person may be gone
- Don't ask "which model is the best". The AI's answer follows whatever the mood on the internet is, and it never lands on your actual

job

## 03 Decide in Thirty Seconds: Give One Task to Two Models

Don't guess. Take one task from your real work — ideally the one that burned you before.

Send the exact same prompt to two or three candidate models. Lay the results side by side and look at three things: can you use it as is, are there any hard errors, how long did it take.

Usable as is, no errors, fast enough — that's your model. Test it once, and you won't have to agonize over it again for the next six months.

Run every test in a fresh conversation. The context from the previous task bleeds into this one, and the result stops being accurate.

- Test at least five tasks. One task won't show you the difference — you're just riding on luck
- Save the tasks you tested and the results. When a model gets updated, rerun the same set and the verdict shows up immediately
- Write the date down while you're at it. Six months later, when the model has been replaced, you'll still be able to match up which version you tested

**Example: a comparison test record sheet — copy it into a spreadsheet and use it**

> Test task: rewrite this customer complaint as a reply under 100 words, polite in tone but don't over-apologize.
>
> | Model | Usable as is | Hard errors | Time taken | Notes |
> |---|---|---|---|---|
> | A | two edits | none | 8 s | |
> | B | | | | |
> | C | | | | |

## A template you can copy

When you're not sure, use this to have the AI help you choose

```
I have a real task on hand. Help me pick a model. Don't give me rankings — follow the format below:
Task: _____
Roughly how long is the material I need to process in one go: _____
Does it need to read images: yes / no
Can I wait: I can wait a dozen seconds / I need a result within a few seconds
Volume: about _____ times a day

Requirements:
1. First tell me which capability this job leans on hardest (reasoning, image reading, speed, long documents), and pick only the single most important one.
2. Give me two recommendations, a "good enough" tier and a "best available" tier, with one sentence on why for each.
3. Say clearly what goes wrong if I pick the wrong tier, and how I would notice.
```

## Common mistakes

### × Picking a model off the leaderboard alone

Leaderboards mostly test exam questions, which are not your job. Spending half an hour testing five of your own real tasks is more accurate than reading ten leaderboards.

### × Running the most expensive model on everything

For fixing typos, sorting into categories, converting formats — a cheap model does just as well, at a fraction of the price. Split by difficulty and you save most of the money.

### × Hard-coding the model name into the code

Models get retired, renamed, and repriced. Put the model name in a config, leave a one-switch toggle, and you don't touch the code when you change it.

### × Trying to save a bad prompt by switching models

If your requirements are vague, ten different models will all be vague. Write the task, the format and the examples clearly first, then think about switching models.

## Checklist

☐ I pick models by the job, not by reputation or leaderboards

☐ For the three needs — image reading, long documents, speed — I confirmed support first

☐ I tested with my own five real tasks instead of choosing on a feeling

☐ The model name lives in a config, swappable at any time, not hard-coded in the code

# 20
# The Context Window: How Much Text It Can See at Once

There's a ceiling on how much text it can see at once. Once you understand where that ceiling is and how to estimate it, you'll know why long documents fall apart partway through a conversation, and how to break them up.

## What this chapter covers

☐ Know what the context window actually limits, and stop assuming it remembers all your material

☐ Roughly estimate how much of the budget a piece of material eats, so you can judge in advance whether it fits

☐ Know exactly how to rescue it when it starts drifting halfway through a conversation

## 01 It Can Only See So Much Text at a Time

The context window (that's all the text it can see at once in a single conversation — what you send and what it replies are both in there) has a ceiling. That ceiling is usually counted in tokens.

A token (the small fragments the AI chops text into, where one character or half a word counts as one) is its unit of measure. What you send, what it replies, the files you paste, the settings the system gives it — all of it comes out of that one budget.

What happens when the budget runs out? The earliest things said get pushed out. It won't warn you. But ask it "what did I say earlier?" and it starts making things up.

- Mainstream models generally handle anywhere from tens of thousands to hundreds of thousands of characters at once. It depends on the specific model, so try it once before you rely on it
- Its replies use budget too. Ask it to write something long and there's less room left to hold your material
- The budget is per conversation, not permanent. Open a new conversation and the count starts over

## 02 How to Estimate What Your Material Costs

The easiest way is to look at the character count. One Chinese character is roughly one to one and a half tokens; one English word is roughly a bit over one token.

So for a 30,000-character report, budget roughly 40,000 to 50,000 tokens. If the model you're using only has a 30,000 budget, it won't fit.

You don't need to be precise. All you need is one judgment: can this pile go in all at once, or does it have to be split.

- One A4 page of Chinese is about 800 to 1,000 characters, so ten pages is around 10,000
- Tables, code and URLs eat budget especially fast — the same character count takes up more room
- If you're not sure, paste half of it first and see whether it answers smoothly

## 03 When the Budget Runs Low, It Starts Getting Confused

Talk long enough and you'll notice it getting dumber: it forgets requirements you gave earlier, it commits an error again that you just corrected. It isn't slacking off — those words got pushed out of the window.

There's something counterintuitive here too: the fuller you pack it, the worse it gets at finding the point. The middle section is what gets skipped most. It remembers the beginning and the end most clearly.

So don't chase getting everything in at once. Fill it to eighty percent and leave some headroom — the answers are often better.

- Put important requirements at the very front, or restate them once at the end
- At a key point in a long conversation, have it summarize into one paragraph first, then open a new conversation and carry on
- Once you notice it starting to forget things, stop forcing the conversation. Re-explaining the background is faster

## A template you can copy

Before you paste long material, have it do the budget math for you

> I'm about to send you some material. Don't rush to answer. Go in this order:
> 1. First tell me roughly how many characters and how many tokens this material is — an estimate is fine.
> 2. Tell me, given your current context window, how much headroom is left after it's in there.
> 3. If the headroom is under thirty percent, suggest how I should split it first, and wait for me to confirm before you start.

## Common mistakes

### × Assuming it remembers every file you sent earlier

The window is finite; once you go over, the earliest content is dropped. For material it must remember throughout, paste it again at the start of each conversation, or paste only the few relevant pages.

### × Pasting an entire manual in one go

Once it's packed full it can't hold onto the point, and the middle content is basically ignored. Filter out the two or three relevant chapters yourself first, then paste those.

### × Using character count as the only measure

At the same character count, tables and code take up more room, and Chinese and English differ too. Leave an extra thirty percent of headroom

when you estimate.

### × Keeping the conversation going after it gets confused

That means the window is full. Have it summarize its conclusions into one paragraph, open a new conversation, paste that in and continue — far faster than correcting it over and over.

## Checklist

☐ I know the window is finite, and that going over it drops the earliest content

☐ Before pasting long material, I estimate the character count and the remaining headroom

☐ I put important requirements at the beginning or the end, never in the middle

☐ When it starts forgetting things, I summarize and open a new conversation instead of correcting it by force

# 21
# How to Feed Long Documents to an AI

Throw a few dozen pages in whole and all you get back is a useless summary. This chapter covers how to split it, how to ask, and how to check — so it actually reads the document.

## What this chapter covers

☐ Split a long document into chunks it can handle, instead of forcing the whole thing in

☐ Know what kind of question gets you an answer with a source you can trace

☐ Build a document Q&A process you can use over and over

## 01 Don't Rush to Throw the Whole Thing In

Hand most people a fifty-page PDF and their first reaction is to upload the entire thing, then type "summarize this for me". What comes back is a paragraph of empty words that anyone could have written.

The problem isn't that it's dumb. It's that what you're asking for is too vague. Ninety percent of that document has nothing to do with the job you're doing this time, so it can only spread its effort evenly. It's like asking someone to find one sentence in a book without telling them which sentence.

The order should be reversed: say what you're trying to do first, then decide which part it gets to see. Those two minutes save you several rounds of rework later.

- Ask yourself one question first: what am I trying to get out of this document? If you can't say it in one sentence, don't ask yet

- If you can pin down the chapters, paste only those chapters
- Tables and images in a PDF often come out scrambled when converted to text. Glance over it yourself before you paste

## 02 Cut It Into Chunks and Ask One Chunk at a Time

Once a document passes ten or twenty thousand characters, cut it up. Splitting by chapter is the least trouble, and keep each chunk to a few thousand characters. Five or six chunks is usually about right — it can read each one straight through.

Ask the same question of every chunk, then have it combine the answers from all the chunks at the end. That way it only has to focus on one short passage each time, and the answers come out more accurate.

When you cut, don't slice a sentence through the middle, and don't separate a heading from its body text. The heading is its signpost — pull them apart and it has no idea which part this passage is about.

- Add one line at the top of each chunk — "this is chapter 3 of a certain report, covering cost structure" — and it won't get confused
- Require a sentence from the original text as evidence with every chunk's answer
- After cutting, try one chunk first. If the answer is wrong, fix the question — don't run all of them in one go

## 03 Make It Cite Sources So You Can Check

The thing to fear most in document Q&A is it stitching several passages together and sounding entirely plausible, when the original text never said that.

The fix is simple: require it to quote the original sentence for every conclusion. If it can't quote one, that passage was made up. It won't volunteer that the passage doesn't actually answer the question.

You don't have to read the whole document to check. Just read the few sentences it quoted and you're done in a few minutes. Don't skip this

step — the time you save here comes due sooner or later.

- Ask it to state which page or which section, so it's easy for you to flip back
- When it says "the document doesn't mention that", first confirm whether you actually failed to paste that part
- For anything involving numbers and dates, always go back to the original and check

## A template you can copy

The standard document Q&A prompt — fill in the blanks and it's ready

```
Below is part 3 of <document name>. The original text is between the 【】 marks.
【paste this chunk of original text here】

Answer only from the content inside 【】. Do not fill in gaps with your own knowledge.
1. Answer my question in three sentences: put your question here.
2. After every conclusion, cite the original sentence it rests on. If there's no support in the original text, don't write it.
3. If 【】 contains no relevant information, just say "this part doesn't mention it". Don't guess.
```

## Common mistakes

### × Uploading the whole PDF and asking for a summary

You get a page of empty words spread evenly across everything, with all the emphasis lost. Figure out what you want first, and paste only the relevant chapters.

### × Asking ten questions at once

It mixes the answers together and gives you half an answer to each, and you still have to follow up one by one. Ask one, get the answer, then ask the next.

### × Never asking for the source

It will naturally stitch several passages together — it reads smoothly, but the original text never said that. Make it quote the original sentence for every conclusion, and spot-check two of them yourself.

### × Pasting a scanned PDF directly

An image-based PDF extracts as garbled text or nothing at all, so it's effectively seeing nothing. Run text recognition first, or just take screenshots and let it read the images.

## Checklist

☐ I state clearly what I want out of the document before deciding which part to paste

☐ For documents over twenty thousand characters, I split by chapter and ask one chunk at a time

☐ I require a quote from the original for every conclusion, and I spot-check a few myself

☐ I check numbers, dates and names against the original text

Part Four

# RAG and Agents

Chapters 22 — 33

22

# Vector search: getting AI to find answers in your own files

You have hundreds of your own files. How does the AI know which one to look at? It turns text into numbers and picks by how close the meanings are. This chapter lays out the process and the traps.

---

## What this chapter covers

☐ You understand how searching by meaning differs from searching by keyword, and which one fits which situation

☐ You know the four steps you need at minimum to build a Q&A tool over your own documents

☐ You know where it most often goes wrong, and how to make that rarer

## 01 It finds what a keyword search can't

Search your company documents for "reimbursement" and you only get the files that literally contain the word. The one titled "Expense request process" never shows up.

Vector search (turning each passage of text into a string of numbers, where passages with similar meaning get similar numbers) solves exactly this. Search "reimbursement" and it can surface "Expense request process". Not a single word matches, and it still hits.

Keyword search compares literal text. This compares meaning. But watch out: it returns the closest few passages, not necessarily the right ones. A human still has to look.

- One passage or one subsection per entry works best. Making a whole document one entry is too coarse.

- It finds the same thing under different wording, e.g. "time off" and "leave application".
- It doesn't understand negation or ranges. Ask for "a quote without tax" and it easily gets it wrong.

## 02 The whole process is four steps

Step one: cut your material into small chunks, a few hundred characters each. Step two: run each chunk through a model, turn it into that string of numbers, and store it. This step is called building the index. You do it once, and it keeps working from then on.

Step three: when you ask a question, the question gets turned into numbers too, and the system looks in the store for the closest chunks. Step four: those chunks of original text are stuffed into the model, and it answers from them.

Step four is the key. The model is answering from the original text in your files, not from what it has memorized. That's why it can tell you your own company's internal rules.

- Give each chunk a little context of its own when you cut it. For example, put the section title at the top.
- Don't retrieve too many chunks. Three to five is enough; more just dilutes the point.
- Have it list which chunks it used, so you can go back to the original text.

## 03 The two places it most often goes wrong

One: the chunking is broken. A sentence gets cut in half, or a table gets scattered, so what comes back is incomplete and the model can only guess.

Two: what comes back looks similar on the surface but means the wrong thing. The model still forces an answer out of that chunk. And it sounds smooth.

So this setup suits questions whose answers are already written down somewhere in your material. Anything that needs calculation, comparison or a judgment call is not its strength.

- Keep the material current: change a file and the index has to be rebuilt, or it keeps answering from the old version.
- Test it first with ten questions whose answers you already know. Only hand it to other people after it gets them right.
- When it can't answer, make it say "that's not in the material". That beats making something up.

## A template you can copy

Make the model answer only from the retrieved material

```
Below are several passages retrieved from my knowledge base, sorted by relevance.
[Passage 1] paste the original text
[Passage 2] paste the original text
[Passage 3] paste the original text

Answer my question using only the passages above: write your question here.
1. Tag every sentence in the answer with the passage it came from, in the format (Passage 2).
2. If these passages together still aren't enough to answer, just say "not enough material", and tell me what information is missing.
3. Don't fill in from your own knowledge, and don't speculate.
```

## Common mistakes

### × Thinking a built index is set for good

Change the material and the index is stale. Pick a rebuild rhythm, such as once a week, or every time a file changes.

### × Treating a whole document as one entry

An entry should be a few hundred characters. Make the whole document one entry and what comes back is one big lump the model can't pull a point out of.

### × Asking it questions that need reasoning

It only finds passages with similar meaning. It can't calculate and it can't compare. If you need analysis, have it find the original text first, then ask separately.

### × Believing it without asking where it came from

It stitches the retrieved passages together smoothly, and the result looks like the original text when it isn't. Make it tag passage numbers, and spot-check two or three yourself.

## Checklist

☐ I know it searches by meaning, not by literal text, so different wording still hits

☐ My material is cut into chunks of a few hundred characters, not one entry per document

☐ I make it tag which chunk each answer came from, and I spot-check it myself

☐ When files change I rebuild the index, and I don't treat old results as accurate

# 23
# What RAG is, and why every company is building one

RAG means making the AI look through your material before it answers. This chapter explains what problem it actually solves, and why companies keep building it.

## What this chapter covers

☐ You can explain in one plain sentence what RAG actually does, to a colleague

☐ You know which kinds of problems are worth RAG and which are just burning money

☐ You can spot which claims in a vendor's proposal are bluff

## 01 In plain terms: look it up first, then write

RAG stands for "retrieval-augmented generation". In plain terms, it means having the AI go through your material first, then answer from that material.

Broken down, it's two steps. Retrieval: search your own collection first. Augmentation: hand what you found to the model along with the question.

You ask a question. It pulls out the few most relevant passages, then hands them to the model together with your question and has it answer from those passages. It's the same thing a person does when writing a report. It just flips through faster than you do.

- It doesn't change the model itself. It just feeds the model material at question time.

- You can swap the model any time. The material collection is the part that's yours.
- Update the material and the answers change with it. No retraining needed.

**What one answer looks like (illustrative)**

```
User question: can a probation period be extended?

Step 1 Retrieve: search the employee handbook and find the 3 most relevant passages
Step 2 Assemble: hand the question + those 3 passages to the model together
Step 3 Generate: answer using only those 3 passages, and tag the source

Final answer: it can be extended once; see company policy for the exact number of days.
(Source: Employee Handbook • Chapter 4, Probation Management)
```

## 02 Why companies have to do this

Because a large model knows nothing about your company. Product pricing, internal processes, customer contracts — it has never learned any of it.

Sure, you can paste the material in and ask. But a handbook runs to hundreds of thousands of characters. It won't fit. And pasting it in again every time is slow and expensive.

RAG slots right into that gap: send in only the few most relevant passages. Fast, and it saves money.

- A wrong answer can be traced: you see which passage it cited and can flip straight back to it
- Permissions can be enforced: who gets to see which file can be blocked right at the retrieval step
- No retraining the model — change the material and you change the answers

## 03 When not to build RAG

If the questions only need common knowledge, or you simply don't have decent material on hand, RAG is money down the drain.

There's another common misunderstanding: using it as a calculator, or as an approval workflow. It can't do those jobs. Perfect retrieval won't help.

The test is crude: you need 20-plus documents that can be written down as text and that people keep looking things up in. Only then is it worth building.

## A template you can copy

To judge whether your case is worth it, send this to an AI and have it do the math for you

```
I need to decide whether a particular case is worth building RAG for (having the AI look through my material before it answers). Judge it from the information below. Don't give me generalities.

1. Material I have available: ___ documents, roughly ___ × 10,000 characters, updated ___
2. Roughly how many people ask this kind of question each day: ___ people, ___ times each
3. How they find answers today: ___
4. What a wrong answer roughly costs: ___

Answer three things:
- Is this case a good fit for RAG? Give a clear conclusion and the reasons.
- If it isn't, what's the cheaper option?
- If it is, what should I do first, and what should I not do yet?
```

## Common mistakes

### × Thinking RAG means it can't make things up

When retrieval finds nothing relevant, it will still invent from memory. The fix is to hard-code it in the prompt: if the material doesn't have it, say "not in the material", don't fill it in yourself.

### × Dumping a whole folder in at once

It's mixed with outdated versions, meeting notes and dead drafts, and the search results get messier the more you search. Clean the material first, then do retrieval. You can't skip that step.

### × Reading only the answer, never what it cited

With no source you can't hold anyone accountable, and you can't spot problems. Require it to tag every conclusion with which file and which passage it came from.

### × Expecting it to make the judgment for you

What it gives you is "what the material says", not "what to do about this". Whether a contract can be signed, whether a quote gets approved — a person still decides.

## Checklist

☐ I can explain in one plain sentence what RAG does, to a colleague

☐ I know how many documents I have on hand and how often they get updated

☐ I've asked myself whether there's a cheaper way that doesn't involve RAG

☐ I've made it tag sources, and I spot-check a few now and then

# 24
# Building a working knowledge base from scratch

You can build one without writing code. This chapter walks through five steps in order: clean the material, chunk it, load it, wire up the Q&A, then test it yourself.

## What this chapter covers

☐ You can get a small knowledge base that answers questions running on an ordinary computer

☐ You know what to prepare at each step, and where to look when something gets stuck

☐ You can use 10 questions to find out whether it's actually reliable

## 01 Step one: clean up the material first

Don't rush to the tools. First make a folder and put only the material you really need in it: product docs, FAQs, operating manuals.

Pick out everything duplicated, outdated or still a draft. Keep only the latest version of each file, and put the date in the filename.

Whatever time this step takes, you get it all back later. If the material is dirty, no amount of tuning downstream will save it.

- One format: convert everything to plain text or PDF, and don't mix in scans
- One file covers one topic, with a clear title
- Start with 10 to 50 documents. That's enough. Don't get greedy.

## 02 Steps two and three: chunk it, load it

Long documents have to be cut into small chunks, a few hundred characters each. Chunk too long and half of what comes back is filler. Chunk too short and a sentence gets broken apart, and the meaning snaps.

Cut along headings and paragraphs, not by brute character count. Put the file name and section name at the top of each chunk.

Then each chunk becomes a string of numbers and gets stored. That's called a vector (turning a passage of text into a string of numbers, where passages with similar meaning get similar numbers). The tool does this step automatically.

- Leave some overlap between chunks: put the last two sentences of one chunk into the next
- Handle tables separately. Don't let them get chopped up.
- Before loading, pull out 5 chunks and check whether what came out still reads properly

## 03 Steps four and five: wire up the Q&A, then test it yourself

Connect retrieval to the model. Retrieve 3 to 5 chunks first, then have the model answer only from those chunks, and require it to tag sources.

Don't rush to launch it the moment it's wired up. Write 10 questions yourself first. 5 whose answers are clearly in the material, 3 the material doesn't cover, and 2 that need two files.

Once you see how many it gets right, you know where you stand. If it forces an answer on those 3 that aren't in the material, go back and fix the prompt.

- When it can't answer, check retrieval first: did nothing come back, or did the right thing come back and get answered badly?
- Nothing retrieved means adjust chunking and keywords. Retrieved but badly answered means adjust the prompt.

- Rerun those 10 questions after every change. Don't go on a feeling that "it seems a bit better".

**Log the 10 test questions like this, and rerun them after every change**

```
1. How many days until a refund lands? (in the material)
2. How do I fix an invoice that was issued wrong? (in the material)
3. Can members get an invoice? (in the material)
4. Is customer service open on weekends? (in the material)
5. Do I pay shipping to exchange an item? (in the material)
6. Can you invoice a foreign company? (not in the material)
7. Can you pick up from my door? (not in the material)
8. Is there a student discount? (not in the material)
9. If I buy a membership and then return the item, is the membership fee refunded? (needs two files)
10. What happens if a large item arrives broken? (needs two files)

Scoring: 1 point for a correct answer, minus 2 for anything invented. Only use it once you're above 8.
```

## A template you can copy

A fixed opening that makes it answer only from the material — paste it before every question

```
You may answer only from the material provided below. You may not use knowledge from outside it.

Rules:
1. If the material doesn't cover something, answer "not in the material". Don't guess.
2. Tag every conclusion with its source, in the format (file name • section name).
3. When passages contradict each other, point out the contradiction. Don't pick one yourself.

Material:
{paste the retrieved passages here}

Question: {your question}
```

## Common mistakes

### × Trying to build the whole company's knowledge base on day one

The bigger the scope, the dirtier the material and the less likely it ever gets finished. Pick one department and 50 files, get that working end to end, then expand.

### × Copying someone else's chunk numbers

Their 800 characters may not suit you. Start at around 500 characters, pull out 5 chunks and read them, and adjust if they don't read well.

### × Blaming the model when nothing comes back

Eight times out of ten retrieval simply didn't catch anything, and the strongest model won't help. Print out the retrieved chunks and take a look first.

### × Leaving it alone once it's live

Material goes stale and answers drift. Every month, rerun 10 real questions, and patch the material wherever it gets one wrong.

## Checklist

☐ I've gone through my material, and the outdated files and drafts are out

☐ I've read 5 of the finished chunks, and the meaning wasn't cut apart

☐ I've written 10 test questions, 3 of which the material doesn't cover at all

☐ When it answers "not in the material", I know whether to add material or fix the prompt

# 25
# How Big Should Each Chunk Be?

Cut chunks too big and half of what you retrieve is filler. Cut them too small and a single sentence gets torn apart. This chapter gives you a starting number and three signals for when to adjust it.

## What this chapter covers

☐ You can set a specific number for your own material instead of going by feel

☐ You know when a bad search result means changing the chunk size, and when it means changing something else

☐ You can set different chunking rules for different kinds of material

## 01 Start With a Number You Can Actually Use

Don't get stuck on theory. For Chinese material, start with chunks of 300 to 500 characters. That range covers most manuals, policies and how-to documents.

Why that number? Around 400 characters is usually just enough to finish one small question, without dragging in three or four other topics along the way.

Leave an overlap of 50 to 100 characters between chunks. The end of one chunk shows up again at the start of the next, so a meaning never gets cut in half.

- Q&A lists: one question and one answer per chunk, under 200 characters
- Manuals and clause lists: cut along subheadings, 300 to 600 characters

- Meeting notes: cut by one person's turn or one topic, not by character count

## 02 Three Signals That You Cut It Wrong

Signal one: half of each retrieved passage has nothing to do with the question. That means the chunks are too big. They're hauling in unrelated material. Go smaller.

Signal two: what comes back is half a sentence, with nothing before or after it. That means the chunks are too small. Go bigger, and increase the overlap at the same time.

Signal three: the answer is clearly in your material but search never finds it. This is usually not about size at all. The keywords don't line up. Don't rush to change the number.

- Change one parameter at a time, then rerun the same batch of questions
- Work your way up between 200 and 800 characters. Don't jump straight to several thousand
- Write down the number and the score each time. Don't go by memory

## 03 Give Every Chunk an ID Card

The body text alone isn't enough. Put the file name, section name and update date at the top of each chunk. Search matches more easily, and answers can cite where they came from.

A counter-example. A chunk says "valid for 30 days". With no file name attached, nobody can tell whether that's the return window or a coupon.

Adding those few lines is like giving each chunk an ID card. Search gets more accurate, and when something goes wrong you can trace it back.

- Write the full file name. No "doc1" or "final version"
- State the date clearly — especially when one file exists in several versions

- Convert tables into "item | value" rows before chunking, so a row never gets split

### What one chunk looks like

```
[File] Employee Expense Reimbursement Handbook_2024.pdf
[Section] Chapter 3 Travel Reimbursement
[Updated] 2024-06
[Body]
Lodging on business trips is reimbursed in three tiers by city. Tier A is first-tier
cities, Tier B second-tier cities, Tier C everywhere else. Per-night caps for each
tier are in the appendix; anything above the cap is on you. Reimbursement requires
the original hotel invoice, and the invoice must be made out to the company name,
or Finance will send it back to be reissued.
```

## A template you can copy

Have the AI chunk a long document for you — paste this instruction

```
Below is a document. Please cut it into small chunks suitable for retrieval.
Requirements:

1. Cut along subheadings and natural paragraphs, not by hard character count.
Keep each chunk between 300 and 500 characters.
2. Put three label lines at the top of each chunk: [File], [Section],
[Updated].
3. Keep roughly 50 characters of overlap between adjacent chunks — that is,
the end of one chunk repeats at the start of the next.
4. Tables get their own chunk, converted into the form "item | value".
5. When you're done, list each chunk's character count and what it's about. I
want to check whether anything got cut in the wrong place.

Document content:
{paste it here}
```

## Common mistakes

### × Using the same chunk size for everything

Q&A lists and long clause documents have completely different rhythms. Split into at least two types: one question per chunk for Q&A, chapter-based cutting for documents.

### × Pushing chunks to several thousand characters in the name of "completeness"

The bigger the chunk, the more irrelevant material comes back with it, the easier the model gets thrown off, and the more it costs. Better to cut more chunks.

### × Never reading the chunks after cutting them

Machine-cut chunks often stop in the middle of a sentence. Pull 5 to 10 chunks and read them start to finish. If it reads awkwardly, adjust.

### × Blaming chunk size every time search misses

Some misses have nothing to do with size — the words in the question just don't line up with the material. Try having the model rewrite the question into a few keywords first, then search.

## Checklist

☐ I've set a specific number for my own material, not "about right"

☐ I use different chunking for Q&A lists and for documents

☐ I pulled 5 chunks and read them start to finish — none stop mid-sentence

☐ Every time I adjust, I rerun the same batch of questions and record the score

# 26
# Which Embedding Model to Pick

An embedding model isn't better because it costs more, and the leaderboard winner isn't necessarily right for you. Pick wrong and everything your search returns afterwards is junk. This chapter gives you a way to test it for yourself in half an hour.

## What this chapter covers

☐ You know which few things matter for your situation, and don't get swayed by leaderboard rankings

☐ You can compare models in half an hour using a dozen or so real questions

☐ You know what switching models costs, and when you don't need to switch at all

## 01 First, Get Clear on What This Step Does

Embedding (turning a piece of text into a string of numbers, so that text with similar meaning gets similar numbers) is the first step in search and Q&A. Once text becomes numbers, the computer can work out whether two sentences look alike.

Get this step wrong and everything downstream is wasted. If the model turns "reimbursement process" and "how do I claim expenses" into two very distant strings of numbers, no amount of tuning the later steps will save you.

So the first principle in choosing a model: don't ask which one is strongest. Ask whether it recognizes the language in your library. How good a model is shows up in how it performs on your content, not in other people's reviews.

- Only vectors from the same model can be compared with each other. Mixing models turns everything into noise
- Switch models and every vector you already stored has to be recomputed
- Dimensions (the length of that string of numbers) aren't better when higher. Enough is enough

## 02 Pick by Your Content, Not by Reputation

If your library is Chinese contracts and Chinese customer-service logs, prioritize models that are solid in Chinese. An English leaderboard winner running on Chinese often does worse than a model built specifically for Chinese.

If your library mixes Chinese and English — technical docs, cross-border orders — pick a multilingual model. These put Chinese and English into one number space, so a Chinese query can find English content.

If your content is heavy on code, tables and part numbers, test that kind of content separately first. General models aren't sensitive to strings like "SKU-1024" and may never surface the original text.

- For short sentences, headings and product names, a lightweight model is plenty — fast and cheap
- For long paragraphs, legal clauses and manuals, use a stronger model. The difference is obvious
- Don't start with the biggest. Get it working on a small one first, then decide whether to switch

## 03 Run Your Own Half-Hour Test

Prepare 10 to 20 questions you've actually run into, and mark the correct passage for each one. That's your exam paper.

Run the same content through two or three candidate models, and see where the correct answer lands. Ranked first counts as a hit; past fifth place is basically a miss.

Look at the ratio, not individual cases. 17 out of 20 versus 11 out of 20 is obvious at a glance. You don't need any complicated metric.

- The questions must come from real users. Don't invent textbook-style phrasings
- Only top three counts as found. Don't comfort yourself with "it's in the top ten"
- Write the results down, so when you upgrade the model later you can rerun the same questions

**Example code (swap in your actual model names)**

```
# Run the same questions through two models and see which ranks the answer higher
from openai import OpenAI
client = OpenAI()

def embed(texts, model):
r = client.embeddings.create(input=texts, model=model)
return [d.embedding for d in r.data]

questions = ["how many days until reimbursement arrives", "what if I lose the invoice", "can I take leave during probation"]
answers = ["reimbursement usually arrives within three business days", "a lost invoice requires a written explanation", "leave during probation requires one day's notice"]

for m in ["candidate_model_A", "candidate_model_B"]:
qv, av = embed(questions, m), embed(answers, m)
for i, q in enumerate(qv):
scores = [sum(a * b for a, b in zip(q, v)) for v in av]
order = sorted(range(len(scores)), key=lambda j: -scores[j])
print(m, questions[i], "-> rank", order.index(i) + 1)
```

## A template you can copy

Ask the AI this to settle your test plan and pass mark

```
I need to choose an embedding model for a batch of Chinese content (turning text into numbers so similarity can be computed).
My content is: (be specific, e.g. "300 customer-service Q&A pairs, Chinese, about 100 characters per passage")
Please:
1. List 3 candidate directions that would fit, and say what kind of content each suits. Don't just recommend the most expensive one.
2. Give me 15 test questions that sound like what real users would ask, not textbook-style.
3. Tell me how to judge which is better, with specific criteria and one pass mark.
4. Explain what I'd have to redo if I want to switch models later.
Every point should land on a concrete number. No vague advice.
```

## Common mistakes

### × Treating an English leaderboard as the standard

Leaderboards mostly test English tasks. Long Chinese sentences, industry terms, abbreviations — they simply don't measure those. The right move is to test with 20 questions from your own library. For Chinese content, only look at Chinese results.

### × Switching models midway without recomputing

Vectors from the new model aren't the same number space as the old ones. Compare them and every similarity score comes out garbage. Switching means rerunning everything in the library — not one item skipped.

### × Cutting passages too finely

One sentence per passage means the vector carries no context. What comes back looks related but is useless. Cut along complete units of meaning, 200 to 500 characters each, with a little overlap on both sides.

### × Drawing conclusions from three test cases

Three lucky hits don't prove the model works. Use at least 20, covering the most common kinds of questions in your library, then look at whether the hit ratio is high enough.

## Checklist

☐ I tested with real questions from my own library, not examples someone else handed me

☐ I know switching models means recomputing every vector, and I've set aside time for it

☐ I picked the model by content type (Chinese, multilingual, code), not by following a leaderboard

☐ I got it working on a lightweight model first, then decided whether to move to something stronger

# 27
# Reranking: Pull the Most Relevant Passages to the Top

The results vector search hauls back are often in the wrong order. Add a reranking step to push the few passages that actually answer the question to the top, and the AI stops drifting off.

## What this chapter covers

☐ You know when reranking is a must, and when adding it just burns money

☐ You can pin down the two numbers: how many to retrieve first, and how many to keep at the end

☐ You can rerank with a prompt alone — no code required

## 01 Why the Order Comes Back Wrong

Vector search (turning text into numbers and finding similar content by how close the numbers are) is a coarse filter. It's fast, but it only judges rough similarity, not whether a passage is actually useful.

The most common failure: the genuinely useful passage sits at number 8, and the 7 above it are all "on topic but not answering the question". Feed the top three straight to the AI and of course it answers off-target.

Reranking fills that gap. Widen the candidate pool to a few dozen first, then use a more careful model to judge each one — "can this actually answer the question?" — and reorder them. It doesn't change the candidates, only the order, so it's fast, cheap and easy to try.

- The coarse pass is responsible for not missing anything; reranking is responsible for not getting it wrong. Different jobs

- Too few candidates and reranking has nothing to choose from. Too many and it's slow and expensive
- Reranking only touches those few dozen candidates, so the cost stays under control

## 02 Retrieve Thirty, Keep Three

Fix both numbers at the start: how many to recall, and how many to keep. Generally, recall 30 to 50, then keep 3 to 5 after reranking for the AI to read. Below that, many genuinely relevant passages never make it into the candidate pool.

Why keep only 3 to 5? Cram in too many and the AI can't find the point. It also gets pulled off course by irrelevant passages — the longer the answer runs, the more likely it drifts.

If a question needs answers stitched together from several passages — "compare model A and model B", say — keep 6 to 8, and try to draw them from different documents. This kind of question suffers most when every candidate comes from the same document.

- Recall at least 5 times the final count, or reranking is doing nothing
- Merge adjacent passages from the same document into one before ranking
- Leave the AI an exit: "if none of these are relevant, say so". Don't force an answer

## 03 You Can Rerank Without Writing Code

When there are only a few dozen candidates, you can absolutely just have the AI read them through and sort them. That's far simpler than wiring up a model API, and for ordinary cases it works well enough.

The key is stating the criterion clearly: don't ask "which one is most similar to the question", ask "which one directly answers the question". Those are often two different things. Ask for "most similar" and what you get back is mostly paraphrases of the same passage. Ask for

"can answer" and the real answer surfaces.

Have it output only the numbers, not restate the text. You already have the text in your own library. Restating is expensive and invites altered wording.

- Don't feed more than 50 candidates at once. Beyond that it starts skipping things
- Ask for one line of reasoning per pick, so you can see why it ordered them that way
- Fetch the original text from your library by number. Don't let the AI rewrite the content

**Example prompt**

```
Below is one question and 30 candidate passages. Do only one thing: pick the 4 that best answer the question.

Criteria:
1. Passages that directly answer the question go first; ones that merely touch the topic go later.
2. Keep only one copy of duplicated content.
3. If none of them can answer, output "no relevant content".

Output format: only the numbers plus one line of reasoning each. Don't restate the original text.

Question: {the user's question}
Candidate passages:
1. {content}
2. {content}
...
```

## A template you can copy

Ask the AI this to turn your recall count and final count into concrete numbers

```
I'm building a Q&A tool: the user asks something, I search out candidate passages first, then have the AI answer.
My material is: (be specific, e.g. "200 product documents, about 300 characters per passage")
What users mostly ask: (give examples, e.g. "how do I turn this feature on", "what's the difference from that other feature")
Please decide for me:
1. How many candidates to retrieve each time, and why.
2. How many to hand the AI in the end, and why.
3. Which kinds of questions need more, and which need only one or two.
4. Give me 3 ways to tell whether reranking is actually doing anything.
Every point should land on a concrete number. No vague advice.
```

## Common mistakes

### × Retrieving three candidates and then reranking

Reranking picks from the candidates. If the right answer was never in the pool, no amount of accurate ordering helps. Widen recall to 30 or more first, then let reranking choose.

### × Asking the reranking model to restate the original text

Restating is slow and invites altered wording — key numbers get miscopied often. Have it output only the numbers and one line of reasoning, and pull the original text from your own library by number.

### × Dumping every candidate on the AI to read

Pile in a few dozen at once and it skips things, grabs the wrong emphasis, and costs more. Rerank down to 3 to 5 first, then let it read.

### × Adding reranking but never comparing

Adding it doesn't mean it helped. Keep 20 test questions and record where the correct answer ranked before and after. If there's no improvement, check whether the candidate pool was too small or the passages were cut too finely. Use the same batch of questions for the comparison — don't swap them.

## Checklist

☐ I retrieve 30 or more candidates first, then rerank down to 3 to 5

☐ I have reranking output only numbers and reasoning, never a restatement

☐ I left a "no relevant content" exit instead of forcing the AI to answer

☐ I compared before and after reranking using the same batch of test questions

# 28
# Agents: Letting the AI Do the Work Itself

From "you ask one question, it answers one question" to "you state the goal, it runs on its own." This chapter covers how it actually works, how to brief it, and how to keep it from making a mess of things.

## What this chapter covers

☐ You know when a job should go to an agent to run on its own, and when it is steadier to split it into steps yourself

☐ You can spell out the three things clearly: the goal, the tools, and the stop condition

☐ You know which actions always need a human confirmation and must never be let go

## 01 How it differs from the AI you normally use

Normally you ask one question and it answers one question, and every step in between is up to you. An agent (letting the AI decide the next step itself and call tools to finish the job) takes those middle steps off your hands.

It reads the situation as it goes. It searches first, notices something is off, and tries a different search. It writes code, runs it, sees the error, and fixes it. You state the goal at the start and look at the result at the end.

The price is that it makes its own calls. If your brief is vague, it runs with its own interpretation. By the time you notice it has drifted, it may be dozens of steps in. So the briefing matters more than which model you pick.

- Its ceiling is not just how smart the model is. It is mostly what tools you can hand it.
- The more tools it has, the more likely it is to misuse them. Start with two or three that are enough.
- Every extra tool is one more way for it to go sideways.
- Cap how many steps it can run in a row, or it will circle in place.

## 02 Say it as three things: the goal, the tools, where to stop

Write the goal so it can be checked. "Help me organize the customer feedback" will not do. "Sort the 500 feedback items into 5 categories by problem type, and write one summary sentence for each" will.

Say plainly what it can touch and what it cannot. It can read files but not delete them. It can query the database but only read, not change it. Any permission you do not spell out, it assumes it has.

Where to stop is the thing people forget most easily. Telling it "try at most 5 times" or "if you cannot find it, stop and tell me" is far more reliable than letting it judge for itself when it is done.

- Put counts and formats in the goal, so it knows when the job counts as finished
- Write "if you cannot do it, stop and explain why." That one line saves a lot of rework.
- Have it leave a one-line record at each step, so when something breaks you can see which step went crooked

## 03 What you can let go, and what needs a gate

What you can let go: looking things up, tidying formats, writing drafts, running tests, editing code in local files. If any of these go wrong, you spot it at a glance, and fixing it is cheap. It does this kind of work faster than a person, and with more patience.

What needs a gate: sending email, placing orders and paying, deleting files, changing production data, publishing to the outside world. Once

one of these goes wrong, you cannot take it back.

The method is simple. For these actions, have it write out what it is about to do first, and only execute after you click confirm. It costs you ten seconds and saves you an incident.

- Try it on a small scope first, 12 files say, and only run it on 500 once that works
- If the same wrong approach fails 3 times, it should stop and ask a human, not keep trying

**Example task brief (you can use it as is)**

```
Goal: merge the 12 csv files in D:\报告 into one table, sort by "date", output
to 合并.xlsx.

You may use: read files, write files, run python.
You may not: delete any original file, or go online.

How:
1. First list the headers of all 12 files and confirm the column names match.
If they do not match, stop and tell me.
2. After merging, report the total row count and whether there are any empty
values.
3. After each step, write one log line saying what you did.

Stop condition: if the same error fails 3 times, stop and explain where you are
stuck. Do not switch approaches on your own.
```

## A template you can copy

When you are not sure whether to hand it to an agent, ask the AI this first

```
I want the AI to do something on its own: (write clearly what you want done)
Help me judge:
1. Is this suited to the AI running it step by step itself, or is it steadier if I split it into steps? Give reasons.
2. If I hand it over, which tools does it need? Which tools must it absolutely not have?
3. Write me a task brief that includes the goal, the available tools, the forbidden actions, and the stop condition.
4. List the 3 places this is most likely to go wrong, and how I should block them in advance.
Requirement: write the goal so it can be checked, with counts and output format.
```

## Common mistakes

### × The goal is too vague

"Help me optimize this" — it will define what optimizing means on its own, then hand you something you do not want. Switch to a checkable version: give the counts, the format, and the pass-or-fail criteria.

### × Handing it a pile of tools right away

The more tools there are, the higher the chance it picks wrong, and the harder it is to trace when something breaks. Start with the two or three that are necessary. Get one running steadily, then add the second.

### × No stop condition

It will retry the same wrong approach and burn through dozens of steps of quota. Write "stop after 3 tries at the same error," and require it to say where it is stuck when it stops — not quietly switch approaches and keep going.

### × Letting it touch production data or publish things directly

Deleted is deleted. Sent is sent. For any action like this, change it so it writes out what it is about to do first, waits for a human to confirm, and then executes.

## Checklist

☐ My goal has counts, a format and pass-or-fail criteria, not just "optimize it"

☐ I only gave it the tools it needs, and no permission to delete things or message the outside world

☐ I wrote a stop condition: if it fails a few tries, stop and tell me

☐ Anything involving payment, deletion or external publishing keeps a human confirmation step

# 29
# Getting the AI to Use Tools

Talk alone only gets you guesses. Connect it to search, calculation and code execution, and the answers go from "sounds right" to "actually right."

## What this chapter covers

☐ You know which kinds of work it has to do by hand rather than answer from memory

☐ You can hand it a tool in one sentence, and say clearly when it must use it

☐ You can check whether it actually used the tool, instead of being waved off with "I looked it up"

## 01 Out of the box it has a mouth, not hands

A large language model (the AI you normally chat with) ships able to do one thing: continue what you said. It cannot go online, cannot open your files, and cannot actually use a calculator.

So when you say "look up tomorrow's weather in Shanghai for me," it does not go look. It writes a passage shaped like a weather answer. The format is perfect. The numbers are all wrong.

This is not laziness. It simply has no hands. You have to hand it a tool before it knows it can act.

- It can search the web: check the news, check prices, check whether a document really contains that sentence
- It can run code: do arithmetic, process spreadsheets, batch-rename files
- It can read files: your uploaded PDFs, Excel sheets and screenshots only become visible to it this way

## 02 Hand over the tool, then say when it must be used

Flipping the switch on is not enough. You have to write into your prompt which situations require it to go look things up, and forbid it from answering from memory there.

For example, "search for the latest material before you answer" is too soft. It may do a token search and move on. Change it to "anything involving prices, dates or model numbers: search first, and if you cannot find it, say you could not find it." Then it will actually search.

You do not have to memorize what the tools are called or how they are invoked. Just say it in plain language: have it look it up, have it calculate, have it run some code.

- A hard rule of "when unsure, look it up" does far more than "be as accurate as possible"
- Have it paste the sources it found, so you can see at a glance whether it really searched
- For arithmetic, just say "write code to compute it, do not do it in your head"

**Example prompt**

> You can now search the web. Follow three rules:
> 1. Anything involving prices, dates, model numbers or policy clauses: search first, and do not answer from memory.
> 2. Put the source link after each conclusion.
> 3. If you cannot find it, write "no reliable source found." Do not guess and fill something in.

## 03 How to tell whether it really used the tool or is faking it

The easiest way is to make it write out its process: which terms it searched, which page it opened, what the intermediate result was.

The second way is to give it a question it can only get wrong without looking. Ask for the version number a small piece of software released

yesterday. If it answers off the top of its head, it did not look.

The third is to check whether the results line up. Numbers that do not match, links that do not open, dates from last year — that is basically made up.

- Have it list "how I plan to look this up" first, and let it start only after you nod
- Important conclusions need two independent sources. Do not trust a single one.
- It saying "I checked" is not evidence. The source link is.

## A template you can copy

Copy this so it states its plan before it acts

```
For this task you can use tools (search, run code, read the files I upload).
Before you start, state in three lines:
1. Which tool you plan to use, what you will look up, and how.
2. Which steps need no tool, where you can just write.
3. Which step you are most likely to get wrong.
Start only after I reply "go ahead."
```

## Common mistakes

### × Assuming it is online automatically

Most chat windows are offline by default, and the switch is buried in settings. When it is offline it still answers in a very convincing way, and you cannot tell. Before you use it, confirm the switch is on, or just ask it "can you go online right now?"

### × Treating "look it up" as a command

Too vague. It may just restate what is in its memory. Say what to look up, how much counts as enough, and what to do if nothing turns up.

### × Letting it do money math and date math

It predicts text. Multiply two decimals and it slips easily. For amounts, schedules, day counts, have it write code to compute, and print the

intermediate results for you to see.

### × Tool given, permission not

Search connected but no key configured, files uploaded but file reading not enabled — it can only sit and stare. When it will not run, check whether the tool is connected first. Do not rush to rewrite the prompt.

## Checklist

☐ I know it has no hands by default, and for anything real-time I confirm the online switch is on

☐ I write "in which situations it must look it up" into my prompt, not just a vague "be more accurate"

☐ I make it paste source links instead of taking "I checked" at face value

☐ For money, dates and schedules I make it write code and run it, not do mental math

# 30
# Breaking Complex Tasks into Steps

Hand it one big thing and what comes back usually is not usable. Have it split the work into small steps, check each one, and the quality of the finished piece changes immediately.

## What this chapter covers

☐ You can turn a big phrase like "put together a plan" into five or six steps it can actually do

☐ You know how much information to give it per step, without stuffing it all in at once

☐ You can check in between steps, catch drift early, and avoid rewriting the whole thing

## 01 Why you cannot throw a big job at it in one go

Ask it to "write a complete quarterly review" and it can only guess what you want. Guess wrong and you get a very tidy document that is useless.

There is another problem: ask for too much at once and it goes slack toward the end. The early parts are detailed, the later parts go fuzzy, and the last paragraph is usually boilerplate.

Split it up and it is different. Each step is small enough that it can do it carefully and you can see at a glance whether it is right. If it is wrong, you fix a small piece.

- One step, one job: either set the skeleton, or gather material, or write the body
- You should be able to understand each step's output. If you cannot, it is not split finely enough.

- Keep one step's work under half a page. Over that, split it again.

## 02 Have it do the splitting, do not sit there thinking it up yourself

A lot of people grind away trying to come up with the steps themselves. This job can go to the AI, and it often splits more finely than you would.

Just say: how many steps is this job, what does each step produce, what do I need to give you first. Once it has listed them, you pick and reorder — much faster than starting from nothing. For example, if step three in its plan is "gather material," then throw it two old reviews up front instead of waiting for it to make something up.

One detail: have it mark "the places where you need to make the call" separately. That way you know which steps to watch.

- Have it flag which step is most likely to need rework, and why
- Keep it to 6 steps or fewer. More than that, merge them, or you will not remember them yourself.
- Do not run it right after the split. Reorder it once yourself, then start.

**Example prompt**

> What I need to do is: <write the task clearly>.
> Do not start writing yet. Please:
> 1. Split it into no more than 6 steps, one sentence each saying what the output is.
> 2. Mark which steps need me to supply material or make a decision first.
> 3. Mark which step is most likely to go off course, and why.
> Wait for me to confirm the order, then start step one.

## 03 Check each step as it lands, do not wait until the end

When a step is done, spend two minutes scanning it: is the direction right, is any key information missing. If it is off, fix it on the spot. Do not let it pile up. When you ask for a fix, say which sentence to

change, not just "optimize it again."

Three sentences are enough for a check: what is good about this step, what is wrong, what changes next. No long review needed. For example: "The skeleton works, but paragraph three needs data, so add two tables next." That is enough.

One more trick: have it close each step with three lines on "what this step settled." Put them together at the end and you have the skeleton of the whole thing. It is a big time-saver when you look back, and you can read it straight off when you report to your boss.

- Never move on before the first step's skeleton is settled
- Save each step's output separately, so you can roll back if you break it
- If the same step drifts twice or more, that step is still too big

## A template you can copy

When you are stuck, use this to make it split one step finer

```
I keep feeling this step is too big, and what comes out is wrong. Please split
it into 3 smaller moves, and for each one write clearly:
1. What input I give it
2. What it outputs
3. How to judge whether it did it right
Show me after splitting, and go on once I say it is fine.
```

## Common mistakes

### × Writing out every requirement in one go

A two-thousand-word prompt and it actually loses the thread. Give the goal and a few hard constraints first, and save the details for the step where they matter. Feed information in batches, and it holds on to it better.

### × Running all the way through without checking

If the step order is wrong, everything after it is wasted. You have to sign off on the skeleton step. Do not let it charge ahead on its own.

### × Asking for a full rewrite to change one section

The good parts up front get broken along the way. Say clearly: "change only this section, keep everything else as it is, do not touch it."

### × Splitting too finely

Twenty steps and the back-and-forth alone wears you out, plus the parts easily stop agreeing with each other. Three to six steps is about right: one step, one output you can check.

## Checklist

☐ I split the big task into no more than 6 steps, each with a small output I can read

☐ I confirmed the first step's skeleton before letting it go on

☐ I spend two minutes checking after each step, instead of looking at everything at the end

☐ When I ask for a local fix, I add "keep the rest as it is"

# 31
# Making the AI Remember Earlier Work

It has no long-term memory, and one conversation can't hold much either. Learn to build a background card and stage summaries, and you can pick up the next day without losing the thread.

## What this chapter covers

☐ Understand why it forgets, and at which step it forgets

☐ Can build a background card for a project and get a new conversation up to speed in ten seconds

☐ Can turn the conclusions you talk out into a summary and feed it straight back next time

## 01 It didn't forget — it never stored it

Every time you open a new conversation, it is meeting you for the first time. Last time you talked for three hours, and it doesn't remember a single word.

There's a ceiling inside one conversation too. Talk long enough and the earliest rounds get pushed out. It only remembers the second half, and it answers with full confidence.

That ceiling is the context window (the maximum amount of text it can hold in view at once). Once it's full, the old material gets shoved out. And it won't warn you with "I've forgotten that part."

- The earlier something sits, the more easily it gets pushed out. Don't put the important stuff only at the beginning
- Don't do several unrelated things in one conversation. Mixed together, they interfere with each other
- When it says "you mentioned earlier that...", stay alert — it may have mixed things up

## 02 Make a background card and get up to speed in ten seconds

A background card is a small file in a fixed format. It records who you are, what you're doing, and what rules to follow. Every time you open a new conversation, paste it in as the first message. Don't think of it as a hassle — those ten seconds save you half an hour of re-explaining.

Keep only what it will actually use: what the project is, who the readers are, the style requirements, the decisions already made, the mistakes not to repeat.

Don't write a long autobiography. One screen is about right. Past one screen it loses the thread instead. A fixed format works better, because each time you only change a few lines in it.

- Writing "what's already decided" is more useful than writing "what I want"
- Put the pitfalls you've hit into it, and it will make half as many repeat mistakes
- When the project changes, go back and edit this card. Don't just add a verbal note inside the conversation

**Background card template**

> 【Project】Write an onboarding guide for new hires, for internal company use
> 【Readers】Fresh graduates who just joined, with no understanding of the business
> 【Style】Short sentences, plenty of examples, no jargon
> 【Decided】Split into three stages: "week one / month one / month three"
> 【Don't】No company history. Keep the whole thing under 3000 words
> 【Last pitfall】It likes to invent process names on its own — always ask me first about anything to do with processes

## 03 Save what you talk out

The most valuable thing in a long conversation usually isn't the final product. It's the decisions you made along the way. Those have to

land in a file. Things like ″write the style like this version″ or ″only use last year's data″ — once it's decided, write it down.

The simplest approach: at each stage, have it output a ″stage summary″, and copy that into a notepad or a document.

Next time you pick the work up, paste in the summary together with the background card. It can pick up from there. You don't have to explain from the beginning, and you don't have to worry about it getting things wrong.

- Keep only conclusions and reasons in the summary. Delete the process
- One file per task. Don't pile everything into the same document
- Put the date in the file name, so you can find it when you need to roll back

## A template you can copy

Halfway through a conversation, use this to have it produce a summary you can carry away

```
Pause for a moment and turn the progress so far into a summary, in this format:
1. Things already settled (list them item by item, one sentence each)
2. Things not settled yet that need my decision
3. Problems you hit, or that I corrected
4. What to do next
Only conclusions. Don't retell the process. Keep it under 300 words.
```

## Common mistakes

### × Using one conversation until the end of time

After a few dozen rounds it starts forgetting the earlier text, and it easily stirs different topics together. One conversation per task. When you're done, open a new one and carry the summary over.

### × Burying key information in the middle of a long conversation

Middle content gets pushed out most easily. Important constraints either go into the background card and get pasted again and again, or get restated each time you ask a question.

### × Expecting it to remember your preferences by itself

Some products have a memory feature, but what it remembers might be a joke you made in passing. Write important preferences into the background card. Don't gamble on it remembering correctly.

### × Writing the summary as a running log

Copying the whole conversation down is useless — pasting it in next time still takes up space. Keep only conclusions, reasons and to-dos. Delete the process, all of it.

## Checklist

☐ I know that when I open a new conversation it remembers nothing, so the first message is the background card

☐ One conversation per task. When I'm done I save a summary, and I don't do three things in one window

☐ Important constraints I write firmly into the background card, instead of relying on it to remember

☐ In each stage summary I keep only conclusions and to-dos, not the process

# 32
# MCP: One Standard Socket for the AI

When the AI wants to read your files or query your database, in the old days every tool had to be wired up separately. MCP is one standard socket. Connect it once, and it still works when you switch clients.

## What this chapter covers

☐ Understand the concrete problem MCP solves: the AI can't reach anything on your computer

☐ Can read an MCP config and say what each line does

☐ Can connect a file or database tool yourself, and verify it in order

## 01 In the old days, every new tool meant rewriting the wiring

You ask the AI to read a spreadsheet on your computer. It can't. You ask it to query the company database. It can't do that either. All it can see is the bit of text you paste into the chat box.

The old approach was to write a separate integration for every tool and every client. Five tools and three clients meant writing fifteen integrations. One change on one side and everyone else has to change along with it.

MCP is a convention. The full name is Model Context Protocol. Its job is to get the AI and outside tools speaking the same language. With it, the tool side does the work once and anyone can connect. Those fifteen integrations compress into one.

- It covers how the AI gets at things outside itself. It says nothing about whether the AI is smart

- The most direct change for you: installing a tool goes from writing code to filling in a few lines of config
- The same tool, used in an editor today and a different client tomorrow — the config is basically copied over

## 02 A config is just three parts: what it's called, how to start it, what parameters to pass

An MCP config, when you get down to it, answers only three questions. What is this tool called, which command starts it up, and what parameters get handed over at startup.

The snippet below is a typical way to write a local file tool. It's a small program running on your computer. The client is responsible for launching it, then asking it "what can you do?"

Name it whatever you like, as long as you recognize it yourself. The directory in the parameters is the scope it's allowed to touch. Whatever directory you write there, that's the only directory it can see.

**Example config: put it in the client's config file**

```
{
"mcpServers": {
"my-files": {
"command": "npx",
"args": ["-y", "@some-org/files-mcp"],
"env": { "ROOT_DIR": "D:/docs" }
}
}
}
```

## 03 Don't rush to use it after installing — verify in three steps

Step one: restart the client and open the tool list in settings. If you can see the name you gave it, the handshake worked.

Step two: have it do something small, like "list the file names in D:/docs". Only when it reports real file names has it actually

connected.

Step three is when you let it read content and write content. Verify one step at a time, so when something goes wrong you know which layer it's stuck at instead of guessing blindly.

- If it doesn't show up in the list, the command is probably written wrong, or the program isn't installed at all
- If it connects but throws an error the moment you use it, it's usually permissions: the directory isn't authorized, or the key isn't filled in
- Before you let it touch files, run it once in read-only mode

## A template you can copy

The first time you connect a new tool, use this to have it report its own status

> I just connected an MCP tool called <tool name>. Do three things in order, and stop after each one to tell me the result:
> 1. List the tools you can call right now. Give only the name and one sentence on what it does. Don't expand.
> 2. Use the read-only one among them to read <specific directory or table name>, and tell me what you actually saw.
> 3. If step 2 fails, paste the error text exactly as it is, and judge whether it's a path problem, a permission problem, or a parameter problem.

## Common mistakes

### × Thinking that installing MCP gets it onto the internet

MCP connects tools on your local machine or your internal network, not the internet. To look up live information you have to connect a search-type tool separately. Don't mix the two up.

### × Installing a dozen tools in one go

The more tools there are, the higher the chance the AI picks the wrong one, and the more context it burns (the amount of text it can read in at once). Install one first. Once it runs smoothly, add a second.

### × Handing over write permission right away

If it can read, it can change things, and a bad change is hard to roll back. On the first connection, give read permission on one test directory only. Once that runs clean, open up write.

### × Putting the key in the config and committing it to git

Config files often get committed along with everything else. Keys go in environment variables, or in a separate file you don't commit. Don't hard-code them just to save trouble.

## Checklist

☐ I know MCP only brings tools in; it doesn't make the model itself smarter

☐ My config has no plaintext key, and it hasn't been committed to a public repo

☐ For a newly connected tool, I give read-only permission first, and only open up write after it verifies

☐ When something goes wrong, I can tell whether it's not connected, no permission, or a parameter written wrong

# 33
# Chaining Several AIs into One Pipeline

When one prompt can't finish the job, split it into three stages handed to three roles, and save the intermediate results as files. Errors can be located, and fixing them doesn't mean starting over.

## What this chapter covers

☐ Can break a big job into 3 to 5 small steps that can each be checked on their own

☐ Can pin down the output format of each step so the next step picks up directly

☐ Know which stages must keep a human check, and which can safely run automatically

## 01 A pipeline just means cutting a big job into small pieces

Ask the AI to write an entire industry research report in one pass, and the result usually goes like this: it looks respectable, but the data is fake and the structure is a mess.

You want to fix one spot, it rewrites the whole piece, and after the rewrite the front half is a mess again. The bigger the job, the less reliable doing it in one go is. Split it up and it gets manageable: step one only gathers material, step two only picks the key points, step three only writes the draft.

Each step does one thing. If it botches the job you can see it at a glance, and you just send that step back to run again. Three stages is the most common length. Past five stages you can't keep track of it yourself. Run it as three stages first, and split it finer once it runs smoothly.

- The input and output of each stage have to be written down, not left living inside the conversation
- Give each stage a single role. Don't let the same step both hunt for material and draw conclusions
- Each stage can be checked on its own at the end. If it doesn't pass, send that stage back to run again

## 02 Intermediate results have to be in a format the next stage can eat directly

The place a pipeline breaks most easily isn't a weak model. It's that the next stage can't read what the previous stage produced.

So every step has to have its format pinned down. For example, step one always outputs "fact + source", and step two picks from that format going forward.

Pinning the format has another benefit. You can open the intermediate file at any time and see which step it's on and which piece of data is wrong. No need to rerun the whole line.

- Save each step's output in its own file, with a number in the name, like 01_material, 02_keypoints
- Use fixed subheadings and fixed separators in the format. Don't write vague requirements like "roughly like this"
- Add one line at the top of the prompt: "Output only the format below, no explanation." It saves half the rework

**Three-stage prompt skeleton — change the words and it's ready to use**

```
# Step 1: only gather material, no conclusions
Input: <source material>
Output format (follow strictly, add nothing else):
## Facts
- one-sentence fact | source
## Doubtful
- one sentence | why it's doubtful

# Step 2: only pick the key points
From the "Facts" above, pick the 5 that best support the conclusion, sorted by importance.
After each one add a sentence on why it matters. Not a full sentence — short phrases only.

# Step 3: only write the draft
Use these 5 to write a 600-word short piece. Each one must have a matching sentence in the text.
No new facts allowed. When you're done, list separately which 5 you used.
```

## 03 Where a person has to stay in the loop

Not every step is worth running automatically. Anywhere a mistake is expensive to fix, put a human checkpoint in.

There are three common checkpoints. When material comes in, check whether it's true. When conclusions come out, check whether they hold up. For example, the AI says a number in the material is from last year, you check and it's from the year before — that kind of error only a person can catch.

The third is before the draft goes out: check whether the tone is right. A checkpoint isn't waste. Two minutes of skimming saves you far more than letting an error roll into the final draft and reworking it there.

- Numbers, quotations and legal clauses always get checked by a human before moving on

- For anything published externally, the last gate has to be a person signing off
- Once it runs smoothly, remove checkpoints gradually. Don't go fully automatic from day one

## A template you can copy

The full wording for hooking the three steps together and running it once

> We'll do this in three stages. After each stage, wait for my confirmation before moving to the next.
>
> [Stage 1 • Gather material] Do one thing only: pull facts out of the material below, one per line, in the format "fact | source". If there's no source, write "no source". Don't add anything that isn't in the material. Material: <paste>
>
> [Stage 2 • Pick the key points] Do one thing only: from the facts above, pick the 5 that best support the conclusion, sorted by importance, with one sentence after each on why it matters.
>
> [Stage 3 • Draft] Do one thing only: use these 5 to write a 600-word short piece. No new facts. When you're done, list separately which 5 you used.

## Common mistakes

### × One prompt doing the whole job

When something goes wrong you can't tell whether it found the wrong material, picked the wrong points, or wrote it badly. You can only redo the whole thing. Split it into stages, and whichever stage is bad gets sent back.

### × Passing intermediate results only through the conversation

Talk long enough and it forgets, and the material you worked hard to find is gone. At the end of each stage, save the result to its own file. The next step reads from the file.

### × Never agreeing on the format

If the previous stage writes prose, the next stage can't process it item by item. Use fixed subheadings and fixed separators. Don't write "roughly like this is fine".

### × Going fully automatic from the start

Letting it run overnight before the first few steps are stable means finding nothing but waste the next morning. Run it manually three times first. Once every step is stable, chain them into one line.

## Checklist

☐ I can say how many steps this pipeline has, and what each step's input and output are

☐ Each step's output is saved as a file. It doesn't depend on conversation memory

☐ I know the three places where I have to look myself

☐ When one step goes wrong, I only rerun that step. I don't start from the beginning

Part Five

# Deployment and Tuning

Chapters 34 — 39

# 34
# Running a Model on Your Own Computer

No internet, no cost, and your material never leaves the building. The price is a smaller, slower model. First check whether your computer can carry it, then decide whether it's worth the hassle.

---

## What this chapter covers

☐ You can judge whether your own computer can handle it, and whether it's worth the trouble

☐ You can get a local model installed and talk to it from the command line

☐ You know what a small local model is good for, and which jobs not to hand it

## 01 First, be clear: why do you want to run it locally?

Three reasons hold up. The material can't leave the building, like medical records, contracts, customer lists. The environment can't get online. You use it heavily over the long run and want to save some money.

Three reasons don't hold up: you heard it's cool; you don't actually have any sensitive material; you're hoping it will be smarter than the cloud.

On the same hardware, a model you can run locally will always be smaller than what the cloud runs. It's dumber and it's slower. That's the price you pay, not a setting you got wrong. The biggest benefit of running locally is that your material doesn't leave the building, and that matters more than saving money.

- If your material is confidential or private, running locally is a hard requirement
- For everyday writing and editing copy, the cloud is the better deal
- If you expect a small local model to match the strongest cloud tier, you'll be disappointed

## 02 Two numbers on your computer: memory and VRAM

To run, a model first has to fit entirely into memory. If it doesn't fit, it won't run, or it'll be so slow it's unusable. There's no way around this step. Roughly speaking, however big the model file is, you need about that much free memory.

VRAM (the graphics card's own memory) is much faster than system memory. If you have a dedicated graphics card, use it first. If you don't, system memory has to carry the load, and the speed differs by several times.

The trick for saving space is called quantization (squeezing down the numeric precision inside the model, which can make the file less than half the size). You lose a little intelligence. For everyday use it's completely enough.

- Open Task Manager and check your memory and VRAM first, then decide what size model to pick
- For the same tier of model, the quantized version is smaller and faster — try that one first
- If it doesn't fit, step down a tier. Don't force it. Freezing up is more annoying than being slow

## 03 Getting it installed is really just three steps

Step one, install some software for running models locally. These tools handle the annoying parts for you, like graphics drivers and model formats, so you don't have to compile anything yourself.

Step two, pick a model from the list with a small number like 7b or 8b. The b stands for billion parameters; the smaller the number, the

fewer resources it needs. File names often carry tags like q4 or q8 — that's the quantization level, and the smaller the number, the smaller the file.

Step three, start it from the command line and ask it something. If the answers are too slow, step down a tier. If they're too poor, try stepping up a tier.

- Get the whole flow working with a small model first, then switch to a bigger one. Don't download the biggest one right away
- Model files are usually several GB. Download when your connection is good, not on a phone hotspot
- Close other memory-hungry programs before you run it — browser tabs count

**Once the software is installed, run these from the command line**

```
ollama list # see which models you've already downloaded
ollama run <model-name> # start a conversation; pick the model name from the software's official list
ollama ps # check whether the model is really running and how much VRAM it's using
```

## A template you can copy

Small local models tend to ramble — use this opening to keep it in line

```
You are a small model running locally. The shorter your output, the better.
Rules:
1. Give the conclusion first, one sentence, no more than 30 characters.
2. Then give your reasons, at most 3, one line each.
3. If you're not sure, just write "I'm not sure" — don't add a paragraph of explanation.
4. No pleasantries, don't repeat my question, don't summarize.
Now wait for my question.
```

## Common mistakes

### × Using it to look up the latest news

A local model's knowledge also stops at its training day, and it won't go online by itself. For new information, you have to paste the material in yourself.

### × Throwing a whole long document at it

The amount of text a local model can read at once is usually smaller than the cloud's. Cram in too much and it misses things. Cut it into small chunks first, feed them in one at a time, and ask only one thing per chunk.

### × Installing the model but ignoring cooling

A laptop at full load for a long time will throttle, and get slower and slower. Prop up the body, plug in the power, then run it. Don't push it on battery.

### × Thinking it's set-and-forget once it's installed

Switching models or changing parameters means restarting the service. After a change, run ollama ps first to confirm the new model is really loaded. Don't waste half a day testing against the old one.

## Checklist

☐ I can say why I'm running locally, and it isn't just because it's new

☐ I've checked my memory and VRAM, and the model I picked fits

☐ I feed long documents in chunks instead of stuffing the whole thing in

☐ For facts and numbers, I still check them myself. I don't relax just because it's local

# 35
# Deploying Inside the Company: Data That Doesn't Leave the Building

Some data can't go out to the internet. This chapter lays out what local deployment actually takes, how much it costs, which companies really need it, and how an ordinary person can start trying it.

## What this chapter covers

☐ You can judge whether the data in front of you can go outside at all

☐ You know what local deployment requires, and how much money and time each piece takes

☐ You can get a small model running on an ordinary computer to try out the results

## 01 Start with one question: can this data leave the building at all?

Local deployment means the model runs on your own computer or your own company's server, without connecting to the internet. Not a single byte of what you type goes to anyone else.

Whether to do this depends on what the data is. Public product copy, material from the web, practice code you wrote yourself — an online service is perfectly fine for all of that, and it's cheaper and faster.

But customer lists, ID numbers, the original text of contracts, financial data that hasn't been published yet — none of that can leave the building. Once it's pasted onto the internet, it's an incident, not a risk.

- Sort everything into three tiers first: publishable, internal use, and absolutely never leaves
- For the publishable tier, don't deploy locally — it's a waste of money
- For the never-leaves tier, don't paste it out even if the company has no rule against it

## 02 What it actually takes to run locally

Three things: a machine with a graphics card, software that can load models, and an interface you can chat with. With all three, you're in business.

The graphics card is the key. A model is really just a pile of numbers. To run it, you have to fit all of it into VRAM (that dedicated memory on the graphics card, much faster than ordinary memory). If VRAM isn't enough, the model won't run, or it'll be too slow to use.

Don't be afraid of the software part. There are ready-made tools now: install one, download a model file, click once and it starts. You don't write any code the whole way through.

- An ordinary office laptop: can only run very small models, mediocre results, fine for testing the water
- A desktop with a dedicated graphics card: can run mid-sized models, enough for writing copy and editing code
- A company server with professional graphics cards: can run large models, but needs someone to maintain it and someone to manage permissions

## 03 Try it on one machine first — don't start a project on day one

A lot of people get the first step wrong: hold meetings, write a proposal, request a budget, and three months later not a single line has run.

The right way is the opposite. Find an idle machine with a graphics card, install the tool, download a smaller model, and spend half a day on your real work with it.

When you're done you'll know two things: whether the quality is good enough, and how big a machine you actually need. Those two answers are worth more than any proposal.

- Try it on real tasks, not on questions like "hello, introduce yourself"
- Try at least 20 real tasks before you can tell whether it's reliable
- If the quality isn't enough, move up one size of model — don't rush to change hardware

**Example: three commands to run a model on your own computer**

```
# Run a model on a local machine (Ollama needs to be installed first)
# 1. Download and install Ollama; once installed it keeps running in the
background

# 2. Pull a mid-sized model down (the first time downloads several GB)
ollama pull qwen2.5:7b

# 3. Start chatting
ollama run qwen2.5:7b

# 4. You can also call it through the API, which makes it easy to wire into
your own scripts
curl http://localhost:11434/api/generate -d '{
"model": "qwen2.5:7b",
"prompt": "Make this sentence more formal: I don't think this plan works",
"stream": false
}'
```

## A template you can copy

Take this to IT or your boss and get everything asked in one go

> We want to run an AI model inside the company, to handle data that can't go outside. Please help confirm a few things:
> 1. Does the company have any idle machines with a dedicated graphics card right now? What's the card model and how much VRAM does it have?
> 2. Can that machine install new software and download model files (some company networks block this)?
> 3. Who's responsible for day-to-day maintenance and restarts on that machine?
> 4. If more than just me will use it, how are accounts and permissions divided?
> 5. Is there any rule about which data absolutely must not go into this system?
> Give me one machine to try for two weeks first. I'll run my real tasks through it, then report how many resources we need.

## Common mistakes

### × Thinking local deployment is free

Machines cost money, electricity costs money, and the person maintaining it costs money. A machine that can run mid-sized models, plus a year of maintenance, costs far more than buying an online service. It's only worth doing if the data can't leave the building — otherwise you're just losing money.

### × Buying the machine before finding out the model won't install

Company networks often block downloads, and graphics drivers can clash with the system too. Get the whole flow working on one machine first, then talk about procurement.

### × Testing with the smallest model, then declaring it doesn't work

A model that small really does perform poorly, but that doesn't prove local deployment is a dead end. Test with at least a mid-sized model, or your conclusion will be wrong.

### × Thinking local deployment is absolutely safe

The model is on your machine, but the chat logs, exported files and screenshots are still in your hands. The permissions you should manage and the temp files you should delete are still just as necessary.

## Checklist

☐ I've confirmed which tier the data in front of me falls into, and whether it can go outside

☐ I know this machine's graphics card model and how much VRAM it has

☐ I've tried at least 20 real tasks, not just chatted for a couple of lines

☐ I know who maintains this machine and who to call when something breaks

# 36
# Making the Model Run Faster

Same model, and one person waits ten seconds while another waits two. This chapter explains where the slowness comes from, what you can tune yourself, and what you can only fix by spending money on a machine.

## What this chapter covers

☐ You can tell whether a slow answer is stuck in the queue, stuck generating, or stuck on your own side

☐ You can use a few no-code changes to cut the waiting time in half

☐ You know when to switch models and when to switch machines

## 01 First work out which part is slow

Waiting for an answer, the time goes to three places: waiting for it to start talking, it dropping out one character at a time, and your own side processing the result.

The wait before it starts is called time to first token (the blank stretch between hitting enter and the first character appearing on screen). This stretch depends heavily on how long your question is.

Once it starts producing tokens, the speed is basically fixed. So to go faster, you mainly rely on two things: make it say less useless stuff, and make it start sooner.

- Even "hello" takes forever: the problem is the queue or the network
- Fast at the start, slower and slower as it goes: normal, that's just how it works
- It stutters when you copy and paste the result: nothing to do with the model, that's your computer or the web page

## 02 Twice as fast without touching code

The most common waste is letting it write a pile of stuff you don't want at all. "Please explain in detail", "please cover ten aspects" — every one of those characters is generated on the spot, and every character costs time.

The second waste is giving too much background. Paste in a three-thousand-word document and it has to read the whole thing from the start every single time.

The third waste is making it think before it answers. Some models output a long stretch of reasoning by default. You don't read that part, but the time is spent all the same.

- Saying plainly "keep it under 100 words" is much faster than deleting afterwards
- For a long document, pull out the three key paragraphs yourself first — don't throw the whole thing in
- Turn off "deep thinking" if you can — simple tasks don't need it

**Example: a general-purpose "save words" tail**

```
# Add this to the end of your prompt to cut most of the generation time

Output requirements:
- Conclusion only, no preamble, no pleasantries
- Total length no more than 120 words
- Don't repeat my question
- Don't explain why you answered this way
- Don't use a "first, second, finally" structure

If you need more information to answer, reply only "Missing: xxx" — don't guess.
```

## 03 Switch models, or switch machines?

If you're using an online service, the slowness usually isn't your fault. Trying a different time of day, or a smaller model, often fixes it immediately.

If you're running it on your own machine, the slowness basically means the graphics card isn't enough. In that case, drop the model size first: quality falls a bit but the speed goes up a lot, and it's plenty for most everyday work.

Don't get the order backwards: change the prompt first, then switch to a smaller model, and only then change hardware. The first two steps cost nothing; the third is the most expensive. There's also a dumb trick that works well: save your frequently used prompts as a document and just paste them when you need them. The time you save not writing them from scratch is worth more than a few seconds of waiting.

- Everyday copy editing and emails: use a small model, fast and good enough
- Writing complex code or long-form analysis: use a large model, worth the wait
- Stick to one model for the same task — don't keep switching, so you can judge the speed

## A template you can copy

Paste this in front of your question to force it to talk less

```
Before you answer, follow these rules:
1. State the conclusion first, in one sentence.
2. Then add at most three reasons, each no more than 20 words.
3. Don't restate my question, no opening remarks, no summary paragraph.
4. Keep the total under 150 words.
5. Expand only when I ask for details — don't expand in the first answer.

My question is:
```

## Common mistakes

### × Asking ten questions at once

It has to answer them all in one go. The longer the output, the longer you wait, and the answers get muddled toward the end. Split it into ten

questions — each one is fast, and you can change direction any time.

### × Repeatedly telling it "more detail"

With every follow-up round, it has to reread everything that came before. If you want detail, say so once — don't squeeze it out bit by bit.

### × Waiting it out during peak hours

Online services are slowest during the hours everyone is at work. If a task can move to the morning or the evening, don't cram it in at three in the afternoon.

### × Thinking a bigger model will automatically be faster

Big models are usually slower. Speed and intelligence are two different things — look at which one your task needs.

## Checklist

☐ I know whether this wait was for it to start, or for it to finish writing

☐ I wrote a word limit into the prompt

☐ I didn't paste the whole long document in as-is

☐ I tried changing the prompt and switching to a smaller model before considering new hardware

# 37
# Fine-Tuning: Teach It Your Own Stuff

You don't have to train a big model from scratch. Give it a few dozen of your own examples and it learns your way of writing, your jargon, your formats. This chapter covers what work is worth fine-tuning and what work isn't.

---

## What this chapter covers

☐ You can tell the difference between a case for fine-tuning and a case for writing a prompt

☐ You know how many examples to prepare, and how to pick them

☐ You can write out a data format you can hand straight to a technical colleague or a platform

## 01 Most of the time you don't need fine-tuning at all

Fine-tuning means taking your own examples and training an existing model a bit more, so it fits your situation better. You don't train a big model from scratch. The cost is lower than a lot of people assume.

But first get one thing clear: are you teaching it knowledge, or style? If you want it to know which products the company sells and what they cost, put that in the prompt. No fine-tuning needed.

A prompt you can change on the spot. Fine-tuning means another full run, so changes are slow. But for jobs like these — writing weekly reports in the company's fixed format, answering customers in the reviewer's tone, turning jargon into standard wording — this kind of "feel" work is where fine-tuning earns its keep.

- Factual knowledge: write a prompt, or hook up a document library. Don't fine-tune
- Format and tone: fine-tuning works well

- Tasks that only need a few words changed: a prompt is faster, and easier to adjust

## 02 Picking the examples matters more than the training itself

Fine-tuning runs on examples. You need a batch of samples, each one pairing an input with the ideal output.

On quantity: generally a few dozen is enough to show an effect, and two or three hundred is fairly solid. What matters is quality, not quantity.

There's one principle for picking examples: pick the kind you want it to learn. Don't stuff in the sloppy examples from day to day. It learns whatever you feed it — bad habits included.

- Examples have to cover the situations you actually run into. Don't pick only the easy ones
- A human has to sign off on the outputs. Don't use the previous model's output as the gold answer
- Set aside 20 samples outside training, to check the result

**Example: one sample looks like this — one JSON object per line**

```
{
"messages": [
{"role": "system", "content": "You are the customer service rep for a company. Replies must be short, professional, and never use exclamation marks."},
{"role": "user", "content": "How much is this one of yours"},
{"role": "assistant", "content": "Hello. The standard edition is XXX yuan per year and includes 5 accounts. Would you like me to send a detailed quote?"}
]
}
{
"messages": [
{"role": "system", "content": "You are the customer service rep for a company. Replies must be short, professional, and never use exclamation marks."},
{"role": "user", "content": "Can you do it cheaper"},
{"role": "assistant", "content": "The price is the same for everyone, but paying yearly saves you one month. Want me to work out the yearly figure?"}
]
}
```

## 03 How to check the result afterwards: don't go by feel

The most common problem with fine-tuning is overlearning. It picks up the format very well, but other abilities get worse. Ask it about anything else and it starts making things up.

So check both ends. One end is those 20 samples you held back: are the answers right? The other end is a handful of completely unrelated questions, to see whether it got dumber.

One more thing: fine-tuning won't make it memorize new knowledge. Change a price last week and it still won't know. Whatever belongs in the prompt still belongs in the prompt.

- Check with a fixed set of questions, and run the same set after every fine-tune so you can compare
- Test both versions side by side, the original and the fine-tuned one. Don't test only the new one

- After fine-tuning you still give the background in the prompt. The two don't conflict. If the gain is only about twenty percent, hold off on shipping

## A template you can copy

Before you prepare data, use this to spell out what you need

```
I want to fine-tune a model. The task is: (state it in one sentence, e.g.
"rewrite customer complaints as internal ticket titles")

The input looks like this: (paste 2 real inputs)
The ideal output looks like this: (paste 2 outputs you wrote yourself)

Please help me:
1. Decide whether this task calls for fine-tuning or for writing a prompt, and
give your reasoning.
2. If fine-tuning is the answer, list how many samples I need to prepare, and
which kinds of situations those samples must cover.
3. Give me a format template for one sample that I can fill in.
4. Tell me which questions to test at acceptance time, including a few that
have nothing to do with the task.
```

## Common mistakes

### × Trying to make it remember the company's new rules

Fine-tuning teaches a way of talking, not memory. Rules, prices, product lists — things that change — belong in the prompt or in a document library, where an edit takes effect immediately.

### × Preparing only a dozen or so examples

That's too few. The model won't learn the pattern, it will just memorize. Generally start at a few dozen; two or three hundred is steadier. And cover different situations.

### × Using the model's own output as the gold answer

That's teaching it its own flaws — the more it trains, the more crooked it gets. The outputs used for training must be finalized by a human.

### × Not testing other abilities after fine-tuning

The format may have improved while ordinary Q&A got worse. Always add a few unrelated questions to the check, and compare against the original version.

## Checklist

☐ I've worked out whether this is learning style or learning knowledge — and I don't fine-tune for the latter

☐ My sample outputs are written by a human, not generated by the model itself

☐ I held back a batch of samples from training, purely for checking the result

☐ I tested with unrelated questions and confirmed it didn't get dumber

# 38
# Get Your Data Ready Before Fine-Tuning

Whether fine-tuning — teaching it again with your own examples — works out is eighty percent about the data. This chapter covers where to dig up data, how many records you need, and how to wash it clean.

## What this chapter covers

☐ You know which old records to dig through for data you can actually use, instead of searching the web

☐ You can put together 100+ clean, non-duplicate training records

☐ You know how to hold back an exam set and how to archive things, so you avoid the fake high score of grading yourself on your own homework

## 01 Where the data comes from: dig through your own old records

Don't go hunting online for ready-made Q&A sets. The style isn't yours, so what comes out sounds like nothing in particular, and users spot it right away.

Good data is buried in what you already have: customer service chat logs, tickets, email threads, Q&A in group chats. And proposals you wrote by hand. These are real people in real situations, and the wording is real too.

Collect 200 raw items first, and you may end up with only 100 usable ones. Cleaning costs you volume — go in expecting that.

- Favor work you've done over and over — that's the work with enough records

- Keep both ends of the conversation: how the user asked, and how you answered at the time
- If there are no old records, start collecting today. Have colleagues save good replies into a shared document

## 02 What one record looks like, and how many you need

One record is one pair: what you send it, and what you want it to send back. The jargon for it is a sample. The format is usually one question, one answer.

On quantity: for a single job, start with 100 to 300 records. Under 100 and it won't learn reliably. Past 500 and the gains start to slow.

Coverage matters more than volume. Five records each for the 20 common situations beats a hundred records for one situation, by a wide margin.

- One meaning needs several ways of asking it, long ones and short ones
- Keep tone, length and format consistent across answers. If you use Markdown, use it everywhere
- Deliberately include a few "can't answer this" samples, to teach it when to say it isn't sure

## 03 Washing the data: delete, redact, hold back an exam set, archive

Delete duplicates. Keep one record per meaning. Too much repetition skews it, and in the end it only knows that one way of answering.

Delete the wrong ones. Don't include answers you wouldn't stand behind yourself — it will copy them, including the brush-offs and slips of the tongue you had that day.

Redact. Swap real names for "Zhang San", phone numbers for "138****0000", and scale amounts down proportionally. Then hold back an exam set: pull 10% to 20% at random, and not one of those records may be

used in training.

- Save it as JSONL (a text file with one record per line) — tools can read it directly
- Keep the training file and the exam file separate, put the date in the filename, and don't let one overwrite the other
- Leave a header line: who made it, when, how many records in total, how many were pulled out

**The data ledger looks like this — fill it in as you tidy up (the last column is the usage split)**

```
index,source,scenario,user_question,my_answer,duplicate,redacted,usage
1,customer service chat,refund,Can I get a refund on the course?,Refundable within 7 days of the course starting and no more than 2 lessons watched,no,yes,training
2,customer service chat,refund,I bought the course and don't want to keep going — what now?,Refundable within 7 days of the course starting and no more than 2 lessons watched,no,yes,training
3,customer service chat,invoice,Can I get an invoice?,Yes — after ordering tap Apply on the order page,no,yes,exam
```

## A template you can copy

Send this to an AI to turn messy Q&A into training data

```
I'm giving you a batch of real Q&A records. Please help me turn them into training data, following these rules:
1. One record per line, output as JSONL, with the field named messages containing two entries, user and assistant.
2. Keep only one record per meaning and delete the rest, then tell me how many you deleted.
3. Replace personal names with "Zhang San", phone numbers with "138****0000", and company names with "Company A".
4. If an answer is missing key information, mark it "to be filled in". Don't make anything up.
5. Group them by scenario, tell me how many records each group has, and which groups are too thin and need more.
```

## Common mistakes

### × Padding the count with Q&A sets found online

The style isn't yours, and what comes out is a mishmash. Better to have fewer records from your own chat logs, emails and tickets — even handwritten ones.

### × Preparing only one way of asking

Once it's live, a user rephrases the question and it doesn't recognize it. Write at least 5 ways of asking each meaning, long and short.

### × Using the same data for training and for the exam

The score will be absurdly high, and it falls apart the moment it goes live. You must hold back 10% to 20% and never touch it in training.

### × Writing long, polished answers

Write 800 words and it learns 800 words. Write answers as long as the real situation calls for. A customer service reply is not a little essay.

## Checklist

☐ My data comes from real records, not Q&A sets scraped off the web

☐ I can put together at least 100 pairs of "the user asks this + I want it to answer that", and they cover a range of situations

☐ Duplicates and wrong answers are deleted, and sensitive details are replaced with fake ones

☐ I held back 10% to 20% from training, purely for the exam

# 39
# How to Tell Whether It Actually Works: Evaluation

Don't say "it seems better" on a hunch. Evaluation — scoring the same fixed batch of questions over and over — is what tells you how much changed between before and after, and where the difference is.

## What this chapter covers

☐ You can write 30 questions of your own and turn them into an exam set you can reuse

☐ You know to track the pass rate, not a few flattering examples

☐ Swapping models, rewriting prompts, adding retrieval — all of it can be measured with the same ruler

## 01 Start by accepting one thing: feelings don't count

You tweak one line of the prompt, try three examples, and it feels much better. Of course it does — you picked those three examples yourself.

Once it's live, users ask things you'd never think of. To compare, you need a fixed batch of questions, and you run the same batch every time. Impressions aren't enough. Impressions always lean toward the last few tries.

That batch of questions plus the scoring rules is called an evaluation set (a fixed set of test questions, together with a standard for what counts as passing). It's like a health check report: only the same measurement, before and after, means anything.

- Pull the questions from real records. Don't make them up

- 30 to 50 questions is enough to start. You don't need hundreds on day one
- For every question, write down what counts as passing

## 02 How to write the questions, how to score them

Go through your real records: customer service chats, questions asked in group chats, requests colleagues sent you. Pick the 30 most representative, with both the common ones and the tricky ones. Don't pick the ones you answered most beautifully.

There are two kinds of scoring. What can be judged automatically, judge with a script. For example, the answer must contain a certain field, or must be valid JSON (a data format with a fixed way of writing it). What can't be judged that way, hand to another model as a judge, scoring from 1 to 5.

In the end, track one number: the pass rate. 21 out of 30 is 70%. After the change, 24 is 80%. That's what getting better looks like.

- When a model is the judge, put the scoring criteria in the prompt. Don't just say "give it a score"
- Run the same batch twice. If the results differ a lot, the questions are unstable and need fixing
- Record the date, what you changed, and the pass rate in the same table, every time

## 03 A good total score doesn't mean there's no problem

70% doesn't mean it's fine. You have to look at what the other 30% got wrong.

Sort the failures into types: didn't understand the question, answered too wordily, invented information that doesn't exist, wrong format. Fix whichever type is largest first. Then run another round and see whether that type went down.

There's one more metric that's easy to miss: when it should say "I don't know", does it? Put at least 5 of those questions in the set.

- Go through the failures one by one. Don't just stare at the percentage
- Change one thing, run one round. Change three at once and you won't know which one did the work
- Don't swap the model and rewrite the prompt at the same time, or you can't compare anything

**A little script for the pass rate (results.jsonl has one line per question, with its score and its type)**

```
# After one round, boil the results down to one number
import json
rows = [json.loads(x) for x in open('results.jsonl', encoding='utf-8')]
passed = sum(1 for r in rows if r['score'] == 1)
print(f"{passed}/{len(rows)} = {passed/len(rows):.0%}")
# Then break it down by question type, to find a whole category that fails
for t in sorted({r['type'] for r in rows}):
sub = [r for r in rows if r['type'] == t]
print(t, sum(1 for r in sub if r['score'] == 1), '/', len(sub))
```

## A template you can copy

The scoring prompt for a model-as-judge — copy and use

```
You are a strict grader. Below you get a question, a few key points for the expected answer, and one answer to grade.
Score on these three points, 0 or 1 each:
1. Did it actually answer the user's question? An answer that misses the point scores 0.
2. Did it introduce specific information not in the key points — numbers, dates, links? If it made something up, score 0.
3. Is the format right? If JSON was required, it must be valid JSON. If three lines or fewer was required, it can't run over.
Output one line only: score|reason. Keep the reason under 30 characters.

Question: {{question}}
Key points for the expected answer: {{expected}}
Answer to grade: {{answer}}
```

## Common mistakes

### × Only looking at the examples that worked

That's fooling yourself. Run the whole set, and write down the ugly results too, or you'll never know where you actually stand.

### × Changing three things at once

Swap the model and rewrite the prompt, the score goes up, and you don't know what to credit. Change one place at a time, run the round, then touch the next.

### × Putting the exam questions into the prompt

The score becomes absurdly high and it collapses the moment it ships. Keep the exam questions hidden; whoever rewrites the prompt must not tune against them.

### × Looking only at the average score

One whole category can fail completely while the average still looks fine. Break it down by question type and find the places where everything fails.

## Checklist

☐ I have a fixed set of 30+ questions, all drawn from real situations

☐ Every question has a written standard for passing, not a gut call

☐ I recorded the pass rate before and after the change, and I changed only one thing at a time

☐ I went through the failures one by one and sorted them into types

Part Six

# Safety, Teamwork and Launch

Chapters 40 — 54

# 40
# The Safety Line: What You Don't Do

There are a few lines you don't cross when using AI: other people's privacy, your company's secrets, and anything you sign your name to. This chapter covers what not to hand over, and how to guard against it.

---

## What this chapter covers

☐ You know which three kinds of information never go into a chat box

☐ You can set a three-sentence rule for your team that people will actually follow

☐ You know which parts of what AI gives you have to be checked by you

## 01 Three Kinds of Things That Don't Go in the Chat Box

First, other people's privacy. A customer's real name, phone number, ID number, home address, medical records, chat logs. Paste those in and you've handed over someone else's information. You can't take it back.

Second, company secrets. Financial reports that aren't public yet, contract terms, source code, product plans that haven't shipped, internal pricing. When this kind of thing leaks, the loss isn't something an apology can cover.

Third, anything that converts straight into money. Keys (a string of characters that lets a program log into an account), database passwords, admin panel addresses, screenshots of internal systems.

- If you're not sure whether it counts as a secret, treat it as one
- If you really need it, swap it for something fake first: real name becomes "Zhang San", company name becomes "Company A", amounts scaled down

- When you're done, delete that conversation. Don't leave it sitting in your history

## 02 It Can Help With This Work, But You Sign Your Name

Law, medicine, tax, investing. In these areas AI can help you understand the concepts, sort out your thinking, and build a list of questions. But it can't make the decision for you.

It doesn't know your specific situation, and it may not know the latest rules. The statute numbers, drug names and dosages, tax rates it hands you may all be made up.

The right way to use it: have it list what you should be asking. Then take that list to someone who's actually qualified. Its job is to help you think of every question, not to give you a conclusion.

- For anything involving a signature, a payment, a tax filing or medication, AI output is a draft at most
- Every law, paper or link it cites — open it and check it, one by one
- Anything official that goes outside has to be read start to finish by a person

## 03 Set a Rule for Your Team That People Can Actually Follow

Keep the rule short. Short enough that people remember it. For example: customer information always goes by code name, company files never leave the company, and anything AI writes gets read by a person before it goes out. Write those three sentences down and pin them in the group announcement. That does more than stressing it at a meeting.

A rule on its own does nothing. Someone has to be available to ask. Naming one person for "when you're unsure, go to them" does more than ten pages of policy. That person needs some standing — not the intern who just started.

Then add one safety net: if something has already been pasted out there, don't hide it. Say so right away. Early word leaves room to fix

it. Hiding it only makes things worse.

- Make a list of the tools people are allowed to use. Anything off the list waits
- Tell new colleagues these three rules on day one, don't bury them on page 8 of the handbook
- Every quarter, spot-check a few conversation logs to see whether sensitive information went in

## A template you can copy

Paste this straight into your team group announcement and you have a rule people can use

> Three rules for using AI. Please follow them:
> 1. Real names, phone numbers, ID numbers and addresses of customers and colleagues — always swap for a code name before you use them.
> 2. Contracts, financial reports, code and pricing that aren't public — never send them.
> 3. Anything AI writes gets read start to finish by a person before it goes out. If you're unsure, ask [name of the person in charge]. If something already went out, don't hide it — say so right away, and there's still time to deal with it.

## Common mistakes

### × "Just a tiny bit"

One short stretch of a chat log already has a customer's phone number and order number in it. Either redact the whole thing, or describe it a different way.

### × Thinking that deleting the conversation fixes it

Your input may already have been recorded. Think it through before you paste. Don't count on cleaning it up afterwards.

### × Taking AI's legal opinion as real

The statute numbers it gives are often made up, and they read very convincingly. Treat it as reference, not as a basis.

### × Assuming it's fine because the company has no rule

No rule doesn't mean it's allowed. Asking first is much safer than guessing.

## Checklist

☐ Nothing I pasted in contains anyone else's name, phone number, ID number or address

☐ No unpublished contracts, financial reports, code or pricing

☐ No keys, passwords or admin panel addresses

☐ Someone has read start to finish anything AI wrote before it went outside

# 41
# Don't Let Sensitive Information Slip Out

The code, logs and customer lists you casually paste in may already have left the company. This chapter lays out what you shouldn't show AI, and how to swap in fake data when you have to use it.

## What this chapter covers

☐ You can spot at a glance which content counts as sensitive and can't go outside

☐ You learn to swap real data for fake data before you ask, and still get back something usable

☐ You know how to set yourself a rule you can actually put into practice when the company has none

## 01 First, Get Clear on Where the Stuff You Paste Goes

You paste a piece of code into the input box and hit enter. That text has just been sent to somebody else's server. It isn't computed on your machine.

Most tools save the conversation by default, and some use it to improve the model. How long it's kept and who can see it differs by vendor — and can change at any time.

So the test is simple: if this text showed up in the company group chat tomorrow, would you panic? If yes, don't paste it.

- Anything public, paste away: code from open-source libraries, error messages you found online, exercises you wrote yourself
- Anything not public gets masked, no exceptions: customer names, phone numbers, ID numbers, order numbers, internal domains, keys

- If you're not sure, treat it as not public. Two minutes of rewriting is cheaper than explaining later

## 02 The Three Most Common Ways It Leaks

First, pasting a whole log. A log usually carries the server address, the database account, user IDs. You thought you were only showing it an error.

Second, sending the entire config file. It often holds keys (a string that logs straight into your account) and tokens for third-party services.

The third one is the sneakiest: screenshots. A screenshot carries the other people's names in the chat window, your browser tabs, the notification popping up in the corner. You thought you only captured the code.

- Before you paste a log, swap the IPs, domains, accounts and user IDs for fake values like example.com
- If you have to paste a config file, delete the values after key and token and keep only the field names
- Before a screenshot, close the other windows. Or better, copy the text out — don't use an image

## 03 When You Have to Use It, Swap the Data for Fake Data

Most of the time you don't need real data. AI wants the structure, not the content. Swap the real name for Zhang San and the amount for 100, and it will still help you straighten out the logic.

After the swap, keep a mapping table for yourself. The plan it hands back was written against fake data, so when you apply it to the real data, check that no field ended up in the wrong slot.

If you truly do need real data, use an internal tool the company has approved. If there's no internal tool, don't use it. That isn't a technical problem, it's a process problem.

- Fake data should keep the original shape: phone numbers stay 11 digits, dates written the way they were
- Don't use "Customer A, Customer B" — easy to mix up. Use numbered ones like "customer_001" so you can match them back later
- Add a line to your prompt: the data below is all fake, just analyze the structure, don't worry about the specific values

**Example code: batch-replace the sensitive bits in a log**

```
import re

def mask(text):
text = re.sub(r'\b\d{11}\b', '[phone]', text)
text = re.sub(r'\b[\w.-]+@[\w.-]+\.\w+\b', '[email]', text)
text = re.sub(r'\b\d{1,3}(\.\d{1,3}){3}\b', '[IP]', text)
text = re.sub(r'(?i)(key|token|secret|password)\s*[=:]\s*\S+', r'\1=[hidden]',
text)
return text

with open('app.log', encoding='utf-8') as f:
print(mask(f.read()))
```

## A template you can copy

Have AI write you a masking script that matches your own log format

```
I have a batch of logs to show someone else, but they can't carry sensitive
information. Each line looks roughly like this:
2024-01-01 10:00:00 INFO user=13800000000 ip=10.0.0.1 msg=login ok

Please write me a Python script that does these things:
1. Replace phone numbers, IPs, emails and ID numbers with [hidden].
2. Replace the value after key= and token= with [hidden], keep the text before
the equals sign.
3. Don't change a single character of anything else, including spaces and line
breaks.
4. The script reads one input file and writes a new file. Don't touch the
original.

Output only the script first, no explanation.
```

## Common mistakes

### × Thinking that deleting the conversation fixes it

All you deleted is the record on your side. The server may still have a backup, and it may already have gone into training data. The moment you send it, you can't get it back. Don't treat deleting as a fix.

### × Changing one spot and sending it

In the same piece of code, a customer name usually turns up in three or four places: comments, variable names, test data. Search the keyword before you paste. Don't go by memory.

### × Asking about company projects from a personal account

Pasting company code into a personal account is moving internal material somewhere the company can't manage. For project work, use the account and tools the project gives you.

### × Pasting the key in so it can help you debug

Nine times out of ten it won't connect because of permissions or format, and pasting the key doesn't fix that. If you really need to show it, swap in fake characters of the same length and keep the format identical.

## Checklist

☐ Before pasting, I asked myself: if this showed up in the company group chat tomorrow, would I panic?

☐ Phone numbers, IPs, accounts and keys in logs, configs and error messages — all swapped for fake values

☐ I used fake data, but the field formats match the real ones, so the answer drops straight in

☐ For project work, I only use accounts and tools the company has approved

# 42
# How to Catch It Making Things Up

The smoother it sounds, the more likely you are to believe it. This chapter gives you a few moves you can use right away to pick the false parts out of an answer in a couple of minutes.

## What this chapter covers

☐ You learn to press for details, so made-up content gives itself away

☐ You know which kinds of content you have to verify, so you don't have to check every sentence

☐ You can split an answer into "use it as is" and "I have to verify this myself"

## 01 Why It Sounds So Real When It Makes Things Up

Because its goal when writing is to read smoothly, not to be right. For it, those are two separate things.

A fabricated passage looks exactly like a true one as long as the sentences flow and the vocabulary sounds professional. While you read, what you're actually judging is smoothness, not correctness.

So to catch its mistakes, reading isn't enough. You have to ask. Reading only makes you feel comfortable; asking is what makes it slip.

- The more specific, the more suspicious: names, years, ID numbers, links, statute numbers, citations
- The more obscure, the more suspicious: what it knows well is a small set it recycles, and the moment you step into a niche it starts making things up
- The more it sounds like a summary, the more suspicious: "studies show" or "the industry generally agrees", with no word on who studied

it or who agrees, is usually filler

## 02 Three Moves, and You'll Know in a Few Minutes

Move one, ask for the source. When it says "studies show", you ask: which study, who wrote it, where was it published, what year. Something it actually knows, it can answer. Something it made up starts to dodge.

Move two, ask the reverse. Turn the conclusion it just gave you upside down: what if the situation were exactly the opposite? Something it genuinely understands will tell you where the boundary is. Something it made up will change its story to follow you.

Move three, have it break out the steps. Have it write out clearly how the conclusion was derived step by step, and what data each step used. Made-up content falls apart the moment you take it apart.

- When you ask for the source, watch whether it switches wording. Going from "studies show" to "generally speaking" means it knows it's on thin ice
- When you ask the reverse, if it argues both sides convincingly, that conclusion carries basically no information
- When it breaks out the steps, stare at the sentence right before "therefore". The false part is often hidden in that step

## 03 What You Must Check, and What You Can Let Go

You don't check every sentence — that's slower than writing it yourself. Sort by consequence and you're done: if being wrong would make you redo the work, check it. If being wrong would just read awkwardly, let it go.

What you must check: commands and parameters you can copy straight into code. Facts and numbers in copy that goes outside, too. Anything involving money and rules, confirm yourself, every time.

What you can let go: the frameworks that help you organize your thinking, tone adjustments, an explanation of a concept you already understand. You'd spot a mistake in those at a glance.

- Run the code first, then talk. Half the errors in the function names and parameter names it gives you show up after one run
- Check numbers and dates yourself, every time. That's where its mistakes cluster
- Anything involving other people's words, commitments or pricing — don't believe a word of it, confirm all of it yourself

**Example prompt: make it confess where it isn't reliable**

> Below is the answer you just wrote. Go through it again in this format:
>
> 1. Fact list: pull out every specific name of a person or organization, year, number and link, and list them separately.
> 2. Degree of certainty: after each one, mark "certain", "probably" or "not sure".
> 3. Source: if you can state the source, state it. If you can't, write "I have no reliable source".
> 4. Rewording: take sentences like "studies show" or "the industry generally agrees" that have no subject, and rewrite them into something you can actually stand behind.
>
> Don't rewrite the body text. Output only this list.

## A template you can copy

When you're unsure, use this to get the same question answered twice and compare

> Answer in two parts. Write the two parts separately, don't mix them together.
>
> Part one: give the conclusion and the reasoning in your own words, as specific as you can be.
>
> Part two: now play someone whose whole job is finding faults. Go sentence by sentence through part one and look for holes. For each sentence, check these four things: no basis, possibly outdated, presenting a special case as the general rule, and a "studies show" with no subject.
>
> At the end, add one separate sentence: which sentence in part one is the least reliable.

## Common mistakes

### × The more professional it sounds, the more you believe it

Heavy terminology plus formal sentence structure is exactly the combination that fools people most easily. Switch your test to "can it state the source and the steps", not "does it read like an expert".

### × Asking once and calling it done

Ask only once and you get the version it's most confident about. Ask again in different words, and wherever the two answers don't match is where it isn't sure of itself.

### × Using it as a search engine

It doesn't look anything up online. It relies on what it memorized during training, which may be years out of date. For the latest information, search yourself, or use a tool with web access — and open the link to read the original.

### × Thinking a confession means the rest was right

Apologizing when it's called out is its default reaction. It doesn't mean it actually fixed anything. After it rewrites, check again the same way. Don't let good manners buy it a pass.

## Checklist

☐ Specific numbers, dates, links and names — I copied none of them directly, I checked every one myself

☐ I asked where the conclusion came from, and it gave a source I can check, not a rephrasing

☐ I ran the code it gave me, and fixed the errors before using it

☐ For anything going outside, I confirmed the facts myself. I didn't just write whatever it said

# 43
# One Checklist: The Most Common AI Coding Pitfalls

When AI-written code goes wrong, eight times out of ten it isn't that the AI is dumb. It's that you didn't say clearly what you wanted at the start, didn't take small steps in the middle, and didn't check at the end. This chapter walks you through it in order, from kicking off to wrapping up.

## What this chapter covers

☐ Know which few things you must spell out before every new conversation

☐ Learn to cut one big requirement into pieces you can check, so a bad change can be rolled back

☐ Get a set of wrap-up checks for judging whether this code is actually usable

## 01 Before You Start, Say What You Want Clearly

The most common failure in AI coding isn't that the AI is dumb. It's that you didn't say clearly what you needed. It doesn't know what language you're using, where the code runs, or what libraries it has to work with.

There's one move that saves you half the rework: open with a bit of project background. Five lines is enough. What language, what framework, what database, whether the code runs locally or on a server, and what must not be touched.

Save that background somewhere, and paste it as the first line of every new conversation. Much less hassle than explaining it again each time.

- Say which files must not be changed. Otherwise it will helpfully rewrite parts you never asked it to touch
- Raise one requirement at a time. Squeeze three requirements into one paragraph and it often only gets the first two right
- Show it the original error message. Don't paraphrase it into "it won't run"

## 02 While It Writes: Small Steps, Every Step Reversible

Ask it to write five hundred lines in one go and you basically can't check any of it. Do it right: cut it into small pieces. First have it write the smallest version that runs. You run it. Then have it add the next feature.

Commit once before every change — that is, save a checkpoint of the current code. If it breaks something, one command takes you back. No hunting line by line.

When it hands you code, first look at which files it changed and which lines it touched. Where you don't follow, ask it to explain. Don't just copy and paste the whole block.

- Change one place at a time. Touch three files at once and when something breaks you won't know whose fault it is
- Have it list its changes: which file, which functions were added or removed, and why it did it that way
- If it doesn't run, don't add more features. Keep writing on top of a bug and the rest is all patches

**Example prompt: have it work in small steps**

> We'll go one step at a time, one thing per step.
>
> Step one: first tell me how you plan to change things — which files, and what you'll change in each one. Don't write code yet.
> After I confirm, then you write.
>
> When you write, follow these rules:
> 1. Only change the files I name. Not a single character in any other file.
> 2. Mark your changes with comments: "change start" and "change end".
> 3. When you're done, give me a list: which files changed, what you added, what you deleted.
> 4. If you think something else should change but I didn't mention it, list it separately and ask me. Don't do it yourself.

## 03 Wrapping Up: How to Tell Whether This Code Is Usable

Running isn't the same as correct. Getting it to run only means you didn't trip on this one path. Every other path may be full of holes.

The minimum bar: have it write a few test cases and run them yourself. Test cases are small programs that automatically check "given this input, it should output that". Focus on the edges: nulls, zero, very large numbers, duplicate submissions.

One more thing that often gets skipped: have it state the assumptions in the code. For example, that ids won't repeat, or that all time zones are Beijing time. The moment one of those assumptions fails, the code goes wrong quietly.

- Have it list the situations where this code will fail. Much faster than working it out yourself
- Code that touches money, permissions, or deleting data: read it line by line. Don't skip
- Before you ship, ask one question: what's the worst thing this code could break? If it can't answer, that means you don't understand it yet

## A template you can copy

The last question before you wrap up — copy it as is

> The code is written. Please do three things:
>
> 1. List every assumption you made here: data format, value ranges, time zone, concurrency. For each one, say what happens if the assumption doesn't hold.
> 2. List the 5 inputs most likely to cause problems, and explain how the code will behave for each one.
> 3. Write 5 test cases covering, respectively: the normal case, null values, an oversized value, a repeated operation, and insufficient permissions. Write them in a format I can run directly, not pseudocode.

## Common mistakes

### × Asking for a whole complete project in one go

What it gives you looks complete, but the pieces don't line up, and you still have to read the whole thing from the top. Ask for the skeleton first, get it running, then add one feature at a time.

### × Pasting without reading the changes

It may delete a validation you didn't notice, or hard-code the config. Before each paste, look at which files and which lines changed. If the change is big, have it explain first and hand over the code after.

### × Sending only "it won't run" as the error report

All it can do is guess. Send the full error, the exact command you ran, and the few relevant lines of code together. One clear message beats five rounds back and forth.

### × Letting it make big changes with no checkpoint

If it breaks, you can't go back, so you keep fixing on top of it and it gets messier and messier. Commit once before you start, or copy the file somewhere. That's one minute of insurance.

## Checklist

☐ Every new conversation, my first line is the project background: language, framework, where it runs, what can't be touched

☐ I give it one thing at a time, and only add the next after the last one runs

☐ After it makes changes, I look at which files and which lines changed, and ask it to explain anything I don't follow

☐ Before wrapping up, I have the test cases and the assumption list, and I've run them myself

# 44
# Rules for Using AI as a Team

Once three or more people use AI together, two kinds of mess show up most: material pasted in the wrong place, and nobody owning up when something goes wrong. This chapter gives you a set of rules you can copy straight out. You can have them in place in half a day.

## What this chapter covers

☐ Settle on three bottom lines with your team that nobody will argue with

☐ Know which prompts to save, and how to name them so you can find them

☐ Be able to say clearly who's responsible when AI-written work goes wrong

## 01 Set the Three Bottom Lines First

When a team's AI use goes wrong, eight times out of ten it isn't a technical problem. It's that there are no rules. Set three first, pin them in your work group, and a new hire sees them on day one.

Line one: customer data, contracts, quotes, employee information — not one character of it goes into the chat box. If you want to try something, swap in fake data. Name the person "Zhang San". Change the amount to 100.

Line two: anything sent outside must be read top to bottom by a person. Line three: label AI-written first drafts. Don't fold them into your own work. All three together come to under ten lines. More useful than holding a meeting.

- Bottom line one: real data doesn't leave the room. Every example uses fake data

- Bottom line two: before anything goes out, a real person has read it all the way through
- Bottom line three: documents carry one line — "First draft generated by AI, revised by a human"

## 02 Good Prompts Need a Shared Drawer

A phrasing one person has tuned is worth nothing if nobody else knows about it. Set up a shared document. Whoever tunes something good puts it in.

Each entry has only four things: what it's for, the exact wording, which words to swap, and roughly what the result looks like. Don't write reflections. Write something that can be copied straight out.

Name them all "use-scenario", for example "weekly report-sales team". Three months later you can still find it. That's the whole point.

- On a team of ten, usually no more than 20 entries stay in use long term. Don't be greedy
- Leave a "last verified" date under each one. Anything nobody has used in over six months gets deleted
- A new hire has to be able to find this document on day one. If they can't find it, it may as well not exist

## 03 AI Writes the Draft; a Person Signs It

Whoever sends it out is responsible. AI doesn't take the blame, and can't. Responsibility can only land on a specific person.

In practice: when it finishes the work, you go through it on three points. Are the numbers right? Are there any citations it made up? Does the tone sound like us?

All of it takes under five minutes, and it blocks the vast majority of embarrassing mistakes. Best to fix one sentence in the team: "Have you read this version?"

- Numbers, names, dates, links — you must check these four yourself

- If the tone is off, have it rewrite based on a finished piece of yours from before. Don't force-fix it by hand

## A template you can copy

Team AI usage agreement — copy it into a shared document and it's ready to use

```
# Our team's AI usage agreement
1. What we don't do: no pasting customer data, contracts, quotes, or employee information; no letting AI reply to customers directly; no handing over content nobody has read.
2. What we do: before anything goes out, a real person reads it through and rewrites it into plain human language; at the end of the document, note "First draft generated by AI, revised by (name)".
3. Prompts live in the shared document. Each entry states the use, the exact wording, the words to swap, and the result.
4. When the same kind of mistake happens twice, add a new rule.
```

## Common mistakes

### × Everyone tunes their own prompts

Ten people, ten different phrasings. Nobody knows about the good ones, and everyone steps in the same holes in turn. Have at least a shared document. Whoever tunes something good pastes it in, with the use and the words to swap written clearly.

### × Letting AI reply to customers directly

One wobbly tone or one wrong number and what you lose is trust. AI can only produce a draft. Before it goes out, a person has to read it and rewrite it so it sounds like something a human would say.

### × Testing on a real customer list

Once pasted in, it may be stored. If you want to test, build a fake dataset: same column names, all fake content. When you're done, put the real data back.

### × No acceptance criteria

If you only say "help me write a proposal", nobody knows afterwards whether it passes. Before work starts, write down what's being delivered, how long it should be, who it's for, and what counts as good enough.

## Checklist

☐ Our team has bottom lines written down, and new hires can see them

☐ Every entry in the shared prompt library can be copied directly — not empty talk

☐ For anything we deliver externally, I can say who read it last

☐ When I try out AI, I use fake data, not real customer information

# 45
# A Project Template You Can Start Work With Right Away

Explaining the background to AI again for every new task wastes a lot of time. This chapter gives you four folders plus one description file. Fill it in and you're working in five minutes.

## What this chapter covers

☐ Use a fixed structure to gather material scattered in all directions into one place

☐ Write a description file that AI understands on the first read

☐ Know what to say in the first round of conversation, and skip two or three detours

## 01 Build Four Folders First, With Fixed Names

Whatever the project, build these four: 00-material, 01-output, 02-reference, 09-archive. The names start with numbers, so the sort order never breaks.

00-material holds the raw files: the spreadsheet the client gave you, meeting notes, chat screenshots. Put them in, don't change them. If you need to edit, save a copy into 01.

01-output holds what you produce. File names carry the date, like "20250310-weekly report-draft". 02-reference holds other people's material. 09-archive holds finished work. Archiving isn't deleting — it's moving aside. Three months later it's still there when you need it.

- Material is read-only. That's the most important rule in this structure

- File names follow "date-content-status". There are only three statuses: draft, in review, final
- Create the empty folders up front too. Once the places are set, things get put away properly. Don't wait until you have a file and then wonder where it should go

## 02 Write a Project Description and Have AI Read It First

Every time you open a new conversation, you have to lay out the background all over again. That's the biggest time sink. The fix is to write it into a file. Write it once, copy it from then on.

The file is only seven lines. What you're doing, who it's for, what you finally deliver, what hard constraints there are. Plus: what you don't want, where things stand now, and what you need its help with next.

Save it as "project-description.md" in the project root. Seven lines usually come to under two hundred words. It fits on one screen.

- If you can't write it, that means you haven't thought it through. Spend ten minutes thinking first, then start
- When the description file changes, say so in the group chat. Don't leave people working from an old version

## 03 First Round: Have It Repeat the Task Back

After you paste the description file, don't rush it into the work. Add one line: first repeat back the task as you understand it, then ask me three things that aren't clear.

If it repeats it back wrong, you see it at a glance and fix it with one sentence. Discovering the direction is off only after it has written three thousand words — that's what really costs time.

Once it's confirmed, let it start. Two minutes here usually saves two rounds of rework. Two rounds of rework and your whole afternoon is gone.

- Letting it ask questions is far more accurate than letting it guess

- One thing at a time. Give the next one after you've confirmed the last
- Where it can't answer, that's exactly where your description file wasn't clear

## A template you can copy

Your first line at kickoff — paste it to AI together with the project description

```
Read the project description below first. Don't start writing right away.

Step one: repeat back the task goal in three sentences, as you understand it.
Step two: list three things that are still unclear and that you need me to fill in.
Step three: after I confirm, start according to the requirements below.

[Project description]
- What we're doing:
- Who it's for:
- Final deliverable:
- Hard constraints: word count, deadline, format that must be used
- What we don't want: no colloquial tone, no emoji, no more than 3 pages
- Where things stand now:
- Already done:

[What to do this time]
Write one specific thing only, and describe clearly what the deliverable looks like.
```

## Common mistakes

### × All the requirements live in the chat history

The rule you set last week is unfindable this week. Anything that gets decided goes into the project description file. Chat history doesn't count as a record.

### × Writing the description like an essay

A long block of adjectives, and after reading it AI still doesn't know what to deliver. Write it as a list, one sentence per item, with word count, format, and deadline all spelled out.

### × Asking for the whole text right away

If the direction is wrong, all three thousand words are wasted. Ask for an outline first, or the first paragraph. Once it looks right, build from there.

### × Stuffing every file into one folder

Material, output, and other people's references all mixed together, and finding a file takes ten minutes. Split them into the four folders, and put the date and status in the file name.

## Checklist

☐ I know where this project's material is and where the output goes

☐ I have a project description file, and pasting it is the first thing I do in a new conversation

☐ I have it repeat the task back before it starts working

☐ My file names carry the date and status, so whoever takes over can understand them

# 46
# Building Your Own Snippet Library

If you've looked the same thing up a third time, save it. This chapter is about building a snippet library you can still find and still understand three months from now, with no software to install.

---

## What this chapter covers

☐ You can tell what's worth saving and what's junk the moment you save it

☐ You can write three clear lines of notes for every snippet, so you still understand it half a year later

☐ You know how to let AI read your library directly, so you don't re-describe everything each time

## 01 Save Only What You'll Use a Second Time

The test is simple: is this the third time you're looking up the same thing? Then save it. Save everything the first time you see it, and the library turns into a dump fast.

What's worth saving: the spreadsheet formulas you use all the time, the commands you use to search logs, a bit of regex (a rule for matching text). Also the config settings you have to dig through the docs for every single time.

What isn't worth saving: the common stuff one search will find, a temporary script you'll use once, old code you don't understand yourself. You won't open it again.

- Once the library passes 200 entries, clean it out; what you actually use is usually no more than 30
- When you're not sure, ask yourself: will I still need this in three months. If you can't answer, don't save it

- Get to 20 entries before you talk about categories; sorting from day one just gets you stuck

## 02 Three Lines Each, Not an Essay

The worst thing for a snippet library is a pile of code with no beginning and no end. Open it half a year later and all you see is symbols. You can't remember what you were thinking.

Write four fixed lines for each one: what it does, when to use it, what needs changing, and what words will find it. The third line matters most, so spell out which words you have to swap for your own.

The fourth line is the trigger words, meaning the words you'll actually search with. For example "dedupe", "export to Excel". Two or three trigger words is enough. Write more and you can't remember which one to search.

- Put the notes above the snippet, not below, so the purpose is the first thing you see
- Make "what needs changing" concrete: column names, file names, paths. Don't write "modify as needed"

## 03 Let AI Read Your Library Directly

Saving it is only the first step. It only counts once you actually use it. The simplest approach: export the whole library into one text file and paste it in when you ask.

Add one line at the top: this is my usual snippet library, read it through first, pick whatever fits my need and adapt it, don't write a separate version from scratch.

That way what it gives you matches your own habits, and you don't have to describe everything from zero every time. The bigger the library, the more that one line saves you.

- Keep the whole library in one file instead of splitting it into a pile; pasting is easier

- Every quarter, move your 10 most-used snippets to the top; both you and the AI find them faster
- When it writes something good, ask right there whether to add it to the library. Don't wait

## A template you can copy

Paste this before your snippet library when you ask

```
Below is my usual snippet library. Read it through first, don't rush to answer.

Rules:
1. If something in my library already works, use it first, and say which entry you used.
2. If it needs changes, change only what's necessary, keep the rest in my style.
3. Only write something new when the library has nothing suitable, and tell me whether this one is worth adding.
4. Don't bring in tools or approaches I haven't used.

[My snippet library]
(paste the whole file content here)

[This request]
(spell out what to do and what the result should look like)
```

## Common mistakes

### × Saving everything you see

Three months later there are five hundred entries and you can't find one. Save only what you're still looking up for the third time; let the rest go.

### × Saving code with no notes

Open it half a year later and all you see is symbols. You don't dare use it and don't dare delete it. At minimum, write what it does, when to use it, and what needs changing.

### × Too many categories

You build twenty categories, then can't decide where to put things when saving, so everything ends up on the desktop. Keep categories under five. Searching by trigger word works better, so don't build a directory tree.

### × Save it and forget it

The environment changes or a tool gets upgraded, and old snippets may just error out. Every quarter, run the ones you use. Fix or delete whatever doesn't run.

## Checklist

☐ Before I save something, I ask myself: is this the third time I've looked it up

☐ Every snippet says what it does, when to use it, and what needs changing

☐ I can find the snippet I want with a single word

☐ I clean out my library every quarter, deleting what doesn't run and what I never used

# 47
# Write Scripts with AI and Kill the Repetitive Work

Clicking the same buttons every day, reformatting the same tables. Have AI write you a small script. Write it once and every day after is easier.

## What this chapter covers

☐ You can spot which of your tasks are worth turning into a script

☐ You can get AI to write a script that runs as-is and that you more or less understand

☐ You know what to test and what to back up before a script runs

## 01 Pick the Work You Do Several Times a Week

What counts as repetitive work? The steps are identical; only the file names or numbers change. For example, making column widths and font sizes the same across ten spreadsheets, resizing a batch of screenshots to one size, converting dozens of Word files to PDF.

That kind of work is the best candidate for a script. Doing it by hand takes you twenty minutes. The script runs in two seconds. Spending half an hour to write it pays for itself by the second week, and every run after that is free money.

The other way around: work you do once a month, and differently each time, isn't worth scripting. Writing the script costs more time than it saves. If you're not sure, do it by hand twice first and see whether it really repeats.

- Count first: three or more times a week with fixed steps, and it's worth writing

- The steps have to be explainable out loud. If you can't explain them, walk through it yourself first and write down each step
- Keep a real file as a sample and have AI write against it

## 02 Three Things You Must Spell Out When You Ask for a Script

If you only say "write me a script that renames files in bulk", it can only guess. The guess will run, but it won't be what you wanted.

You have to cover three things: where the files are, what they look like now, and what you want them to look like. Add one more line, "don't touch the original files". Nail those three and the code it gives you will mostly run as-is.

You also have to describe your computer, for example "I'm on Windows, Python is installed, I don't want to install anything else". Only then does it know which style to give you. Otherwise you may not even be able to run the code it hands back.

**Example prompt**

```
I have a folder, the path is D:\reports\raw.
It's all Excel files, with names like "January 2024 sales.xlsx".
I want to do three things:
1. Rename the files into the 2024-01-sales.xlsx format
2. In each file, make the first row bold and the background light gray
3. Save the results to D:\reports\output, and don't touch a single original file

My computer is Windows, and Python is already installed.
Please give me a script I can run directly, with a one-line explanation in front of each block of code.
Before running, tell me which files it will touch.
```

## 03 First Run: Test on Three Files

Don't run it against the whole folder right away. Copy three files into a new folder and run it there first. That one step blocks about 80%

of the traps.

When it finishes, open the results and look: are the names right, is the content intact, did any strange files show up. When all three check out, go full-scale. Don't relax just because the first one worked; the first one is usually the most well-behaved of the lot.

Have the script back things up first. Tell AI "copy the originals into a backup folder before changing anything" and it writes that in as it goes. If it really does break something, you can still get it back.

- Get it running in the sample folder first, then run everything
- For important folders, copy them by hand before you run; don't trust the script alone
- After the run, spot-check five files, not just the first one

## A template you can copy

When you ask AI for a script, cover everything in this format

```
Write me a script, requirements as follows:
1. Input: the folder "[fill in your path]", containing [fill in file type], about [fill in count] files
2. For each file: [fill in step one], [fill in step two]
3. Output: save to [fill in output path], file naming rule is [fill in rule]
4. Don't modify the original files; copy them into a backup folder before running
5. If a file can't be handled, skip it and tell me its name
6. Put a one-line explanation in front of each block of code, and tell me at the end how to run it
```

## Common mistakes

### × One sentence, then expect a script

Too little information, so it can only guess what your files look like. The code looks decent and errors out on the first run. File names, format, and what to change them into: write them out one by one. It's faster than five rounds of edits.

### × Running code you don't understand

The code may have delete or move commands hidden in it. Have AI walk through it line by line. When you hit words like del, rm, move, ask exactly which files it touches and whether you can still get them back. If you can't get a clear answer, don't run it yet.

### × Using live files as guinea pigs

Running against hundreds of live files on the first try means you can't recover if it goes wrong. Copy three or five into a temporary folder first. Get it right there, then run everything. Three extra minutes saves you a whole day.

### × Randomly hacking at errors yourself

For an error you don't understand, copy it verbatim to AI and say what you did and what result you wanted. Letting it fix things is ten times faster than guessing, and it won't break the parts that were fine.

## Checklist

☐ I can say how many times a week I do this task and whether the steps are fixed

☐ My prompt says where the files are, what format they're in, and what they should become

☐ Before the first run I copied samples into a separate folder instead of touching the live files

☐ For code that deletes or moves things, I had AI explain it line by line

# 48
# Handling a Big Batch at Once: Batch Processing

Changing a hundred files one at a time is too slow. Break the work into "the same set of actions, repeated a hundred times", and let AI write that routine into something that runs.

## What this chapter covers

☐ You can break a big job into "handle one" plus "repeat many times"

☐ You know how many to run at a time, and how to keep going after it crashes

☐ You can add "skip it and log it" to a batch job so it doesn't stop everything midway

## 01 Break the Big Job Into: Handle One, Repeat Many Times

A hundred contracts to move onto a new template, a hundred images to watermark. Both sound like big deals. Break them open and there are only two small jobs: get one right, then do that a hundred times.

The second part barely needs you. Repeating things is what computers are best at. You just have to explain the first part clearly, and it runs the other ninety-nine on its own. It's faster than your hands, and it won't miss any.

A dumb way to break it down: take a sheet of paper and write down each step of "fix one", until you don't have to think about it anymore. That sheet is your requirements spec for AI, and it's more reliable than running it through your head once.

- Get one working first, then ten, then everything

- File names need a pattern, like 2024-01, 2024-02, so the script can grab them one by one
- Put the output in its own folder; don't overwrite the originals

## 02 Know How Many to Run at a Time

Don't throw a thousand files in at once. Run ten first and see how long it took and whether the results are right. Ten is fine, then fifty, then a hundred. From the time ten took, you can also estimate how long the full run will need.

Don't walk away while it runs. Plenty of jobs die halfway precisely because the person left and the computer went to sleep. Turn sleep off in the power settings, or just go make a cup of tea and come back.

After the run, check the counts: how many went in, how many came out. If the two numbers don't match, some files got dropped along the way. That's when the failure list is your clue.

**Example code**

```
import os, time

src = r"D:\contracts\raw"
dst = r"D:\contracts\output"
os.makedirs(dst, exist_ok=True)

files = [f for f in os.listdir(src) if f.endswith(".docx")]
total = len(files)
failed = []

for i, name in enumerate(files, 1):
try:
# swap in the "handle one file" code here
print(f"{i}/{total} done: {name}")
except Exception as e:
failed.append(f"{name} -> {e}")
print(f"{i}/{total} skipped: {name}")
time.sleep(0.2)

with open(os.path.join(dst, "failed_list.txt"), "w", encoding="utf-8") as f:
f.write("\n".join(failed) or "all succeeded")

print(f"done: {total - len(failed)} succeeded, {len(failed)} failed")
```

## 03 When Something Fails Midway, Don't Let It Stop the Whole Batch

Out of a hundred files there will always be one or two odd ones: encrypted, empty, with strange symbols in the name. By default, one error and the whole batch just stops there.

Have AI add a layer of protection: skip whatever errors out, log the name in a "failure list", and keep going with the rest. These files are a tiny fraction, but they can jam the entire batch.

After the run, just look at the list. Handling those one or two separately is far less work than starting over. Keep the script that worked for that same job; next time you change two paths and it's ready to go.

- Ask for a failure list that says which file and what the error was
- Don't run the same batch over and over; it gets messier each time
- No errors doesn't mean everything worked. The counts have to match

## A template you can copy

When you hand a big batch of work to AI, describe it like this

```
I want to process the files in one folder, and every file gets the same
treatment:
1. Open the [fill in file type]
2. Do [fill in step one] and [fill in step two]
3. Save to the "output" folder, keeping the file name unchanged

Please write it to these requirements:
- After each file, print "3/100 done: filename" on screen
- If one can't be opened, skip it and log it to "failed_list.txt"; don't stop
everything
- Don't modify the original files
- At the end print one summary line: how many succeeded, how many failed
- For now only process the first 10 so I can look; once I confirm it's fine,
run everything
```

## Common mistakes

### × Going full-scale on the first run

A thousand files, it crashes halfway, and you don't know which ones were changed and which weren't. Run ten first, confirm they're right, then add more.

### × Overwriting the original files directly

Get it wrong and the originals are gone, with no sample left to compare against. Have the script write to a new folder. Check the results, then decide whether to delete the originals.

### × Only checking whether it errored

A program finishing quietly doesn't mean every file succeeded. A hundred went in, ninety-seven came out. You have to find those three one by one.

### × Batching anyway with messy file names

With no pattern in the names, the script easily misses files, and easily changes temporary files it shouldn't touch. Spend ten minutes standardizing the names first; it's much faster than hunting down what got missed afterwards.

## Checklist

☐ I got "one" working before adding the repetition; I didn't go full-scale straight away

☐ The output goes into a new folder, and the original files are untouched

☐ After the run I checked the in and out counts

☐ I had the script skip and log errors instead of stopping everything

# 49
# Hooking AI into a Front-End Page

You want the input box on that web page to actually talk back, which means wiring the page up to a model. This chapter covers three things: what gets sent, where the key lives, and how to show the answer while it's still being written.

## What this chapter covers

☐ Know what actually goes into a single chat request

☐ Know why a key must never sit in front-end code

☐ Can build the 'shows up as it's generated' effect, instead of a dozen-odd seconds of dead waiting

## 01 What the Page Actually Sends When You Hit Send

When you press send, what leaves the page isn't a sentence. It's a small block of text in a fixed format called JSON (a common way for computers to pass data to each other).

It carries three things: which model to use, what was said before, and what's being said now. Beginners often leave out the middle two. Leave them out and you get 'why doesn't it remember what I just told it'.

One more thing that's easy to forget: the model itself keeps no memory. When people say 'it remembers', what's really happening is that you resend the earlier conversation every single time. You have to do that yourself. Nothing is stored on the model's side.

**Example code**

```
// The front end only collects input and shows answers. No key in here.
async function ask(question) {
const res = await fetch("/api/chat", {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({
history: recentTurns, // the last few turns of the conversation
message: question // what the user said this time
})
});
if (!res.ok) throw new Error("request failed");
const data = await res.json();
return data.text; // pull out the body text
}
```

## 02 Never Put the Key in the Front End

Calling a model takes a key (a string of characters, basically the password to your account). Some people write it straight into the page code to save trouble. It really is convenient while testing. In production it's an incident.

Users can see all of your page code. Press F12, or right-click and view source, and the key is sitting right there. Someone takes it, burns through your quota, and you find out the next day when the bill looks wrong.

The right way is to put your own server in between (a server or a cloud function whose whole job is to call the model on the page's behalf). The page only talks to it. The key stays there.

- A key in front-end code is the same as hanging your house key on the doorknob
- Have the server do two things while it's there: cap the length of a single request, and cap how many times one person can ask per minute
- Keep the key in an environment variable. Don't hard-code it, and don't commit it to the repository

## 03 Don't Make People Wait at a Blank Screen: Show It as It's Written

The model pushes text out a piece at a time. With default settings it hands you the whole thing only once it's finished, so the user stares at a spinner for a dozen-odd seconds.

Turn on streaming (have the words arrive piece by piece in real time) and the page shows characters appearing one after another. The wait is the same length. The feeling is completely different.

One more detail: clear the input box and show 'thinking' the moment they hit send. Otherwise users assume the button did nothing, click it five more times, and you pay for every one of those.

- Build the 'generating' state first. It does more than any animation.
- Disable the send button while generating, so people can't click again and again
- When something fails, say it in plain language on the page. Don't dump a wall of English errors on the user

## A template you can copy

When you have AI build this layer for you, say it like this

```
I want to connect a large model to a web page. Please give me code that follows these points:
1. The front end only collects input and displays answers. No keys of any kind in it.
2. Put a server-side endpoint in between. Read the key from an environment variable.
3. Stream the answer back, so the page shows it piece by piece.
4. When a request fails, show a plain-language message on the page and give a retry button.
5. Each request carries only the last few turns. Never an unbounded pile.
6. Finally, write me step-by-step instructions for getting it running locally.
```

## Common mistakes

### × Key written into the front end

Users can see all of your front-end code, so the key is public. Put it on the server. The page only calls your own endpoint, and the key never leaves your machine.

### × Sending the entire chat again every time

The longer the conversation, the more you send each round, and the slower and more expensive it gets. Send only the last few turns, and compress the earlier conversation into a short summary first.

### × No timeout and no retry

One network hiccup and the page hangs on a spinner forever. The user's only option is to refresh. Set a timeout of a dozen seconds or so, and give a 'try again' button on failure.

### × Throwing the raw error at the user

Users can't read English error text. They just conclude your thing is broken. Turn it into 'couldn't connect just now, please try again', and log the details on your own side.

## Checklist

☐ I've checked that there's no key anywhere in the front-end code

☐ When a request fails, the user sees plain language, not a string of English

☐ The send button is disabled while generating, so users can't click repeatedly

☐ I know what context each request carries, and it doesn't grow without limit

# 50
# Hooking AI into a Back-End Service

Connecting AI doesn't mean understanding models. Let the server send the request for you, keep the key hidden, keep timeouts and retries under control. That's the whole job.

## What this chapter covers

☐ Know that connecting AI on the back end is just one more network request, nothing mysterious

☐ Can keep the key on the server, so it never turns up in the front end

☐ Can set timeouts, retries and a fallback, so the page doesn't go blank when the model is down

## 01 Put Plainly: the Server Sends the Request for You

You click a button on the page. The browser sends what you said to your own server. The server passes it on to the model provider, gets the answer back, and returns it to the browser. That extra hop in the middle is what 'connecting AI on the back end' means.

Why not just let the browser call the model directly? Because it has to carry a key. That key is called an API key (the string you present when calling a model). Whoever has it can spend your money.

So there's exactly one rule: the key lives on the server and nowhere else. The browser knows your server's address and never sees the key.

- One call is one HTTPS request. No different in kind from calling a payment API or sending an SMS
- The request holds three things: which model, what you said, and the longest reply you'll accept

- What comes back is a block of text. Where to store it and how to display it is on you

## 02 How to Hide the Key So It Doesn't Leak Out

The most common incident is a key written into front-end code. Front-end code is public, and anyone who opens the browser can see it. That's your bank card PIN taped to the door.

The right way is an environment variable (a separate place on the server for storing secrets, outside the code files). The code reads it at runtime. It doesn't appear in the code itself, not once.

To be safer still: build your own endpoint layer that only allows logged-in users and only allows a fixed handful of actions. Don't build an all-purpose channel where anything you pass in gets asked.

- Keep the key in a .env file, and add .env to .gitignore
- Mask the key when logging, leaving only the first few characters
- Set a spending cap on that key, so a bad day at least can't burn all the way through

**Example code: the back end forwards one request (Python)**

```
# The key comes from an environment variable. No plaintext in the code.
import os, requests

def ask(prompt: str) -> str:
r = requests.post(
"https://your-provider-host/v1/chat/completions",
headers={"Authorization": f"Bearer {os.environ['AI_KEY']}"},
json={
"model": "your-model-name",
"messages": [{"role": "user", "content": prompt}],
"max_tokens": 500,
},
timeout=30,
)
r.raise_for_status()
return r.json()["choices"][0]["message"]["content"]
```

## 03 It Will Go Down. Plan the Way Out First.

Model services time out. They rate-limit you (too many requests in a short window, so the other side simply refuses them). They throw errors out of nowhere. This isn't an exception. It's the normal state of things.

Three things you have to do: set a timeout (give up after 30 seconds, say), retry on failure, and have fallback wording ready. Retry at most twice, with a longer wait each time.

Then add one more limit: one user can ask at most a few times per minute. Otherwise someone hammers your endpoint and the bill gets ugly.

- Don't set the timeout too long. After 30 seconds of waiting, users want to close the page
- Only retry on timeouts and server errors. If the parameters are wrong, retrying a hundred times won't help
- Log the character count of every call. End-of-month reconciliation depends entirely on it

## A template you can copy

Copy this: wrap your calls in a timeout and a retry

```
import time

def ask_with_retry(prompt, tries=3):
for i in range(tries):
try:
return ask(prompt)
except Exception:
if i == tries - 1:
return "The service is a bit busy. Please try again shortly."
time.sleep(2 ** i) # wait 1s, 2s, 4s before retrying
```

## Common mistakes

× **Writing the key into the front end**

Anyone who opens the browser can see it, so it's public. Keep the key in a server environment variable only, and let the front end talk only to your own endpoint.

### × Building one all-purpose endpoint

Anything you pass in gets asked, which is giving the model away for free. Hard-code the role and the task inside the endpoint, and accept only the few fields you actually need.

### × No timeout

If the other side never responds, your process just hangs there, and a few of those and the service falls over. A timeout is mandatory. 30 seconds is a common value.

### × Making users sit through long tasks

Generating a long report takes tens of seconds. Don't leave the page spinning. Return 'submitted' right away, run it in the background, and push a notification when it's done.

## Checklist

☐ The key is on the server only. A search of the front-end code and the repository can't find it

☐ The endpoint does a fixed few things. It isn't a channel where anyone can ask anything

☐ Timeout, retry and fallback wording are all in place, so the page doesn't go blank when the network drops

☐ Character count and elapsed time of every call are logged, so month-end reconciliation works

# 51
# Using AI on Your Phone

Using AI on a phone is about grabbing the moment: talk while you walk, snap a photo of what you're looking at. The key move is saving your usual prompts as shortcuts, so you don't retype them every time.

## What this chapter covers

☐ Know which three kinds of work on a phone are worth handing to AI

☐ Can save common prompts as keyboard shortcuts and pull them up by typing a few letters

☐ Can tell which jobs belong on the phone and which belong back on the computer

## 01 On a Phone, the Valuable Word Is 'Right There'

A sentence comes to you on the subway. You pull out your phone, say it once, and have it smoothed out for you. In a meeting you photograph the whiteboard and have it turned into three conclusions. A computer can't do either of these.

The phone's advantage isn't compute power. It's that you are on the spot. Standing, walking, waiting — it works the whole time. Ideas, things you notice, messages you need to answer on the spot: they all happen on the phone.

So don't do long jobs on the phone. Writing a 5,000-word proposal, reworking a pile of spreadsheets — go back to the computer. The screen is small, and switching back and forth wears you out.

- Speaking is several times faster than typing. For a longer question, just use voice input
- A photo plus 'what is this and how do I use it' is faster than searching for keywords

- Ask it to do one small thing at a time. On a small screen, that's the only way you can read it and fix it

## 02 Save Your Usual Words as Shortcuts Instead of Typing Them

Retyping a long prompt every time is the quickest way to give up on the whole thing. Phone keyboards generally have 'text replacement' or 'shortcuts' (type a few letters and a whole passage expands).

The method is simple: save the prompt, give it a short name. Type gw, for example, and it expands into a fixed set of rewrite instructions. After that you only add the source text.

Five to ten of them is plenty. Cover the work you do most: rewriting copy, writing weekly reports, translating, summarizing meeting notes, polishing emails. Don't get greedy — anything you save but never use, you also never remember.

- Pick letter combinations you don't normally type for the shortcut names, so everyday typing doesn't expand them by accident
- Bake the word count, the tone and the audience into the shortcut, so you don't have to add them each time
- Remember to export them before you change phones. Keyboards generally have a backup option

**Example: a prompt you can save as a shortcut (store it in your keyboard, name it gw)**

```
Rewrite the passage below as a version for the client:
1. Drop internal jargon and use wording the client understands
2. Keep it under 200 characters
3. Keep only the conclusion and what you need the client to do. No build-up.

Original:
```

## 03 The Phone Collects, the Computer Does the Work

The phone is good for starting things: jot a line, take a photo, record a voice memo, throw it somewhere for now. Don't rush to organize

it. At that moment you wouldn't have the energy anyway.

Deal with it all when you're back at the computer. A cloud notes app or your own chat history both work. The point is to keep 'jotting it down' and 'editing it properly' separate.

Don't force long conversations on the phone. The screen is small, scrolling back is a chore, and by the end you can't remember what you said earlier either.

- Use a cloud notes app as a holding bin. Throw things in from the phone, edit on the computer
- Tidy up important results the same day. Don't save it for the weekend, or the details are all gone
- One-off questions: ask and move on, no need to save. Only things you'll use again are worth saving

## A template you can copy

Copy this: an opener for voice questions on your phone. Say it, then go straight into your question

> I'm on the go and dictating this, so there may be typos. Read it for meaning and don't get stuck on the misspellings.
> Here's my question. Give me the conclusion in one sentence first, then three supporting reasons, each under 20 characters:

## Common mistakes

### × Writing long documents on the phone

Small screen, cursor hard to place — editing costs more than writing. On the phone, produce only outlines and short lines. Write the finished draft back on the computer.

### × Not flagging that voice input is messy

Voice input often produces wrong characters, and AI will read them literally. Start with one line: 'there may be typos, go by the meaning'.

### × Treating chat history as a notebook

Once there are many conversations you can't find anything, and it's easy to delete the wrong one. Pick one place for results — a folder in your cloud notes app, for example.

### × Sending sensitive content carelessly while out

Public Wi-Fi isn't necessarily safe. Client materials, contracts and that sort of thing can wait until you're on a network you trust.

## Checklist

☐ I've saved my five most-used prompts as keyboard shortcuts

☐ I know the phone is only for short work. Long documents and spreadsheets go back to the computer

☐ When I ask by voice, I start with 'there may be typos'

☐ Important results go into my cloud notes the same day, not left sitting in the chat, so I can find them next time

# 52
# Working Out What an AI Feature Actually Costs

AI costs money by the word, and you pay for both what goes in and what comes out. This chapter shows you how to use one formula to estimate a single request down to a fraction of a cent, and then how to find the spot that eats the most money.

## What this chapter covers

☐ You can use one formula to work out roughly what a single request costs

☐ You know how input, output and context each affect the bill

☐ You can find three places to save money, and you know which one to change first

## 01 You Pay by the Word, Not by the Call

Model providers charge by token. A token (think of it as a "chunk of text"; one Chinese character usually counts as one or two) is the smallest unit of billing.

Note that both directions cost money: what you send counts as input, what comes back counts as output. And output is generally more expensive than input, often two or three times as much.

So "let it say a bit more" costs money. If all you want is one conclusion, don't let it write eight hundred words. The first cut you make to save money is usually to make it write less. Budget in "words per month", not "money per call".

- A thousand Chinese characters is roughly one thousand to fifteen hundred tokens; estimating at the high end is safer

- Every turn sends the earlier conversation along with it, so the longer the chat runs, the more the input costs
- Images and long documents cost money too; one image often costs as much as several paragraphs of text

## 02 A Fraction of a Cent per Request: Do the Math Yourself

There is just one formula: cost = input tokens × input price + output tokens × output price. You can find the prices in your provider's dashboard. Copy them down and plug them in.

Here's an example, and the numbers are made up. Input costs 1 yuan per million tokens, output costs 4 yuan per million tokens.

Take one request with 2000 tokens of input and 500 tokens of output. Plug those in and you get 0.002 yuan plus 0.002 yuan, which is 0.004 yuan.

Four tenths of a fen per call. Ten thousand calls a day and that's forty yuan. A month, and it's twelve hundred.

- Estimate "how many calls a day" first, then multiply by the unit price. That's more useful than staring at the unit price
- Output length is the easiest thing to overlook; capping max_tokens saves a big chunk
- Drop the two numbers into the code below and run it to get the answer

**Example code: plug in the prices and get your monthly bill**

```
INPUT_PRICE = 1.0 # price per million input tokens, use the one from your
dashboard
OUTPUT_PRICE = 4.0 # price per million output tokens

in_tok, out_tok = 2000, 500 # roughly how much one request uses
times_per_day = 10000 # how many calls a day

per_call = in_tok / 1e6 * INPUT_PRICE + out_tok / 1e6 * OUTPUT_PRICE
print(f"one call costs {per_call:.4f} yuan, about {per_call * times_per_day *
30:.0f} yuan a month")
```

## 03 The Three Places That Save the Most, Starting with the Biggest Cost

First, cut the context. Don't stuff the whole document in every time; pull out just the relevant passages first. This one usually saves the most.

Second, switch to a smaller model. For work like fixing typos, sorting things into categories, or pulling out fields, the cheap tier is plenty. Save the expensive tier for the genuinely hard questions.

Third, caching (storing results you've already computed, so the next identical question can just take them). For repeated questions and shared reference material, one stored copy is enough. Every cache hit saves you one call's worth of money.

- Look in your logs to see which kind of request is most common, and start with that one
- Cap the output length and the model won't wander off
- Change one thing at a time, then look at the bill again, or you won't know how much you saved

## A template you can copy

Copy this: have the AI check your cost estimate for anything you missed

```
I'm estimating the cost of an AI feature. Please check my numbers for anything I've left out.
What I know:
- one request uses about ___ input tokens and about ___ output tokens
- ___ calls a day
- input price ___ yuan per million tokens
- output price ___ yuan per million tokens
Please do three things:
1. Work out the cost of one request and of one month (30 days).
2. Point out the cost items I may have missed.
3. Give me three concrete changes I can make right away to lower the cost, and say roughly what share of the bill each one saves.
```

## Common mistakes

### × Counting output but not input

Input is usually far bigger than output, especially in a long conversation. Count both ends, don't just stare at the few words of the answer.

### × Stuffing the whole document in

Tens of thousands of characters every time means the money goes on repeated content. Pull out the relevant passages first, or use retrieval to pick them for you.

### × Using the most expensive model for everything

Using an expensive model for simple work is like driving a truck to buy groceries. For categorizing, extracting fields and fixing typos, a small model is enough.

### × Finding out the bill exploded after launch

Before launch, run a hundred real requests and estimate from the actual counts. After launch, check usage every day and set an alert line.

## Checklist

☐ I can use one formula to work out roughly what one request and one month cost

☐ I've counted both input and output, and I've capped the output length

☐ I know which kind of request costs me the most

☐ I ran real requests before launch, and I check usage every day after it

# 53
# The Checklist You Have to Walk Through Before Launch

An hour before launch, walk through this list. Keeping the most expensive kinds of mistake outside the door saves you ten times the trouble of putting out fires afterwards.

## What this chapter covers

☐ You can put together a pre-launch checklist that's actually useful for the project in front of you

☐ You know which mistakes can't be taken back once they happen, and which ones have to be blocked first

☐ You can get the AI to go through the list with you item by item, instead of settling for one "looks fine to me"

## 01 First, Get Clear on Which Mistakes Can't Be Undone Once You're Live

Some mistakes you can just fix. Others can't be taken back once they happen. Pick out the second kind first, and your checks will have a focus.

Deleting a user's data, shipping test configuration to production, hard-coding a key into the code. For these three, an apology afterwards doesn't put anything back.

Take a sheet of paper and write three lines: what's the worst thing that could be lost, who's the worst person who could see it, how long would it take to repair. Once you've written them, you'll know where to look first. Nobody else needs to see those three lines.

- Whether data can be recovered matters far more than whether the interface looks good
- Any action that "goes out with one click" needs one more confirmation step
- Check the things that can't be undone first, the fixable ones after

## 02 Make the AI an Inspector, Not a Rubber Stamp

A lot of people ask the AI one question before launch: "do you see any problems?" Most likely it comes back with "overall it looks fine". That isn't a check, that's asking for reassurance.

The right way is to hand it a specific list and have it answer one item at a time. If it can't answer one, it should write "I can't check this one". No fudging.

Ask rigid, literal questions. Things like "is there a hard-coded password" and "what will the user see when something fails". The more specific the question, the less room it has to brush you off.

- Ask one item at a time and finish it before moving on; don't dump twenty on it at once
- Make it name the file or the page. If it can't point to a location, it hasn't checked properly
- When it says "no problem", follow up with "what makes you say that"

## 03 Walk Through It Yourself; Don't Just Read the Code

The AI saying it's good doesn't make it good. Be a new user yourself. Try it on a phone, with someone else's account, with the network cut off.

Walk three paths. The smoothest one: from signing up to actually using the thing. The one most likely to go wrong: wrong password, an oversized file, the connection dropping halfway. The one that should never work: opening the admin pages without logging in.

For each step, note three things: what you clicked, how many seconds you waited, what you saw. Past three seconds with nothing happening, the user is already gone.

- Try it on someone else's phone. Yours is full of logged-in sessions, so you won't see the problems
- Upload a very large file and cut the network. Those two moves shake out most of the faults
- Set aside half an hour to go through it calmly, instead of remembering one minute before launch

**Example command: search for hard-coded keys before launch**

```
# run this in the project folder to see whether passwords or keys are written straight into the code
grep -rn "sk-\|api_key\|password" ./src
```

## A template you can copy

Send this to the AI before launch, so it checks item by item and admits what it can't check

```
You are my pre-launch inspector. My project is: (one sentence on what it does and who it's for).
Go through the five items below one at a time. For each one answer only three things: is there a problem, where is it (file or page), and how should I fix it.
1. Is there a hard-coded password, key or backend address anywhere in the code?
2. Is there fake data left in the database from testing?
3. What does the page show when the user enters something invalid, when the network drops, when someone uploads an oversized file?
4. Can someone who isn't logged in open the admin pages directly?
5. If something goes wrong, how do we roll back to the previous version? Write the steps out clearly.
For anything you can't check, just write "I can't check this one, you need to confirm it by hand". Don't guess.
```

## Common mistakes

### × Only ever tried it on your own machine

Your machine has cache, has login state, has every dependency installed. Try a different machine, a different browser, mobile data. This is where problems usually surface, and they surface in bunches.

### × Relaxing because the AI said "no problem"

It can't see your runtime environment or your real data. All it can judge is whether the code looks right. Make it point to specific locations, then go click through it yourself. Don't skip that step.

### × Test data mixed in with real data

Fake orders and fake users left over from testing can get treated as real after launch, and messages can go out to actual people. Clean it up before launch, or just use a separate test database.

### × Leaving yourself no way back

Before launch, confirm how you'd get back if something goes wrong. Is the old version still there, can the data be recovered, who can shut it off with one switch. If you haven't worked out the way back, don't launch yet. A day later costs you nothing.

## Checklist

☐ I can say what the worst case is, and I know how to recover from it

☐ There are no plaintext passwords or keys in the code; configuration and code are kept apart

☐ I walked the main flow myself on a phone, with a fresh account, and with the network off

☐ If something goes wrong I know how to roll back to the previous version, and I've done it once with my own hands

# 54
# How to Keep Improving After Launch

Launch isn't the finish line. Pick three numbers, review once a week, change one thing at a time, and your product gets smoother to use while people stick around.

## What this chapter covers

☐ You know which three numbers to watch after launch, so a pile of reports doesn't drown you

☐ You can pick, out of a pile of complaints, the one thing that really should change this week

☐ You can use "small change plus before-and-after comparison" to confirm whether a change actually helped

## 01 Pick Three Numbers First; Don't Try to Watch Everything

The easiest mistake to make after launch is wanting to see everything and ending up seeing nothing. Pick three numbers. Don't add a fourth.

One for whether anyone shows up (say, how many people open it each day). One for whether anyone actually succeeds with it (say, the share who get through the main flow). One for whether anyone is complaining (say, error counts, refunds).

The numbers don't need decimal places. Note them down at the same time every day and after a week the direction shows up. The table doesn't need to be pretty; notes on your phone are fine.

- You have to be able to read the three numbers once a day, in under five minutes
- Record the numbers, not your feelings: "today felt smooth" doesn't count

- When a number gets worse, don't rush to change things. First check whether this week was unusual, a holiday or a promotion

## 02 Pick One Complaint; Don't Change Ten Things at Once

Changing everything users mention is the same as changing nothing. Sort the complaints by how often they come up, touch only the top one, and write the rest down for later.

Change one thing, then watch for a week. Change five at once and even when something works you can't say which one did it. You won't even know when you've wasted the effort.

Before you change anything, copy down the current numbers. Without a before-and-after, you can never tell whether you improved it or just had a lucky week.

- If three people raise the same problem, it's worth putting on this week's list
- If one person raised it and it needs a big rewrite, note it down and leave it
- The smaller the change the better: if copy will do, don't touch the flow; if the flow will do, don't touch the structure

## 03 Set Aside Half an Hour a Week to Review with the AI

The review doesn't need to be long. Throw the week's numbers, the users' own words and what you changed at the AI together, and let it help you find the pattern. Half an hour is enough.

The key is real material. If all you say is "things haven't been great lately", all it can give back is a pile of correct-sounding filler that looks like advice and not one line of it is usable.

Ask it for three possible causes and judge for yourself which one fits best. Don't let it hand you the plan. You decide the plan, because only you know what cards you're holding.

- Copy the users' complaints word for word. Don't summarize them first, because summarizing sands off the key details
- Land the review on one sentence: next week we change only this

**Example: a five-row table per week beats any dashboard**

```
Date | Opens | Completed main flow | Errors | What changed this week
Mon | 120 | 43 | 7 | changed the button copy on the signup page
Tue | 96 | 38 | 9 | nothing
Wed | 141 | 61 | 4 | nothing
Thu | 133 | 58 | 3 | nothing
Fri | 150 | 66 | 3 | nothing
```

## A template you can copy

A prompt for the weekly review; just fill in your material

```
Here's how my week went. Please help me review it.
Project: (one sentence on what it does and who it's for)
The three numbers: opens ___, completed main flow ___, errors ___ (last week they were ___, ___, ___).
What I changed this week: (be specific, e.g. changed the button copy on the signup page)
The users' own words (copy them, don't summarize):
1.
2.
3.
Please do three things:
1. Point out the most abnormal change among the numbers, and say which step might be going wrong.
2. Group the users' words into at most three kinds of problem, ordered by how often they came up.
3. Give three possible causes, and for each one spell out "if that's really what's happening, which number should I watch next week to check it".
Don't give me a pile of suggestions. Give me the one place most worth touching next week.
```

## Common mistakes

- × **Launch it and walk away**

Launch is only the start. The first two weeks are the densest with problems; spend five minutes a day on the numbers and on the errors. Finding out after users have already left costs far more than five minutes a day.

### × Changing too many places at once

Change five places and when it goes well you don't know why, when it goes badly you don't know why either, and next time you're guessing. One place at a time, one look per week. That's how improvements accumulate.

### × Only looking at averages

An average of 100 a day could be 600 on Monday and almost nobody the rest of the week. An average hides the worst day. Look at each day's number on its own.

### × Taking users' politeness at face value

A user saying "it's pretty good" doesn't mean it's usable. They may just be too polite to say otherwise. What's actually useful is where they got stuck and what they clicked that did nothing. Watch behavior, don't just listen to what people say.

## Checklist

☐ I watch only three numbers, I can read them in five minutes a day, and I don't reach for more

☐ I change one place at a time, and I copy down the current numbers before I do

☐ I can name the single problem that came up most this week

☐ What I keep is the users' own words, not my own summarized version

Part Seven

# Hands-On Scenarios

Chapters 55 — 66

# 55
# Using AI to Write Weekly Reports and Office Documents

The writing you do every week eats the most time, because it isn't hard, just annoying. This chapter turns weekly reports, meeting minutes and briefing materials into fill-in-the-blanks: pile the raw material in first, then have it assemble things in your format.

---

## What this chapter covers

☐ You can write a weekly report your manager will actually read, in ten minutes

☐ You can turn scattered meeting notes into minutes you can send out as-is

☐ You know which parts AI writes better than you, and which parts you have to fill in yourself

## 01 Weekly Reports Are Hard Because You Don't Know What Your Manager Wants to Know

Most people start a weekly report from "what I did this week", and end up with a running log. The person reading it actually wants three things: how far along things are, what the risks are, and what happens next week.

So don't open with "write my weekly report for me". First list what you did this week as a running log, one item per line. Don't organize it, just get it all down. That's your raw material.

Then tell it who the reader is and what they care about, and have it rearrange the material into three sections: progress, risks, next week's plan.

- Write enough of that log. Give it seven items and it has something to choose from; give it three and it has to invent the rest
- Write "what got done" as a sentence with an outcome. "Researched the options" is weaker than "went with option B, integration starts next week"
- The risks section is not optional. A weekly report with no risks reads like nothing is moving

## 02 Meeting Minutes: Record What Matters First, Organize Later

In a meeting you only need to note three things: who said what conclusion, what got decided, and who is responsible for what. You don't need the discussion in the middle. That's process, not outcome.

When the meeting ends, throw those scattered sentences at it and have it sort them into four columns: conclusions, to-dos, owners, deadlines.

A recording works too. Run it through a transcription tool first, then paste the text in. Transcription typos don't get in the way of understanding, so you don't need to proofread every word.

- Every to-do needs a person and a time on it. "Follow up on this" means nobody touches it before the next meeting
- Have it add a separate section for "things left undecided". Those are the ones most worth chasing
- Read it once yourself before sending: did any line get attributed to the wrong person?

## 03 Briefing Materials: Settle the Conclusion First, Then Lay Out the Evidence

With material for your boss, the worst thing you can do is spend three pages building up to the conclusion. The right order is: page one states the conclusion and the recommendation, and the evidence comes after.

Have AI compress your conclusion into a single sentence first. If you can't write that sentence, you haven't thought it through yet. Don't start on the material.

Once the conclusion is set, have it lay things out in four sections: conclusion, evidence, risks, what support you need.

- One point per page. Cram three ideas onto one page and none of them stick
- Numbers need a date and a source. "Up 30% last month" is far more useful than "up a lot"
- Have it restate your conclusion in your manager's own words, and see whether the meaning survived

### Weekly report prompt, ready to copy

```
Here is what I did this week. It's a running log, not organized:
<paste your running log here>

Please write a weekly report that meets the requirements below:

Reader: my department manager. He only cares about progress, risks and next week's schedule. He doesn't read process details.
Length: three sections, no more than four lines each.

Section one, progress: merge what I finished this week into two or three items. Write each one as "what I did, what came of it".
Section two, risks: list what is currently stuck or might slip. For each one, say where it's stuck and whose help is needed. If there really aren't any, write "no risks this week".
Section three, next week: list the three things to push forward next week, one sentence each.

Requirements:
- Don't write empty phrases like "actively driving forward" or "attaching great importance".
- Don't invent anything I didn't mention in my log.
- If you think something should be added but you don't have enough information, list it separately and ask me.
```

## A template you can copy

Meeting minutes prompt, ready to copy

```
Here are the scattered notes from this meeting. The format is a mess; ignore that:
<paste the notes here>

Please organize them into four parts:

1. Conclusions: what was decided in the meeting, one sentence per item.
2. To-dos: who, does what, done by when.
3. Undecided: things that were discussed but not settled. Say clearly which disagreement they're stuck on.
4. Needs my confirmation: places where the notes are unclear. List them and ask me.

Requirements:
- Only organize what I wrote down. Don't add information you didn't see.
- Keep names as they are. Don't replace them with "a colleague".
- If a to-do is missing a deadline or an owner, mark it "to be confirmed" and flag it in part 4.
```

## Common mistakes

### × Asking it to write the report from nothing

It can only invent. The result looks presentable, but not one line matches what you actually did, and fixing it takes longer than writing it yourself. Always feed it material first.

### × Giving it only three items

With too little material it starts filling in details, and you have to check every one. Write more items than you need and let it pick.

### × Skipping the risks section

A weekly report with no risks tells the reader the work is easy or that you aren't pushing it. Say when you're stuck, and say it louder when you need someone's help.

### × Sending it straight after the cleanup

It may merge two things into one, or attribute someone else's conclusion to you. Read it line by line before you send it. That's two minutes.

## Checklist

☐ I listed what I did this week as a raw log first and had it organize that, instead of asking it to write from nothing

☐ I told it who the reader is and what they care about, instead of just saying "write my weekly report"

☐ I told the truth in the risks section, naming where things are stuck and whose help I need

☐ I read it line by line before sending, and checked for invented content and misattributed quotes

# 56
# Using AI for Data Analysis: Asking a Spreadsheet for Answers

AI is bad at arithmetic but good at writing code that does arithmetic. This chapter shows how to hand it a spreadsheet, ask the question you actually have, and get an answer whose numbers are right.

## What this chapter covers

☐ You can describe a spreadsheet's structure well enough that it writes correct analysis code on the first try

☐ You can use "it writes the code, you run it" to avoid arithmetic errors

☐ You can tell which conclusions are usable and which only look like conclusions

## 01 Start by Describing What the Table Looks Like

A person understands a spreadsheet from one glance at the header row. AI can't see your table. It only knows what you type out for it.

So step one is to describe the structure: how many columns, what each one is called, what type it holds, and whether there are blank cells. Three to five lines is enough.

The least work is to paste the first few rows straight in. Keep the header row, strip anything sensitive.

- Say what the types are: this column is a date, this one is an amount, this one is a category. Guess the type wrong and everything downstream is wrong
- Say where the blanks are. It may read a blank as 0, and the average comes out wrong

- If the data is large, give only the structure and the first twenty rows. Don’t paste the whole table in

## 02 Don't Let It Do the Math in Its Head; Have It Write Code

Arithmetic is a weak spot for large models. Adding and subtracting three numbers is fine. Averaging hundreds of rows, it gets it wrong often.

The right way is to have it write code, Python or SQL, then run it yourself and let the machine do the math. AI writes the calculation steps; the machine calculates.

Then throw the results back to it and have it explain them. The numbers come out accurate, and the explanation comes out in plain language.

- Ask for code that computes the answer, not for the number itself. A number it produces off the top of its head is usually invented
- Have it print the intermediate steps too: total rows, rows left after dropping blanks, how many records per category. If the counts don’t line up, something is wrong
- Ask the same question on the same data twice. If the two numbers differ, it was guessing

**Example prompt: have it write code instead of doing the math in its head**

```
I have a spreadsheet. Its structure is:
<paste the column names and types, or the first 3 rows of data>

What I want to know: <your question>

Please answer like this:
1. First write Python code that computes the answer, using pandas. I'll run it myself.
2. In the code, add these checks and print the results: total row count, row count after dropping nulls, how many records each key category has.
3. After the code runs, explain what the results mean.

Note:
- Don't tell me the answer number directly. I don't have your run results, and I don't trust numbers you calculated.
- If my question can't be answered with this data, say so plainly and explain what data is missing.
```

## 03 Conclusions Need Evidence

"Conversion dropped" is useless. What's useful is "conversion fell from 3.1% last month to 2.4%, and the drop is concentrated in the new-user segment".

Have every conclusion carry a number, something to compare against, and a time range. If a conclusion can't produce all three, keep it out of the report.

One more thing: have it list "the questions this data can't answer". That's far more useful than forcing an answer.

- Correlation isn't causation. "People who buy A also often buy B" doesn't mean pushing A will sell B
- Don't trust a segment conclusion drawn from a tiny sample. Three people out of a hundred is noise, not a trend
- Have it state the boundary where the conclusion holds: under what conditions it stops being true

## A template you can copy

Conclusion review prompt, ready to copy

```
Below are conclusions I drew from the data. Please review each one.
<paste the conclusions here>

For each conclusion, check it against these four points and report:

1. Is there a specific number? If not, mark it "missing number".
2. Is there something to compare against? If not, mark it "missing comparison".
3. Is there a time range? If not, mark it "missing time range".
4. Is this conclusion correlational or causal? If the text claims causation but the data only supports correlation, flag it.

Finally, list the questions this data can't answer but a reader might ask.

Requirements: be strict. I'd rather you say a conclusion doesn't hold than say "there's something to it".
```

## Common mistakes

### × Pasting the whole table in

It's slow, it misses things, and sensitive data can leak. Column names, types and the first few rows are enough.

### × Asking "what's the average" outright

It's doing the math in its head, and on hundreds of rows it will be wrong. Switch to having it write code, run it yourself, and let the machine calculate.

### × Trusting numbers it produces on the spot

Every number it gives has to be reproducible from code or from the raw data. If you can't reproduce it, treat it as unsaid.

### × Treating correlation as causation

Two things moving together doesn't mean one causes the other. Say in the report whether it's correlation or causation. Don't blur it.

## Checklist

☐ I described the column names, types and blank cells clearly

☐ When I needed numbers, I had it write code and ran it myself before using the result

☐ Every conclusion carries a number, a comparison, and a time range

☐ I know what this data can't answer, and I wrote that into the report

# 57
# Handing Images, Audio and Files to AI

Today's models can look at images, listen to audio and read PDFs, but they don't look the way people do. This chapter covers how to hand things over, how to ask, and what they most often get wrong.

## What this chapter covers

☐ You know what's worth handing over as an image and what should be converted to text first

☐ You can write a prompt for answering questions about an image, instead of just saying "take a look at this"

☐ You know which kinds of images it misreads most, and can avoid them up front

## 01 It Can See, but Only Roughly

A model doesn't study an image detail by detail the way a person does. It first compresses the whole image into a rough description, then answers you from that description. Big shapes it gets right. Small details it often misses.

The rule of thumb: few objects, large text, clear contrast, and it's reliable. A dense image, handwriting, glare, a tilted photo, and it starts making things up.

So before you hand over an image, ask yourself: what are the three most important pieces of information in it? If they're buried in small print, crop them out and give those separately.

- One question per image. Ask five at once and accuracy drops
- Have it describe what's in the image before it answers your question. If the description is wrong, you catch it immediately

- For a table that happens to be in an image, convert it to real table data first. That's far more accurate than having it read the picture

## 02 Transcribe Audio to Text First

Most models can listen to audio directly, but for everyday work, transcribing first is more reliable. Transcription is accurate these days, and once it's text you can check it yourself.

Once it's transcribed, have it summarize and pull out the action items. Don't worry about filler words and repetition; tell it to skip them.

There's only one case where handing over audio directly is worth it: you want it to hear the tone, say to judge whether the other person is unhappy. That case is rare.

- Add "mark any word you aren't sure about with a question mark" to the transcription step, and you can pick out what to check
- Cut long meeting recordings into chunks first, around ten minutes each. One long stretch and it tends to skip the middle
- When the speakers are hard to tell apart, say up front that three people are talking. Otherwise it mixes up who said what

## 03 Don't Dump the Whole File In; Look at Its Structure First

You can hand over PDFs, Word files and Excel files directly. But with many pages and dense tables, it reads slowly and inaccurately, and it may answer after reading only the first half.

Look at the structure first: how many pages, how many chapters, whether there are lots of tables and figures. Then decide whether to give it the whole thing or just the relevant chapters.

For scanned files, the PDFs made out of images, first check whether you can select the text. If you can't, it's an image, and you need to run text recognition on it first.

- Put the key requirements in the first three lines. On a long file, requirements further down tend to get ignored
- Have it give you the table of contents or a chapter list first. Getting that right is the proof it actually read the file
- For files involving contracts, amounts or dates, re-check the important items yourself

### Example prompt: asking about an image

```
What I want to ask about this image is: <your question>

Please answer in this order:
1. First describe in one sentence what the image is overall.
2. Then answer only my question above.
3. If you see something in the image but aren't sure about it, list it separately and mark it "uncertain".

Note:
- Don't infer information that isn't in the image from general knowledge.
- If the image is too blurry or the key information is unreadable, say it's unreadable. Don't guess.
```

## A template you can copy

Long document prompt, ready to copy

```
Let me describe this file first:
- Roughly <page count> pages, <chapter count> chapters in total
- What I care about is chapter <chapter number> through chapter <chapter number>
- My question is: <your question>

Please proceed like this:

1. First list the chapter titles you can see. For any part you can't list, say you didn't read it.
2. Answer only within the chapters I specified. Don't let anything outside that range creep in.
3. After each answer, note which page or which section the information comes from. If you can't, that tells me you didn't actually locate it.

Requirement: don't fill gaps in the file with general knowledge.
```

## Common mistakes

### × Cramming one image with questions

The more questions, the fuzzier it gets. Usually the first two are right and the rest are invented. One question per image, then move to the next one.

### × Using a screenshot instead of copying the text

If the text can be copied, copy it. It's faster and exact. Use screenshots only when the layout itself carries meaning, like reviewing a design mockup or the formatting of an error message.

### × Having it read handwriting and dense tables

These are the two things it misreads most. Retype the handwriting yourself, export the dense table as data. Either takes less time than fixing it afterwards.

### × Assuming it read the whole file

On a long file it may have processed only the beginning. Have it list the chapters and cite page numbers in its answers, and you'll know what it actually read.

## Checklist

☐ I give one image or one file at a time, and ask one question

☐ I have it describe first and answer second, instead of just throwing the question at it

☐ When the key information is buried in detail, I crop it out or convert it to text first

☐ I know it's unreliable on handwriting, tightly packed tables and blurry images, so I check those myself

# 58
# Using AI for Translation: Accurate and Still Human

Machine translation stands out at a glance. The problem usually isn't the words but the sentence structure and the tone. This chapter covers how to get a translation that reads like a person wrote it, and how to check it afterwards.

## What this chapter covers

☐ You can write prompts that produce natural translation instead of word-for-word conversion

☐ You can pin the style down with a glossary and a description of the reader

☐ You know what to check first after translating, and which errors do the most damage

## 01 Say Who It's For First

The same Chinese paragraph, translated once as a report for your boss and once as an introduction for a customer, comes out completely different. If you don't say which, it falls back to the safest written register.

So the first line has to state three things: which language, who reads it, and what occasion it's for. Those three settle half the style.

Then add a line on register: casual or formal, whether to keep the original terms, whether to use honorifics in forms of address.

- Say how much the reader knows. Experts and general readers need completely different vocabulary

- Name what must not be translated literally: idioms, slogans, product names, job titles
- Tell it which terms have fixed translations. A glossary table is the least work

## 02 Have It Understand First, Then Translate

Long Chinese sentences are the main cause of translation wrecks. One sentence carrying three or four ideas, translated literally, becomes something nobody can read.

Have it restate the meaning in its own words. Once you've confirmed it hasn't gone off track, have it translate. One extra step, half the errors.

Or just require this: break it into short sentences first, then translate, twenty words per sentence at most.

- Where something is ambiguous, have it list both readings and let you choose. Don't let it quietly pick one
- Say up front whether the names of people, companies and products stay in the original or get transliterated
- Specify number and date formats. Chinese orders year-month-day differently from English, and mixing them causes trouble

**Example prompt: include the reader and the register**

```
Translate the Chinese below into English.

Reader: overseas customers evaluating our product. They understand the business but not Chinese.
Occasion: the product introduction on our website. Professional tone, not stiff.

Translation requirements:
1. Break the sentences up before translating, at most 20 words per sentence.
2. Don't translate word for word. Chinese short clauses and parallel structures read like slogans in English. Turn them into ordinary declarative sentences.
3. Use these fixed translations for these terms: <glossary>.
4. Fill in the subjects Chinese leaves out. Don't make the reader guess who is doing what.

Source text:
<paste the Chinese source>
```

## 03 How to Check the Translation

The most effective move isn't asking it to retranslate. It's asking it to only find faults. Ask it to rewrite and it changes everything; ask it to find faults and it can point out which sentences don't sound native.

Concretely: send the translation back and ask "which sentences read like translationese, what did the source mean, and how would you fix them". Have it suggest changes, and you decide whether to take them.

Finally, go through four things yourself: numbers, negatives, conditions, time. These four are where translation goes wrong most often, and the errors are serious.

- Have it flag the sentences it isn't sure about, and put your own checking there
- Going from Chinese to English, check the subjects. Chinese drops them often, and English readers can't follow if they aren't filled in

- Negatives and conditions most easily flip. Compare not, unless and only if line by line against the source

## A template you can copy

Translation review prompt, ready to copy

```
Below is a Chinese source text and its English translation. Please review it only. Do not rewrite the whole thing.

Output three parts:

1. Sentences that read like translationese: list the sentence and your suggested rewrite, and explain why the original is unnatural.
2. Places where the meaning may have drifted: numbers, negation, conditions, time, missing or repeated subjects. Go through each one against the Chinese source.
3. Sentences you aren't sure about: list them and say what you're unsure of.

Requirements:
- Don't change the meaning in order to make it prettier.
- If the translation has no real problems, say so. Don't manufacture edits.

Chinese source:
<paste the source>

English translation:
<paste the translation>
```

## Common mistakes

### × Just saying "translate this for me"

All you get is the flattest written register, and it reads fake at a glance. Add the reader, the occasion and the register, and the quality jumps a level.

### × Letting it rewrite the translation

It turns your translation into a different piece. The problems and the style all change. Switch to "find faults, suggest fixes" and keep the

decision with yourself.

### × Not checking numbers and negatives

"No less than" becoming "less than", "30%" becoming "three times". These errors are the most damaging. All four of these must be checked line by line against the source.

### × Translating long sentences literally

One Chinese sentence carrying three things, translated literally, becomes a string of clauses and the reader is lost halfway. Break it up first, then translate.

## Checklist

☐ I stated the target language, who reads it, and what occasion it's for

☐ I gave a glossary, or said which terms stay in the original

☐ I had it restate its understanding before translating, or required it to break sentences up first

☐ After translating, I checked all four line by line: numbers, negatives, conditions, time

# 59
# Using AI to Understand Your Whole Project

Dumping an entire repository on AI is a waste. It can't read it all, and it wouldn't read it accurately. This chapter covers the map first, details second approach that actually gets it to understand your project.

## What this chapter covers

☐ You can have it sketch the project structure first and confirm it grasped the big picture

☐ You can feed it just the few relevant files instead of the whole repository

☐ You know how to get it to admit plainly that it hasn't seen a particular part

## 01 Have It Draw a Map First

When you take over a new project, you start with the directory structure. AI is the same. Without the structure, all it can do is guess.

Step one, give it three things: the directory tree, the README, and the config files. Then ask one question: what does this project do, and how does it start up.

If it answers correctly, it understood the broad shape. If the answer is vague, either you didn't give enough information or it never read it all.

- Two or three levels of the directory tree is enough. Deeper than that and it loses the main points

- Ask straight out: which file does this project start executing from. That answer reveals whether it really understood
- Have it point out which directories are auto-generated and can be skipped, and you'll know whether it actually read anything

## 02 Feed It Only the Few Relevant Files

A mid-sized project runs to hundreds of thousands of lines. It won't fit. And even if it fits, relevant and irrelevant code mixed together means it can't grab hold of anything.

The right order is to locate first: which files does the feature you're changing touch. Search for keywords. Finding three to five files is enough.

Give it those files plus one line: this feature only involves these files. Narrow the scope and the quality of the answer changes immediately.

- Have it say which files you'd still need to see to change this feature. Then go confirm each one yourself
- Give the call chain too: who calls this function, and what this function calls
- Test files are more useful than comments. Tests spell out what the feature was supposed to do

## 03 Get It to Admit What It Doesn't Know

AI has a bad habit. It makes things up even for parts it never saw. Say "this project has a cache layer" and it assumes there is one, then starts writing cache code.

So every time, add one line: answer only from the files I've given you, and mark anything you haven't seen.

Whatever it marks is the file you need to supply next time. This habit heads off a lot of trouble where it rewrites a function that doesn't even exist.

- Have it tag each conclusion with its basis: which file, which section. If it can't, it guessed
- Ask directly which unseen parts would make its answer unreliable, and it can list them
- For questions about configuration and deployment, make sure it has seen the actual config files before it answers. It can't guess those

**Example prompt: have it understand the project first**

```
Here is information about my project. For now, do only one thing: understand
it.

Directory structure:
<paste a two-level directory tree>

README and config files:
<paste the README and the main config>

Please answer:
1. What does this project do? In three sentences.
2. Which file does execution start from? Once a request comes in, what is the
processing order?
3. Which directories are auto-generated and don't need reading?
4. In the information above, where did you guess? Mark those places.

Requirement: answer only from what I've given you. For parts you haven't seen,
say plainly "not provided". Don't fill in from general experience.
```

## A template you can copy

Pre-change scoping prompt, ready to copy

```
I want to change this feature: <describe the feature>

Here are the files I'm providing:
<file 1, file 2, file 3>

Please answer in order:

1. How is this feature implemented right now? Go in call order and name the file and function at each step.
2. To change this feature, what other files do you need to see? List them and say why.
3. What might this change affect? Include callers, downstream code, and tests.
4. Among the parts you haven't seen, which ones would make your answer unreliable?

Requirements:
- Use only the files I provided as your basis.
- For files you haven't seen, say "not provided". Don't speculate about how they're implemented.
```

## Common mistakes

### × Shoving the whole repository in

It won't fit, and even if it did, the point gets lost. Locate a few files yourself first, then hand those over. It's an order of magnitude more efficient.

### × Asking without giving the directory tree

It doesn't know where files live, so it guesses from file names, and it guesses wrong often. One minute to paste a directory tree saves half an hour of back and forth.

### × Believing it when it says "I understand"

It will say it understands at any moment. Test it with specific questions: where execution starts, who calls this function. If it can't answer, it didn't understand.

### × Letting it change files it never saw

It will edit against an imagined implementation, and the result won't match your project. Before it changes a file, give it that file.

## Checklist

☐ I gave it the directory tree and config files first, and had it draw the project map

☐ I located the relevant files myself and fed it only those, not the whole repository

☐ I required it to cite its sources and to say where it was guessing

☐ Before it changed a file, I confirmed it had actually seen that file

# 60
# Using AI for Code Migration and Upgrades

Moving old code to a new framework, or an old version up to a new one, is where AI helps most and also does the most damage. This chapter covers how to migrate safely and how to roll back when something goes wrong.

## What this chapter covers

☐ You build a mapping table before touching anything, instead of letting it do the whole thing in one go

☐ You can migrate one file first to calibrate the style

☐ You know what's most dangerous in a migration, and how to stop it up front

## 01 Build the Mapping Table First

A migration is essentially swapping old patterns for new ones. So before any code is written, have it produce a mapping table: which new form each old form maps to, what changed in the arguments, what got dropped entirely.

Go through that table item by item. Start only once you understand it. If you don't, go read up first. If you can't check what it changes, the change was wasted.

The table has another use: it's your acceptance checklist. When the change is done, tick off the items one by one.

- Have it list the things with no direct equivalent separately. Those are the parts that eat the most time
- Say exactly which version you're going to. Adjacent versions and a jump across major versions are very different jobs

- Have it mark which downstream call sites each change affects

## 02 Migrate One File First

Don't let it migrate everything at once. Pick a file of middling complexity that has test coverage, and have it change only that one.

After the change, run the tests, look at the code style, and look at how it handled the items with no easy mapping. Once you're satisfied, have it migrate the rest the same way.

This step isn't about saving time. It's calibration. Once you know how it changes things, you can let go on the rest.

- Have it summarize this migration's rules in a few lines. Paste them in on every later file so the style stays consistent
- Commit after each file. When something breaks you can roll back precisely instead of starting over
- Run the tests as soon as each file is done. Don't save them all for the end

## 03 The Real Danger Is Behavior Changing Quietly

Code that compiles and tests that pass don't mean behavior stayed the same. Old code may contain odd but necessary edge-case handling, and the new form quietly "fixes" it away.

The defense is an equivalence test: run the same batch of inputs through the old and new versions, and the outputs have to match exactly.

Have it write that comparison test for you. Test especially around amounts, dates and sorting. One difference there is an incident.

- Have it list the places this migration might change behavior, and confirm each one is what you want
- Changed defaults are especially dangerous. Old version defaults to A, new version defaults to B, and it's wrong silently
- Keep the old code path until the very end, behind a switch. If something breaks, one flip takes you back

**Example prompt: get the mapping table before migrating**

> I want to migrate this code from <old version/old framework> to <new version/new framework>.
>
> Please don't write code yet. First give me three things:
> 1. A mapping table: old pattern → new pattern, one per line. Mark the ones where the arguments changed.
> 2. The parts with no direct equivalent: list them separately and say how they're usually handled.
> 3. The places where behavior might change: for each one, say what the old behavior was and what the new one becomes.
>
> Once I confirm, then change the code. When you do:
> - Change only this one file. Don't touch the others.
> - Keep the original logic. Don't "optimize" anything along the way.
> - Add a comment at each change marking it as a migration change.

## A template you can copy

Equivalence test prompt, ready to copy

```
I want to verify that behavior is exactly the same before and after the
migration.

Old code:
<paste the old code>

New code:
<paste the new code>

Please write a test script that does these things:

1. Build a batch of inputs covering: normal values, empty values, boundary
values (0, negative numbers, the maximum), malformed values, and duplicate
values.
2. Feed the same batch of inputs to the old implementation and the new one.
3. Compare the outputs one by one. Print any difference at all: what the input
was, what the old output was, what the new output was.
4. Print a one-line summary at the end: how many cases were tested, how many
didn't match.

Requirements:
- Put both implementations in the same script. I want to be able to run it
directly.
- Don't modify either implementation.
- For floating-point comparisons, use a tolerance. Don't compare for exact
equality.
```

## Common mistakes

### × Letting it migrate the whole project at once

The change is too large to debug. When something breaks you can't find which step did it. Go file by file, one at a time.

### × Starting without reading the mapping table

You don't know what got swapped for what, so you can't accept the result either. The mapping table is the first gate. You have to pass it.

### × Calling it a success because the tests pass

Edges the tests don't cover are where things break. Add an equivalence test and compare old and new outputs on the same inputs.

### × Leaving no way back

Delete the old code and your only option is to fix forward. Keep the old path behind a switch. The cost of flipping back is near zero.

## Checklist

☐ I got the mapping table first and understood every line of it

☐ I migrated one file as a trial run, and only spread out after confirming the style and approach were fine

☐ I committed after each file, and ran the tests

☐ I wrote an equivalence test, confirmed old and new behavior matched, and covered the key boundaries

# 61
# Prompts Need Version Control Too

You change a prompt once and it may get a little better or a little worse, but you can't say what you changed or which version started the decline. Manage it like code and those problems go away.

## What this chapter covers

☐ You can save prompts as files, with the version and the purpose written down

☐ You can use a fixed set of questions to test whether a change really made things better

☐ You know how a team should share and update prompts

## 01 A Prompt Is Code

That block of text you tuned through repeated rounds in a chat window is an asset. Close the window and it's gone, and next time you start from scratch.

Save it to a file. Put the purpose in the file name, and at the top of the file write: what it does, who uses it, how many versions it's had, what changed in each.

Copy the old one before you edit. A broken prompt doesn't throw an error. It just quietly gets worse, so you need a way back.

- One file per version, or store it in a version control tool. Don't overwrite the original in place
- Write one line at the top of each version: what problem this version solves. Six months from now you'll thank yourself
- Store the prompt together with sample outputs. A prompt on its own tells you nothing about quality

## 02 How Do You Know a Change Helped

Put together a fixed set of questions, twenty to fifty of them, covering common cases and edge cases. Every time you change the prompt, run the whole set.

Then compare the old and new outputs. Focus on the failures: did anything that used to come out right now come out wrong.

That question set is your regression test. It doesn't guarantee the new version is better, but it does guarantee you didn't break what already worked.

- Put the hard cases you've actually run into in the question bank. Don't fill it with easy ones
- Look at at least ten outputs per change. One or two and you'll miss things
- Record the effect of each change: which type of question got better, which got worse

**Example: what a prompt file looks like**

```
# Prompt: customer service email reply
# Purpose: turn a user complaint email into a sendable reply draft
# Version: v3 (2026-10-20)
# Changes in this version: added "if you're not sure, say you're not sure", to
reduce invented compensation offers
# Previous version: v2's tone was too soft, users thought we were admitting
full responsibility
# Validation question bank: prompts/tests/service-mail-questions.jsonl

You are a customer service agent. Below is a user complaint email. Write a
reply draft.

Requirements:
1. First state what facts we have confirmed, then state what we will do about
it.
2. For parts of the resolution that are uncertain, write "we'll need to verify
and get back to you". Don't make a direct promise.
3. Don't admit responsibility we haven't confirmed, but don't dodge it either.
4. Keep the whole reply under 200 words.
5. Don't open with boilerplate like "we sincerely apologize for the
inconvenience".

User email:
<paste the email>
```

## 03 How to Share Them on a Team

A prompt one person tuned often doesn't work when someone else uses it, because the context is missing. Share the context along with it: what this prompt assumes it can read, and who uses the output.

Set up one directory, sorted by scenario: writing code, writing documents, doing analysis, replying to customers. New hires pick from it directly.

Whoever changes something explains what they changed. Without a note, the next time something breaks nobody can trace it.

- Writing down which scenario a prompt is for matters more than the prompt itself

- Clear out the unused ones regularly. Leaving them there misleads new hires into thinking this is the standard approach
- Keep prompts that touch sensitive company information separate, with restricted access

## A template you can copy

Prompt revision request, ready to copy

```
This is my current prompt:
<paste the old prompt>

Please revise it for me. The goal is: <the specific problem, e.g. "reduce invented content", "shorter output", "follow the format more consistently">

When you're done, give me three things:

1. The full text of the new prompt.
2. A point-by-point account of what you changed and why.
3. A list of side effects this version might cause: could it make some type of answer worse?

Requirements:
- Keep every constraint from the original. Don't delete one rule in order to fix another spot.
- If my goals conflict with each other, point that out directly. Don't force a compromise.
```

## Common mistakes

### × Tuning only inside the chat window

Close it and it's gone. Two hours of tuning, and next time you start over. Saving to a file takes ten seconds.

### × Editing the original file in place

You find it's worse than before and the old version is already gone. Copy it before you edit. The cost is near zero.

### × No fixed question set

You try two or three questions off the cuff after each change and drift further off without noticing. Twenty fixed questions would hold the line.

### × Sharing only the prompt

Nobody knows what it assumes or what the output is for, so it performs badly when they use it. Context matters as much as the prompt.

## Checklist

☐ All my prompts are saved as files, with purpose and version written at the top

☐ Before each change I copied the old one, so I can go back

☐ I have a fixed question set, and after each change I run it and compare for regressions

☐ Prompts I share come with the scenario and the background notes

# 62
# A/B Testing: How to Know a Change Really Helped

You think the new version is better. Users may not agree. This chapter covers the simplest way to measure the real effect, instead of deciding by feel.

## What this chapter covers

☐ You can design a comparison test that actually reaches a conclusion

☐ You know how many samples you need and which metrics to watch

☐ You can tell "it really got better" apart from "we got lucky this time"

## 01 Pick One Metric First

"It feels better" is not a metric. A metric is a number you can calculate: how often users accept the output, average number of edits, completion rate, complaint count.

Watch one primary metric at a time. Look at five numbers at once and you'll never reach a conclusion.

The primary metric has to connect to the problem you set out to solve. Users click more but the problem stays unsolved. That isn't improvement.

- Set the metric before you start. You can't look at the data first and then pick a flattering number
- Besides the primary metric, keep two or three guardrail metrics to see whether you broke something else

- Collect qualitative feedback too. A user saying where it felt awkward points at the next fix better than a number does

## 02 Split People into Two Groups

The simplest approach: split the same pool of users randomly into two groups, one on the old version and one on the new, and run it for a week or two.

Too small a sample and you can't conclude anything. A difference between a few dozen people is probably noise. You need at least a few hundred uses.

If true randomization isn't possible, alternate by time: old version on odd days, new version on even days. It's less rigorous than random assignment, but far better than deciding by gut.

- The two groups need to be alike. All veteran users on one side and all new users on the other, and the difference tells you nothing
- Don't change anything else during the test, or you can't tell what deserves the credit
- Don't look at results before you've run long enough. Looking early gets you swayed by mid-run noise

## 03 How to Read the Results

The new version doing slightly better doesn't mean it's actually better. Ask the follow-up question: could this gap just be luck?

A rule of thumb: with only a few dozen uses in each group and a gap of a few percent, treat it as no difference. A gap of more than double is probably real.

What deserves more attention is the failures: where the new version is clearly worse. Better overall but worse in one category usually points at the most valuable thing to fix.

- Have AI calculate statistical significance, the probability that a gap this size is pure chance, instead of eyeballing it

- Break it out by scenario. Don't look only at the overall number. An overall improvement can hide one scenario falling apart
- State the boundary in your conclusion: better for which users, in which scenario

**Example prompt: have it compute significance**

```
I ran an A/B test. The data is:

Old version: <N1> uses, <A1> met the target
New version: <N2> uses, <A2> met the target

Please do three things:
1. Calculate the rate for each group, and the absolute difference.
2. Calculate the probability that a gap this size is pure chance, using a
significance test, and say which method you used.
3. Give a conclusion: is this gap a real difference or noise? If the sample is
too small, tell me how much more data I need at minimum.

Requirements:
- Don't jump straight to "the new version is better". Give the reasoning first.
- If the data I gave you isn't enough to conclude, say so plainly.
```

## A template you can copy

Test design prompt, ready to copy

```
I want to verify whether this change has any effect: <describe the change>

Background: <user scale, usage frequency, what I'm able to change>

Please design a test that can reach a trustworthy conclusion, covering:

1. Primary metric: what number measures the effect? How is it calculated?
2. Guardrail metrics: which numbers would show I didn't break something else?
3. Grouping: how do I split people? What I can actually do right now is <random assignment / alternate by time / something else>.
4. Sample size and duration: how much data do I need at minimum, and how long do I have to run before concluding? Show the estimate.
5. Likely confounders: what else during the test could affect the results? How do I avoid it?

Requirement: design it around my actual constraints. Don't give me a perfect plan I can't carry out.
```

## Common mistakes

### × Starting the test without a metric

Pick a flattering number once the data is in, and you have no conclusion at all. The metric has to be written down before you start.

### × Concluding from too small a sample

A few percent difference across a few dozen uses is basically noise. Have it estimate how much data you need, and run to that number.

### × Changing several things at once

Even when things improve, you don't know which change did it. One variable per round. That's the floor.

### × Looking only at the total, not by scenario

An overall lift can hide one important scenario falling apart. Look by segment. Failures carry more information than wins.

## Checklist

☐ I set the primary and guardrail metrics before the test started

☐ I worked out how many samples I needed, and didn't conclude from a few dozen uses

☐ During the test I changed only this one thing, nothing else

☐ I looked at the results by scenario, and checked the failures

# 63
# Long Tasks: Getting AI Through Hours of Work Without Drifting

The most common problem with hours-long AI work isn't capability. It's that it forgets the goal along the way and drifts further off. This chapter covers how to cut a long task into a shape you can pull back together.

## What this chapter covers

- ☐ You can cut a long task into stages that each produce something
- ☐ You can set checkpoints and pick up again after an interruption
- ☐ You know how to put a cost ceiling on a long task

## 01 Why Long Tasks Drift

The longer the task, the more content piles up in the middle. The further in it gets, the more likely it loses the goal set at the start and clings to whatever detail is closest at hand.

So the key to a long task isn't making it smarter. It's pinning the goal at the front and keeping progress recorded outside the conversation.

Recorded outside means: write a status file after each step. The file holds only four things: the goal, what's done, the next step, and known issues. If it breaks off, you can pick up again.

- Put the goal first, and paste it again at the start of every new conversation
- At the end of each stage, have it summarize where things stand and what's still missing to reach the goal. You check it

- Clear out irrelevant discussion promptly. Don't let it pile up inside the same thread

## 02 Cut It into Stages That Produce Something

A good cut gives each stage an artifact you can check: a file that runs, a list, a table. A stage you can't check is pointless.

Before each stage, have it state what that stage will finish and what finished looks like. If it can't say, the cut is wrong. Cut it again.

When the artifact appears, check it. Only start the next stage once it passes. That's far less work than checking everything at the end.

- Keep each stage to an hour or two of work. Longer and you won't notice problems building up in the middle
- Have it list, at each stage end, what got done, what didn't, and what's left over
- Record leftover issues separately. Don't let it tack new requirements onto them

## 03 Set Gates and Ceilings

A long task needs a few points where it has to stop and ask you. Before operations that delete data, change configuration or spend money, for instance.

Set a cost ceiling too. A few hours of running can cost far more than you expect. Say up front: stop at this amount.

Also set a stop-loss: after several failed attempts in a row, stop and talk it over. Don't keep retrying in the same hole.

- Tell it explicitly which operations need your sign-off and can't be executed on its own
- Have every report include what it has consumed so far, so you know what you're spending
- Three attempts with no progress and it's time to change approach, or take it over yourself

**Example prompt: setting the rules for a long task**

> This is a long task. Let's set the rules before we start.
>
> Goal: <one sentence on what finished looks like>
> Acceptance criteria: <how we judge it's done>
>
> Work in stages, and follow these rules in every stage:
> 1. Before starting, say what this stage will do and what it will produce, and wait for my confirmation.
> 2. At the end of the stage, give me: what got done, what didn't, and leftover issues.
> 3. Write a status file with only four items: goal, done, next step, known issues.
>
> Hard requirements:
> - For operations that delete data, change configuration or spend money, ask me first. Do not execute them yourself.
> - If the same problem isn't solved after three attempts in a row, stop and tell me. Don't keep trying.
> - Paste the goal above again at the start of every stage.

## A template you can copy

Status file template, ready to copy

```
# Task status

## Goal
<one sentence: what finished looks like>

## Acceptance criteria
<how we judge it's done, as specific as possible>

## Done
- <what got done, with where the artifact is>

## Next step
- <the one thing to do immediately next; write only one>

## Known issues
- <where it's stuck, what was tried, why it didn't work>

## Off limits
- <files, configs and data that must not be touched>

## Consumed
<rounds used, cost so far>

---
When you start a new conversation, paste this whole file in first, then say
"continue".
```

## Common mistakes

### × Stating one enormous goal in a single breath

It loses the goal somewhere in the middle, and you don't know where it got to. Cut it into stages with artifacts and check each one.

### × Not writing a status file

The conversation gets long and it forgets, and one disconnect means explaining everything again. The status file is the save point for a long task. It takes five minutes to write.

### × No stopping conditions

It will retry the same mistake a dozen times, burning time and money. Three attempts with no progress and you call a halt.

### × Running it all day in a single conversation

Quality declines the further in you go. Close out a stage, open a fresh conversation, paste the status file and carry on. That's actually steadier.

## Checklist

- ☐ The task is cut into stages, each with an artifact I can check
- ☐ I wrote a status file, so I can interrupt and resume at any point
- ☐ I set a list of operations that have to stop and ask me first
- ☐ I set a stop-loss condition and a cost ceiling

# 64
# The AI Failure Mode Checklist: When Not to Trust It

It doesn't fail at random. It fails in a fixed set of ways. Once you know them, you catch the nonsense the moment it starts.

## What this chapter covers

- ☐ You can recognize the six most common failure modes
- ☐ You have a way to check each one on the spot
- ☐ You know when not to use it at all

## 01 The Six Most Common Ways It Goes Wrong

First, invented sources. It hands you a book title, a link, a statute number that looks completely real and doesn't exist. Where citations are required, this is the most damaging one.

Second, arithmetic errors. Three numbers is fine. More than that and it's wrong. Whenever you need an exact result, have it write code and run it yourself.

Third, stale knowledge. Its knowledge stops at the day it was trained. New versions, new policies, new prices: you may get last year's answer, or something even older.

- Fourth, agreeing with you. You show which way you lean, and it goes along, making its objections softer each time
- Fifth, pretending it read something. Your file was too long, it didn't finish, but it answers as though it did
- Sixth, overconfidence. When it says "definitely" or "certainly", that's often exactly when it's least sure

## 02 How to Check It on the Spot

Anything cited has to be clickable. If it won't open, it's invented. No exceptions to this one.

For numbers, have it write code or show the arithmetic, then check it yourself. If it won't show the steps, treat it as never having calculated anything.

For anything time-sensitive, ask directly when the information was current as of. If it can't say, treat it as old.

- To test whether it's just agreeing with you, rephrase and ask about the opposite view. See if it switches sides immediately
- To test whether it finished reading, have it give a specific detail from the file, like which section or a particular number
- When it sounds absolute, follow up with "under what conditions would this conclusion not hold"

**Example prompt: have it self-check its own answer**

```
Do a self-check on your previous answer. Answer point by point:

1. Does anything in that answer contain a citation? If so, list each source, and say how confident you are that it actually exists.
2. Are there numbers? How were they derived? Show the arithmetic or the code.
3. How much of this came from my input and how much from your training knowledge? Mark anything you can't separate.
4. Did my phrasing push you toward agreeing with me? If any of my premises are wrong, point them out now.
5. Which sentence are you least sure about? How should that sentence be verified?

Requirement: be honest. If you're not sure, say "not sure". Don't give a definite answer just to sound reliable.
```

## 03 When to Skip It Entirely

Final judgments about safety, law or medicine can't rest on it alone. It can organize information and lay out options, but a person has to

decide.

Things you're accountable for: signing contracts, filing taxes, publishing announcements. It can write the draft, but you read every word before it gets used.

Things it can't see: what's going on inside your company, the relationships on your team, what happened yesterday. It doesn't know, and asking is a waste of time.

- Don't paste other people's private information or company secrets into it
- Any calculation involving money, do it yourself or with code
- For any conclusion it gives you, you have to be able to say how you verified it. If you can't, it isn't verified

## A template you can copy

Review checklist for high-stakes answers, ready to copy

```
I plan to use this answer in a formal setting. Please review it against the
checklist below and give a conclusion on each point:

<paste the content you plan to use>

1. Facts: for every factual statement, mark it "verifiable" or "needs
verification". For the verifiable ones, say where to verify them.
2. Citations: for every source cited, mark how confident you are that it really
exists. Anything less than high confidence means I go check it separately.
3. Numbers: the source and method of calculation for every number. Anything
calculated has to be reproducible.
4. Currency: which items might be out of date? Roughly when are they from?
5. Gaps: to judge this properly, what information is still missing?

Finish with one sentence: can this content be used directly in a formal
setting? If not, what most needs to be added?
```

## Common mistakes

### × Trusting the citations it gives

Inventing sources is its most consistent flaw. Every citation has to open, or resolve to something real. If it doesn't, treat it as never said.

### × Using it to do the money math

It's unreliable on multi-digit arithmetic. Do amounts yourself, or have it write code and run it.

### × Mistaking confidence for accuracy

Tone has nothing to do with accuracy. The more absolute it sounds, the more you should ask under what conditions it fails.

### × Letting it make decisions you're accountable for

It can organize information but it can't carry responsibility for you. For contracts, taxes and public announcements, it can write the draft, but a person sets the conclusion.

## Checklist

☐ I actually traced the source for every citation I plan to use

☐ Every number was either calculated by me or run through code

☐ I confirmed the as-of date for anything time-sensitive

☐ Before finalizing I had it self-check, and then I reviewed it myself

# 65
# Bringing AI into a Team: Getting Non-Coders Started

One person using it well doesn't help if the team doesn't follow. This chapter covers how to roll it out, what to start with for the biggest effect, and which moves turn people off.

## What this chapter covers

☐ You can pick a scenario that saves time right away and use it as the demo

☐ You know how to build a shared library people actually want to use

☐ You avoid the traps that trip up most rollouts

## 01 Don't Start with Training

Opening with a two-hour training session full of theory leaves most people still unable to use it. That's the most common way this fails.

The approach that works goes the other way. Pick one specific person and one specific task. Help them save two hours with AI. Then let them say one sentence about it in the group chat.

One real story about saved time beats ten slides. People trust colleagues, not trainers.

- Pick work that repeats every week and needs no judgment: tidying spreadsheets, writing standard-format documents, fixing typos
- The first time, sit with them and finish it together. Let them see the result themselves instead of watching you demo it
- Let the person who benefited do the talking. If you tell the story, people think you're selling something

## 02 Build a Library People Want to Use

What belongs in the library isn't the prompt so much as the scenario: "I need to write my weekly report" maps to this block of text, copy it and you get a result.

Every entry needs three things spelled out: what scenario it's for, where to paste it, and roughly what the output looks like. Miss one and people won't dare use it.

Fewer and better is the rule. Ten that actually work beat fifty that nobody reads.

- Write scenario descriptions in their words, not your jargon
- Regularly ask which entries they used and which they didn't, and delete the ones nobody used
- Let people edit entries, but require a one-line note on what they changed

## 03 Set Boundaries Without Making It Scary

You have to state clearly what can't be uploaded: customer lists, ID numbers, undisclosed financial data, the text of contracts. These are hard lines.

But don't turn boundaries into a scare story. Saying "these things can't go in, everything else is fine" is far more useful than a vague "be careful with security".

And don't set a "you must use AI" target. Usage forced that way is pure box-ticking, and it makes people resist.

- Keep the boundaries to three lines or fewer and post them where everyone sees them. More than that and nobody remembers
- Offer a safe alternative: if the original can't go in, send a processed version
- Make failure acceptable. Someone making a mistake and saying so is worth more than hiding it

**Example prompt: designing a small pilot rollout**

> Help me design a small pilot rollout. No theory. It has to be doable.
>
> Background: my team has <headcount> people, <headcount> of whom write code. The most time-consuming repetitive work on the team each week is: <name two or three things>
>
> Give me:
> 1. Which thing should we try first? Why that one, and what's the reasoning.
> 2. To find someone willing to try it, what should I say? Give me something I can say word for word.
> 3. The first time I sit down with them, what are the steps? What happens in each.
> 4. If it works, how do I make the result visible to more people without it looking like I'm selling something.
> 5. If it fails, how do I close it out without discouraging everyone else.
>
> Requirement: every suggestion has to be achievable at my team size. Don't hand me a plan that needs a formal project.

## A template you can copy

Shared entry format, ready to copy

```
## Scenario
<in a colleague's own words: what I have to do every week>

## How to use it
Copy the block below and replace the angle-bracketed parts with your own:

<the full prompt>

## Roughly what the output looks like
<paste a real output excerpt with sensitive details removed, so people know
what they'll get>

## Notes
- <when this entry doesn't apply>
- <what must not be pasted in>

## Who uses it
<a name, or write "verified working">

## Recent changes
<date, who, what changed>
```

## Common mistakes

### × Kicking off with an all-hands on theory

It sounds reasonable in the room and nobody does anything afterwards. Help one person save time first. That spreads faster than training.

### × Building a big comprehensive library

Too many entries and nobody browses it, and it ends up as unmaintained documentation. Ten that work beat fifty for show.

### × Covering risk and not usage

All anyone remembers is "we're not allowed", so they just don't use it. Lead with scenarios that work, then draw the three hard lines.

### × Setting a usage-rate target

> Forced usage is all going through the motions, and it breeds resentment. What matters is whether time was actually saved, not how many times it was used.

## Checklist

☐ I demoed on one concrete task first, instead of organizing training

☐ Every entry in the shared library has a scenario, usage instructions and a sample output

☐ I drew clear lines around what can't be uploaded, and offered alternatives

☐ I set no box-ticking usage targets. I only look at whether real time was saved

# 66
# A Year from Now: How All of This Will Change

The specific tools in this book will go out of date. The things underneath them won't. This chapter lays out what changes, what doesn't, and how to keep up.

## What this chapter covers

☐ You know which capabilities will most likely get stronger and cheaper

☐ You know which practices don't change with the tools

☐ You have your own way of keeping up, without chasing every piece of news

## 01 What Will Change

It will be able to take in more: from reading only text, to seeing images, hearing audio, watching video and operating a computer directly.

Prices will keep falling and speeds will keep rising. Scenarios too expensive to even consider today may be something you use freely a year from now.

Models will get better at acting on their own: calling tools themselves, checking their own results, retrying themselves. The "direct it step by step" approach will matter less.

- Specific version numbers, prices and parameters will all be out of date. Don't memorize them
- Don't tie your workflow to one product. Interfaces changing is the normal state of things

- Every six months, retry the scenario you gave up on. It might work now

## 02 What Won't Change

People who can state what they need clearly always have the edge. The stronger the tools, the bigger that edge, because a tool is an amplifier, and what it amplifies is your own clarity.

Verify in small steps, keep a way back, check afterwards. None of those three depend on how smart the model is. When something goes wrong, the cost is always yours to carry.

Knowing what not to hand to AI is a kind of judgment that only gets more valuable.

- State the goal, give enough context, define acceptance criteria. Those three don't expire
- Check the key conclusions yourself. That rule doesn't get cancelled because the model got stronger
- Keep your own judgment in reserve. Don't hand decision-making over wholesale

## 03 How to Keep Up

Don't chase every piece of news. Something new lands in this field every day. You can't keep up, and most of it has nothing to do with the work in front of you.

A rhythm that works is once a quarter: take the thing eating the most time right now and check whether there's a new approach that removes it. If there's a problem, look into it. If not, keep using what you have.

When there's an obvious jump in capability, when it suddenly does reliably some kind of work it couldn't do before, that's when to spend time learning it systematically.

- Reread the basics once a year. The basics haven't changed, but your understanding of them has

- Follow progress specific to your industry. Don't just watch the general hype
- Keep two or three vendors as fallbacks. Don't bet everything on one

**Example prompt: the quarterly review**

```
Help me do a review of whether this workflow still holds up.

Here's how I work right now:
<describe the two or three things you currently use AI for, and how you do them>

Please tell me:
1. Which of these steps might now have an easier way? Give your reasoning. Don't just say "you could try new tools".
2. Which step am I doing myself that could be handed off?
3. Which step looks like it could be cut but actually shouldn't be? Why?
4. If I only have an hour to check, which one should I try first?

Requirement: don't recommend specific products. I care about the approach, not the tool.
```

## A template you can copy

Four quarterly questions, ready to copy into your notes

```
Spend twenty minutes each quarter answering these four questions:

1. What are the three things I repeated most this quarter?
2. Of those three, which one might now take half the time?
3. What did I give up on last quarter, and can I do it now?
4. Is there anything I've been doing a certain way without ever asking why?

When you're done, do one thing only: pick the item above that saves the most time and change it this quarter. Don't change all four at once.
```

## Common mistakes

- × **Chasing every new model**

There's news every day, you can't keep up, and most of it has nothing to do with your work. Once a quarter is enough.

### × Memorizing specific parameters

Version numbers, context window sizes, prices. They change every few months. Memorizing them just makes you look out of date.

### × Locking your workflow to a single product

The interface changes, the price goes up, the service shuts down, and you have to rebuild. Keep two or three fallbacks and switching costs far less.

### × Assuming the basics need no review

The tools change. Stating what you need, verifying in small steps, keeping a way back: those don't. But every year you understand them one layer deeper.

## Checklist

☐ I know which parts of this book will go stale, and I won't memorize them

☐ I've held on to the three things that don't change with the tools: state the goal, verify in small steps, keep a way back

☐ I have a fixed review rhythm instead of keeping up by scrolling news

☐ I have options from more than one vendor

# Conclusion: The Rest Is You, Doing the Work

Sixty-six chapters later. If you read all of it in one go, you probably feel both that you learned something and that it is a bit of a blur. That is normal.

Knowledge is a blur until it becomes a move.

## If you remember one sentence

Remember this one: AI is not answering your question — it is continuing your sentence.

Give it a vague opening and you get a vague continuation. Give it a clear, bounded opening with an agreed definition of done, and what comes back is usable as-is.

Every technique in this book — structured prompts, chain-of-thought, role setting, few-shot examples, output format constraints — is ultimately doing the same thing: making the opening clear.

## One small thing to do tomorrow

Do not try to apply all sixty-six chapters at once. Pick one thing you repeat every week: a status report, meeting notes, a piece of boilerplate code.

Then use the templates in this book to write the prompt for that one task, and save it to a text file. Next time, copy and paste.

Do it once and you will know whether this is worth your time.

## About the book itself

This book is written for ordinary people, which is why it deliberately leaves things out: model internals, training details, benchmark comparisons. Not because they do not matter, but because they do not matter for what you are trying to do right now.

If you later find that you need them, by then you will know what to look up. That is far more useful than being handed a pile of nouns you cannot use on day one.

## Finally

Tools will keep changing. The model that works well today may be gone next year; today's best practice may be next year's cautionary tale.

One thing will not change: people who can state their own requirements clearly have the advantage with any tool.

That is what this book actually teaches. Not AI — that.

Good luck.

# Afterword

The main text ends at chapter 66. What follows is not part of the tutorial. It is a few things I wanted to say to you separately.

## If you have a question, write to us

This book was written one chapter at a time, so there are certainly places where we were unclear or got something wrong. If you find a passage that reads badly, that is wrong, or that should have been a chapter and is not, we would like to hear from you.

Either address works. Pick one:

cyan714@outlook.com

xyymc_666@qq.com

We also run a QQ group where people work through concrete problems together:

QQ group: 956766495

Working alone, you get stuck easily. Someone in the group has probably hit the same wall, and one question can save you half a day.

## About this book

AI Programming Guide is written and maintained by Canglan Workshop.

You may do the following without asking us:

- Learn from it yourself, share it with colleagues, print it out and keep it on your desk
- Copy the prompt templates and code straight into your own projects
- Quote ideas or passages with a credit to AI Programming Guide by Canglan Workshop

Please do not:

- Sell it, or bundle it into a paid course or a paid community
- Remove the author's name and publish it as your own
- Reproduce long passages without attribution

The code and prompts in this book are meant to be used, not admired. The more you adapt them to your own work, the more useful they become.

One caveat: every method here is based on practice at the time of writing. Tools and products change, and some specifics may be out of date in a year or two. We deliberately avoided pinning version numbers and prices to particular products so that the book would age better.

You are responsible for the consequences of what you do with this book. For decisions involving money, law, safety or medicine, rely on a qualified professional. This book can help you think a problem through; it cannot make the decision for you.

Current edition: first edition, October 2026, sixty-six chapters, bilingual.

## A few last words

Writing the end of this book, I kept asking myself one question: when you close it, what will you actually take away?

Probably not the content of all sixty-six chapters. You will forget most of the chapter titles, and forget which page a particular template was on. That is normal. I cannot remember everything I wrote either.

But there are a few sentences I hope you keep.

The first: AI is not answering your question, it is continuing your sentence. Every technique in this book grows out of that one line.

The second: being able to state clearly what you want is a rare skill, and it has nothing to do with tools. Plenty of people work for years without ever practising it.

The third: take small steps, and always leave yourself a way back. That is not just how you work with AI. It is how you do anything.

One more thing: do not let this book make you anxious.

You will see a lot of talk about who AI will replace, and people eager to mention how fluently they use it. Listen to that for long enough and you start to panic, but the real gap is much smaller than the story. Most people open a chat box, type "write me something", get something unusable, and close it.

You now know one thing they do not: how to say what you mean. That is enough to start.

The rest comes with doing it once.

If this book saved you an afternoon, it has earned its keep.

Good luck.

Canglan Workshop

October 2026

# 结语：剩下的靠你动手

66 章读完了。如果你是从头读到尾的，我猜你现在既觉得有点东西，又觉得有点乱。这很正常。

知识在变成动作之前，都是乱的。

## 如果只记一句话

记住这句：AI 不是在回答你的问题，是在接着你的话往下写。

你给它一个模糊的开头，它就只能给你一个模糊的续篇。你给它一个清楚的、有边界的、说好什么算完的开头，它给你的东西就能直接用。

这本书里所有的技巧——结构化提示词、思维链、角色设定、少样本示例、输出格式约束——说到底都是在做同一件事：把开头写清楚。

## 明天可以做的一件小事

别急着把这 66 章都用上。挑一件你每周都要重复做的事，比如写周报、整理会议纪要、写重复的样板代码。

然后用书里的模板，把这件事的提示词写出来，存到一个文本文件里。下次再遇到，直接复制粘贴。

做一次，你就知道这东西到底值不值得你花时间了。

## 关于这本书本身

这本书是写给普通人的，所以它有意省略了很多东西：模型内部结构、训练细节、各家跑分对比。这些不是不重要，是对你现在要做的事不重要。

如果你后来发现自己需要这些，那时候你已经知道自己该去查什么了。这比一开始塞给你一堆看不懂的名词有用得多。

## 最后

工具会一直变。今天好用的模型，明年可能就换了；今天的最佳实践，后年可能就是反面教材。

但有一件事不会变：能把自己的需求说清楚的人，在任何工具面前都占优势。

这本书教的其实不是 AI，是这件事。

祝你顺利。

# 后记

正文到第 66 讲就结束了。后面这些不属于教程，是一些想单独跟你说的话。

## 有问题，随时找我们

这本书是一篇一篇写出来的，难免有说不清楚或者写错的地方。你读到哪里觉得别扭、觉得不对，或者觉得该补一章没补，都欢迎写信告诉我们。

两个邮箱都可以，随便挑一个：

cyan714@outlook.com

xyymc_666@qq.com

也建了一个 QQ 交流群，平时大家在里面对付各种具体问题：

QQ 群：956766495

一个人琢磨容易卡住。群里多半有人踩过同一个坑，问一句能省半天。

## 关于这本书

《AI 编程指南》由沧蓝工坊编写并持续维护。

下面这些事你可以做，不用问我们：

- 自己学、给同事看、打印出来放在工位上
- 把书里的提示词模板和代码直接复制到你的项目里用
- 引用书里的观点或片段，注明来自《AI 编程指南 • 沧蓝工坊》

下面这些事请别做：

- 拿去卖，或者打包进付费课程、付费社群

- 改掉作者署名当成自己的东西发布
- 大段照搬而不注明出处

书里的代码和提示词是给你用的，不是给你供着的。改得越贴合你自己的活，它越有用。

需要说明的是，书里的方法都基于编写时的实践。工具和产品会变，有些具体做法过一两年可能就过时了。涉及具体产品的地方我们尽量不写版本号和价格，就是为了让它耐用一点。

按书里的内容操作，产生的后果由操作者自己承担。涉及钱、法律、安全、医疗的判断，请以专业人士的意见为准。这本书只能帮你想清楚问题，不能替你做决定。

当前版本：初版，2026 年 10 月，共 66 讲，中英双语。

## 几句想说的话

写到最后，我一直在想一个问题：你合上这本书的时候，真正会带走什么。

大概不会是这 66 讲的具体内容。你可能会忘掉大部分章节标题，忘掉某段提示词在哪一页。这很正常，我自己写的东西也记不全。

但有几句话，我希望你能记住。

第一句：AI 不是在回答你的问题，是在接着你的话往下写。这本书里所有技巧，都是从这一句长出来的。

第二句：能说清楚自己要什么，是一种很稀缺的能力，而且跟工具无关。很多人工作了很多年，也没练过这件事。

第三句：小步走，留退路。这不只是用 AI 的方法，是做任何事的方法。

还有一件事想说：别因为这本书产生焦虑。

你大概会看到很多人在谈 AI 会取代谁，也会看到有人把自己用得多熟练挂在嘴边。这些话听多了容易慌，但真实的差距没有传说中那么大。大部分人只是打开对话框问一句「帮我写个东西」，拿到一堆没法用的结果，然后关掉。

你现在比他们多知道一件事：怎么把话说清楚。这就够开始了。

剩下的，用一次就会。

如果这本书帮你省下了一个下午，那它就没白写。

祝你顺利。

沧蓝工坊

2026 年 10 月