Week02｜课时 4｜把 Data Contract 做成工程门禁：Schema、语义、质量、兼容性

Table of contents

Data Contract 不是字段清单，而是输入系统的 gate 2
这节课解决什么问题 2
参考学习时间（60–75 分钟） 3
本课你将完成什么 3
本课产出 3
先看一张总图 4
0. 先把 4 个对象彻底分开 4
1. 为什么 JSON Schema 不等于 Data Contract 5
为什么付费课程里必须把 contract 讲厚 5
2. 一份真正可执行的 contract，至少要有 5 层 6
2.1 Shape / Schema：回答“它是不是这个对象” 8
2.2 Semantics：回答“它是不是我们想要的那个意义” 8
2.3 Metadata / Evidence：回答“以后能不能证明自己没瞎编” 8
2.4 Policy / Access / PII：回答“谁可以碰，碰到哪一步” 9
2.5 Quality / Freshness / SLA：回答“它现在够不够格进入系统” 9
3. 真实解剖一：ticket_contract.json 10
3.1 你先看什么 10
3.2 这一类 contract 至少要守住什么 10
3.3 这类 contract 最常见的两个失败 10
4. 真实解剖二：doc_asset_contract.json 10
4.1 文档 contract 最容易被删掉的字段 10
4.2 一份文档 contract 到底在守什么 11
4.3 如果这些字段丢了，citation / audit / entitlement 会怎么坏 11
5. 兼容性不是一句“升级到 v2”就够了 12
兼容性不是技术问题，而是下游承诺问题 12
5.1 三个最值得练的变更案例 12
5.2 更完整的兼容性矩阵 13
6. 为什么底层支持 schema evolution，上层反而更需要 gate 13
7. 这套 gate 真正怎么进入工程链 14

8. 你跑 pytest tests/contract/ -v 时到底在验证什么 14
你至少应该看到什么 15
如何读 contract test 失败 15
8.5 行业新信号｜为什么 contract 现在更像工程门禁 15
9. 直接动手：怎么把这节课落到 repo 里 15
第 1 步：先看 contract，不先造新 contract 15
第 2 步：把练习样例统一放到 fixture 目录 16
第 3 步：用 Docker devbox 跑 contract tests 16
第 4 步：兼容性怎么记录，不必再起新脚本 16
10. 本课最容易误解的 5 件事 17
11. 小结 17
12. 课后最小行动 17
13. 下一讲衔接 17

Data Contract 不是字段清单，而是输入系统的 gate

这一讲要把 Week02 前三课的判断真正压成工程门禁：

contract 只有在能放行、能拦截、能报警、能进入 CI/CD 时，才算系统能力。

前面学的是输入风险、资产地图和多模态 metadata；这一讲开始回答：这些约束怎么被写成真正可执行的 gate。

回看课时 3 进入课时 5 返回 Week02 总览

下载讲义

提供适合离线阅读的 PDF 版和适合批注整理的 Word 版。

PDF 版 · 打印 / 离线阅读 Word 版 · 批注 / 二次整理

这节课解决什么问题

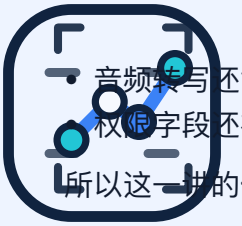
到这里，你已经完成了两件非常关键的事：

- 你知道哪些资产值得进入 AI 系统
- 你也知道这些资产至少要带哪些 metadata 和 PII 边界

但如果你还停在“表格清单 + 规则说明”的层面，系统依然会在真正运行时出事。

因为上游最常见的问题，不是“完全没有数据”，而是：

- 字段还在，但**语义悄悄变了**
- 枚举还能跑，但**口径已经漂了**
- 文档正文还在，但 page_no / section_path 丢了



- 音频转写还能读，但 speaker_role / start_ts / end_ts 没了
- 权限字段还在表里，但**没有变成真正的 gate**

所以这一讲的任务，是把课时 2 和课时 3 的输出，正式抬升成：

机器可读、可校验、可拦截、可讨论兼容性的 Data Contract。

参考学习时间（60–75 分钟）

如果你只阅读正文，大约需要 35–45 分钟；如果你跟着本课一起解剖两份 contract、判断兼容性，再跑一遍 pytest tests/contract/ -v 去读 gate 的反馈，建议预留 60–75 分钟。

本课你将完成什么

学完这一讲，你应该能做到：

1. 解释为什么 Data Contract 不能只写字段名和类型。
2. 用 **5 层结构** 看懂并复核一份真正能进入工程系统的 contract。
3. 区分 additive / conditional / breaking 三类变更。
4. 在仓库里直接使用和完善四类现有 JSON contract。
5. 跑通一次 contract tests，并把样例输入和兼容性判断收口到 repo。

本课产出

完成这一讲后，你至少应该产出这些工件：

- contracts/data/ticket_contract.json
- contracts/data/doc_asset_contract.json
- contracts/data/audio_asset_contract.json
- contracts/data/video_asset_contract.json
- tests/contract/fixtures/week02/
- 一份对 additive / conditional / breaking 的兼容性判断记录

> 这一讲的目标不是创建另一套平行 contract 系统，而是把课程里的 contract 思维直接压进项目当前真实存在的 JSON contract。

先看一张总图

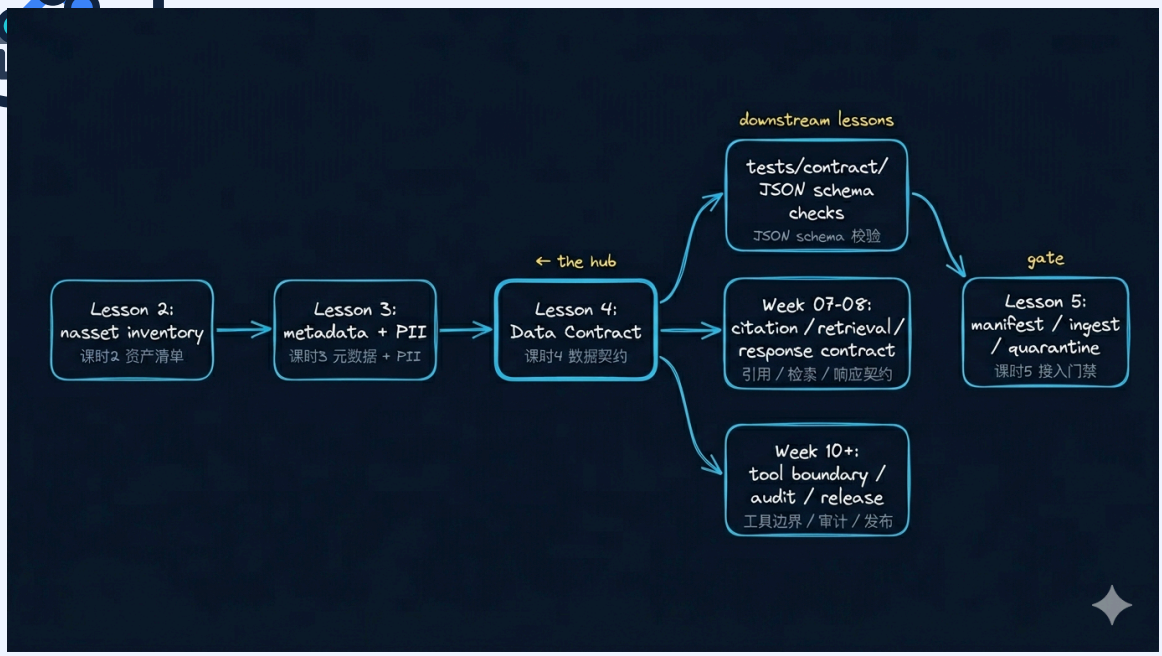


Figure 1: Week02 课时 4 总图

Week02 到这里，已经从“输入认知”正式进入“输入工程”：

- 课时 2 回答：**哪些资产值得接入**
- 课时 3 回答：**这些资产最低要带什么 metadata 和 policy**
- 课时 4 回答：**这些规则怎样变成机器可读的 gate**
- 课时 5 回答：**这些 gate 怎样真正驱动 manifest 和 ingest**

0. 先把 4 个对象彻底分开

在开始谈 contract 之前，先把最容易混淆的 4 个对象拆开。

对象	它解决什么
JSON Schema	字段形状、类型、枚举、格式是否合法
Data Contract	shape + semantics + evidence + policy + quality 是否一起构成准入标准
Manifest	这一次 ingest 到底准备接哪一批数据、按什么模式接
Policy	哪些字段、动作、角色受限制，哪些必须脱敏、拦截或升级审查

i 先记住这句

Schema 回答“它长得像不像”，contract 回答“它够不够格进入系统”，manifest 回答“这次到底怎么接”，policy 回答“谁能碰、能碰到哪一步”。

1. 为什么 JSON Schema 不等于 Data Contract

很多团队第一次写数据契约时，会直接把 JSON Schema 当成全部答案。

这样做的问题是： 你确实可以约束“字段在不在、类型对不对”，但还无法回答：

- 这个字段**业务上到底代表什么**
- 哪些值代表正常状态，哪些值代表风险信号
- 它缺失时是 warning 还是 reject
- 谁能看、谁能搜、谁能把它喂给模型
- 上游变更之后，哪种变化允许自动放行，哪种必须拦截

对比项	JSON Schema 更擅长回答什么	Data Contract 还必须再回答什么
结构	字段、类型、枚举、required	这些结构在业务上意味着什么
合法性	记录是否“长得像”目标对象	记录是否“足够安全且足够可用”
兼容性	结构变化有没有破坏校验	变化是否会带偏检索、生成、工具动作
运行时门禁	能不能做基础 schema 验证	什么时候放行、隔离、拒收
审计与责任	很有限	owner、policy、PII、release、run evidence 必须闭环

! 关键判断

凡是会被系统长期消费的对象，都应该有 machine-readable contract。 但 contract 如果只剩 schema，它还不是工程门禁，只是一份更正式的字段说明。

为什么付费课程里必须把 contract 讲厚

- 因为真实项目里“字段没错但意义变了”的问题，远多于“字段直接不存在”。

- 因为 quality、freshness、owner、SLA 决定的是运行时是否放行，而不是文档是否写得完整。
- 因为 Week02 里的 contract 会直接服务 Week03 ingest、Week08 RAG、Week10 tools、Week14 release。

2. 一份真正可执行的 contract，至少要有 5 层

为了避免把 contract 学成“更长的字段表”，这一讲建议你把它记成 5 层。

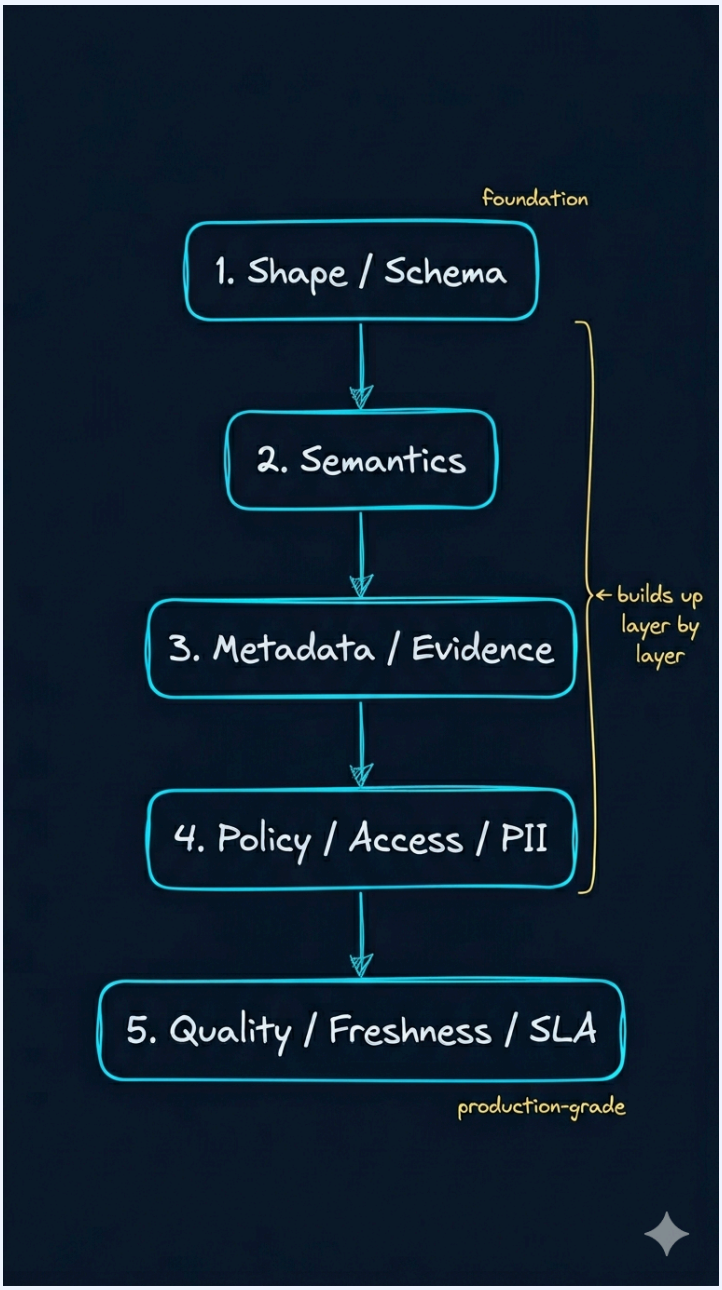
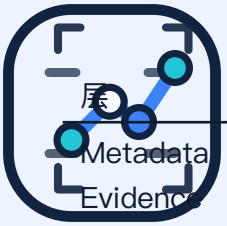


Figure 2: Week02 contract 五层结构图

层	它回答什么	ticket 示例	document 示例	gate 会问什么
Shape	字段与类型对不对	status 是否存在且为 string	page_no 是否为 int	结构是否完整
Semantics	这个字段业务上是什么意思	updated_at 代表什么时间	doc_version 是否代表正式版本	语义是否漂移



层	它回答什么	ticket 示例	document 示例	gate 会问什么
Metadata / Evidence	能不能回指来源	ticket_id / tenant_id	page_no / section_path / bbox	能不能引用与追责
Policy	谁能碰，怎么碰	requester_email 要不要 mask	license_tag 是否允许分发	是否越权 / 违规
Quality / SLA	够不够新、够不够干净	枚举违规率、新鲜度	文档版本延迟、缺页率	是放行还是拦截

2.1 Shape / Schema：回答“它是不是这个对象”

这一层最基础，但不能省。

它通常包括：

- type
- required
- enum
- format
- 嵌套对象的结构定义

如果这一层都没有，系统根本不知道输入对象是不是 ticket / document / audio / video。

2.2 Semantics：回答“它是不是我们想要的那个意义”

这里最容易被忽略，但实际最容易翻车。

比如：

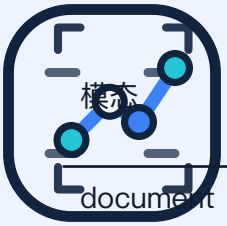
- status 还是 string，但原来表示工单生命周期，现在上游偷偷改成了排班占用状态
- updated_at 还是时间字段，但从“业务更新时间”换成了“ETL 入库时间”
- doc_version 还是存在，但开始混入草稿版本

这些都可能让 schema 继续通过，但系统已经开始稳定地产生错误判断。

2.3 Metadata / Evidence：回答“以后能不能证明自己没瞎编”

如果 contract 不强制保留证据链字段，系统就会陷入“答得像对的，但无法追责”的状态。

模态	如果 metadata 不进 contract，最常见的后果
ticket	租户、产品线、升级链缺失，后续动作边界会漂



模态	如果 metadata 不进 contract，最常见的后果
document	page_no / section_path / bbox 缺失，引用和溯源会崩
audio	speaker_role / start_ts / end_ts 缺失，对话归因会乱
video	frame_ts / scene_ref / transcript_ref 缺失，多模态定位会失真

2.4 Policy / Access / PII：回答“谁可以碰，碰到哪一步”

这一层不是法务附件，而是系统行为边界。

真正要进入 contract 的，至少包括：

- pii_level
- access_scope
- license_tag
- allowed_actions
- redaction_required

没有这层，你就无法在运行时真正阻止“能查到但不该被模型消费”的输入进入下游。

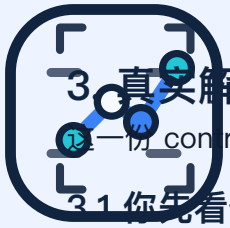
2.5 Quality / Freshness / SLA：回答“它现在够不够格进入系统”

这一层决定系统不是“能吃就吃”，而是“满足最低标准才进来”。

常见判断：

- freshness 是否超窗
- required locator 是否缺失
- enum 是否漂移
- 文档是否缺页
- transcript 是否只有文本、没有时间戳

质量信号	对应动作
非关键字段缺失	warn
局部记录有问题，但整批仍有价值	quarantine
关键字段缺失、contract 不匹配、语义严重漂移	reject



3. 真实解剖一：ticket_contract.json

这一份 contract 代表的是最容易“看起来没坏，但行为已经偏掉”的输入对象。

3.1 你先看什么

建议先按这个顺序阅读：

1. shared identifiers
2. lifecycle fields
3. tenant / access / policy fields
4. evidence / provenance 相关字段

3.2 这一类 contract 至少要守住什么

模块	典型字段	它为什么关键
身份定位	ticket_id, tenant_id, product_line	不然多租户边界会模糊
生命周期	status, created_at, updated_at	不然 KPI、优先级、升级逻辑都会偏
责任边界	owner_team, requester_email, access_scope	不然系统会把不该看的信息暴露出去
证据链	source_ref, conversation_ref, attachment_refs	不然后续引用和审计无从落地

3.3 这类 contract 最常见的两个失败

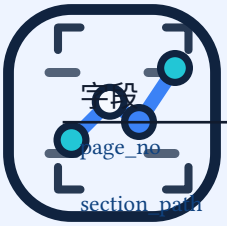
失败模式	表面看起来怎样	实际后果
枚举扩展未收敛	schema 还是通过，status 还是字符串	路由、统计、工具动作一起偏
时间字段语义漂移	updated_at 还存在	增量窗口和 SLA 监控一起失真

4. 真实解剖二：doc_asset_contract.json

文档 contract 的难点不在“有没有正文”，而在“能不能稳定定位与回指”。

4.1 文档 contract 最容易被删掉的字段

字段	很多人为什么想删	真删了会怎样
doc_version	觉得版本号“以后再补”	无法区分当前有效版本



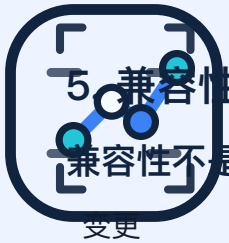
字段	很多人为什么想删	真删了会怎样
page_no	觉得 chunk 里已有内容	引用页码直接失效
section_path	觉得正文搜索能找到	上下文层级没了
bbox	觉得只对视觉任务有用	图文定位、截图证据链崩掉
license_tag	觉得是治理信息	分发和二次使用边界不清

4.2 一份文档 contract 到底在守什么

问题	contract 要回答什么
这是不是正式版本	doc_version / source_type / publish_state
以后能不能按页引用	page_no
能不能按章节还原上下文	section_path
图文块能不能被精确圈定	bbox
内容能不能用于模型和对外输出	license_tag / access_scope / pii_level

4.3 如果这些字段丢了，citation / audit / entitlement 会怎么坏

丢失字段	citation 会怎么坏	audit 会怎么坏	entitlement 会怎么坏
page_no	回答里无法回指页码	无法证明引用来自哪一页	用户看到内容，但无法证明授权页范围
section_path	章节级上下文无法恢复	复盘时找不到原始章节结构	章节级授权和裁剪会失真
bbox	图表或局部截图无法稳定圈定	证据截图很难复核	局部区域是否可展示难判定
license_tag	生成时无法区分可引用与不可分发内容	审计时无法说明为什么允许输出	分发边界直接失守
access_scope	结果可能混入不该暴露的段落	无法还原授权链条	检索和生成都可能越权



5. 兼容性不是一句“升级到 v2”就够了

兼容性不是技术问题，而是下游承诺问题

变更	技术上能不能支持	课程里怎么判
新增非关键字段	通常能	additive
新增枚举但下游未准备	技术上能，但语义危险	conditional
改字段语义 / 改 locator 含义	技术上可能还能过	breaking
改时间字段定义	技术上常能兼容	breaking

如果你不区分变更类型，系统只能在两种极端之间摇摆：

- 要么什么都放行，结果静默带偏
- 要么什么都拦截，结果系统永远推进不了

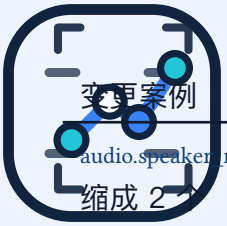
更实用的做法是至少分三类：

变更级别	典型例子	风险	建议动作
additive	新增可选字段 priority_label	低到中	通常可放行，但要记录
conditional	扩充 status 枚举、收紧 freshness 阈值	中	需要 review 后决定
breaking	删除字段、重命名字段、把可选改成必填、语义漂移	高	默认拦截或给迁移窗口

> ⚠ 记住这一条
>
> **语义变更，即使 schema 不变，也可以是 breaking。**

5.1 三个最值得练的变更案例

变更案例	为什么不是一眼就能判断	推荐级别
ticket.status 新增 in_progress	结构没坏，但会穿透统计和 tool route	conditional
document.doc_version 从 optional 改 required	老数据可能直接不再合法	breaking



变更案例	为什么不是一眼就能判断	推荐级别
audio.speaker_role 枚举从 4 个缩成 2 个	老转写样本会失配，归因语义会塌	breaking

5.2 更完整的兼容性矩阵

变化类型	schema 会不会感知	业务是否会受影响	默认是否自动放行	是否需要人工 review
新增可选字段	会	可能	是	建议
新增枚举值	会	高概率	否	是
删除字段	会	一定	否	是
可选改必填	会	一定	否	是
字段重命名	会	一定	否	是
语义漂移但结构不变	不一定	很大	否	必须

6. 为什么底层支持 schema evolution，上层反而更需要 gate

很多人看到 Iceberg 或湖仓体系支持 schema evolution，就误以为：

> 底层都能演进了，上层为什么还要这么严格？

这是两个层次的问题。

层次	关注点
底层 schema evolution	存储层能否兼容字段演进
上层 contract gate	业务语义、权限边界、引用能力、工具动作是否还能成立

也就是说：

- 底层能存下来，不代表上层还能安全使用
- 底层能追加字段，不代表 retrieval / tool / audit 就不用重新评估

7. 这套 gate 真正怎么进入工程链

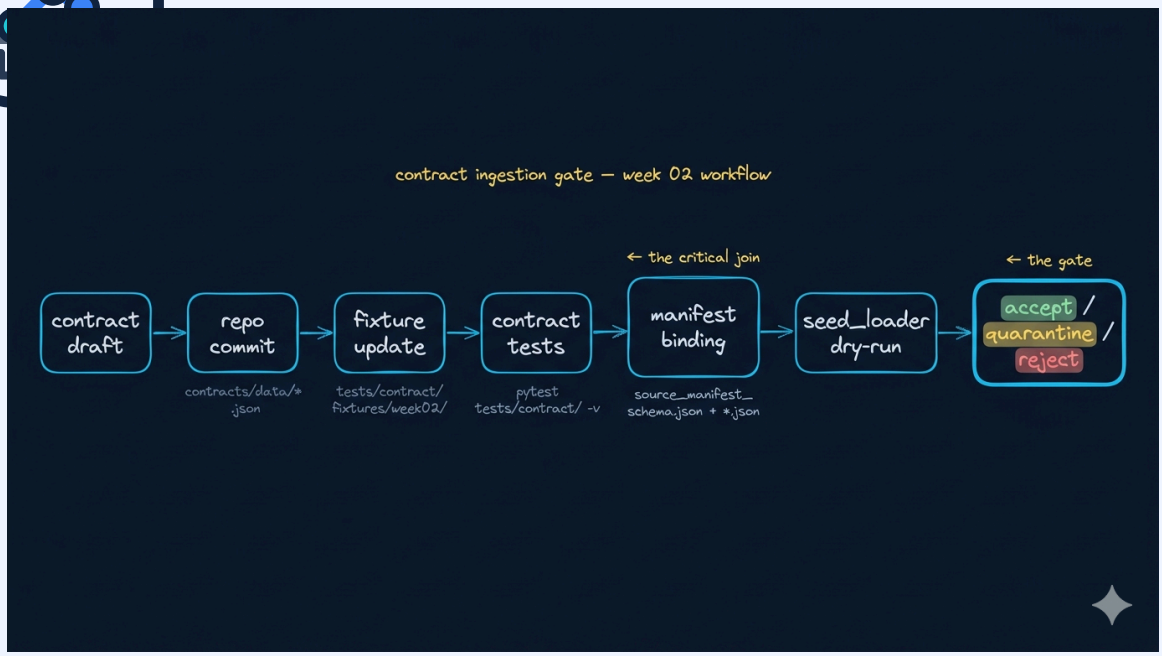


Figure 3: Week02 gate 工程链路图

这张图要表达的是：

- contract 不是孤立文件，而是一个从草稿进入 repo 的长期接口承诺
- fixture 不是演示样例，而是把 contract 变成可验证对象
- pytest 不是最终验收，而是第一道持续门禁
- manifest 和 seed loader 会继续消费这些标准，并做运行时判断

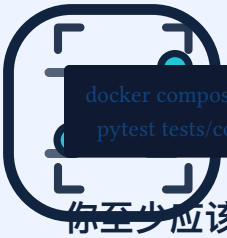
8. 你跑 `pytest tests/contract/ -v` 时到底在验证什么

很多同学只会看“绿了还是红了”，但这一讲必须往前再走一步。

你真正应该理解的是：

你在看什么	它说明什么
JSON schema 是否能被正常加载	contract 本身是否结构合法
fixtures 是否仍满足当前 schema	课程里定义的最小样例是否还站得住
manifest schema 是否稳定	后续 loader 是否还有统一入口
哪条测试失败了	是 contract 变了、fixture 过期了，还是语义已经漂了

统一命令：



```
docker compose --profile tools --env-file infra/env/.env.local -f infra/docker-compose.yml run --rm devbox \
  pytest tests/contract/ -v
```

你至少应该看到什么

- tests/contract/test_json_schemas.py 被执行
- contract schema 被成功加载
- 如果你故意改坏 enum / required / type，测试会失败

如何读 contract test 失败

失败现象	你先怀疑什么
required 缺失	fixture 漏字段，或 contract 的 required 过严
enum 不匹配	上游枚举漂移，或 fixture 已经过时
format 错误	contract 太松 / 太严，或者样例本身就是脏数据
兼容性 diff 报 breaking	下游承诺被打破，不能只看“技术上还能不能跑”

8.5 行业新信号｜为什么 contract 现在更像工程门禁

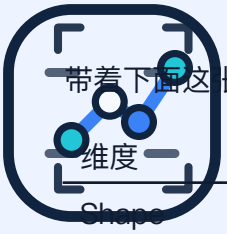
1. 成熟的 data contract 思路已经不只管 schema，还会把 quality、roles、SLA 一起纳入结构。
2. 越来越多团队会用 CLI / CI 去做 lint、test 和 compatibility check，而不是把 contract 只放在 Wiki。
3. 底层 schema evolution 越强，上层 compatibility policy 越不能省，否则“能演进”会变成“静默漂移”。

9. 直接动手：怎么把这节课落到 repo 里

第 1 步：先看 contract，不先造新 contract

建议先按这个顺序阅读：

```
contracts/data/ticket_contract.json
contracts/data/doc_asset_contract.json
contracts/data/audio_asset_contract.json
contracts/data/video_asset_contract.json
```



带着下面这张检查表去看：

维度	你要检查什么
Shape	required、type、enum 是否完整
Semantics	关键字段的业务含义是否明确
Metadata / Evidence	page_no / bbox / section_path / speaker_role / frame_ts 等是否被保留
Policy	access_scope / pii_level / license_tag 是否进入可执行约束
Quality	哪些字段缺失、枚举漂移、新鲜度不达标时应被拦

第 2 步：把练习样例统一放到 fixture 目录

本课不再要求你另外创建一套课程专用的 gate / diff 脚本。 更合理的主路径是：

```
mkdir -p tests/contract/fixtures/week02
```

你至少要准备：

- 一组正例样本
- 一组反例样本
- 一份变更判断记录

第 3 步：用 Docker devbox 跑 contract tests

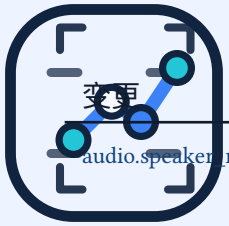
统一用 Docker devbox：

```
docker compose --profile tools --env-file infra/env/.env.local -f infra/docker-compose.yml run --rm devbox \
  pytest tests/contract/ -v
```

第 4 步：兼容性怎么记录，不必再起新脚本

本课的兼容性练习，重点是判断逻辑，不是再造一套 diff 工具。你可以直接用一张表记录：

变更	级别	原因
ticket.status 新增 in_progress	conditional	结构可兼容，但下游语义和路由需要 review
document.doc_version 改成 required	breaking	老数据会直接失配



变更	级别	原因
audio.speaker_role 缩枚举	breaking	旧数据和角色归因语义会一起坏

10. 本课最容易误解的 5 件事

误解	正确理解
contract 就是字段表	contract 是输入门禁
schema 能过就说明系统没问题	语义漂移一样会把系统带偏
兼容性只看字段是否存在	还要看枚举、语义、policy、evidence
pytest 只是形式化检查	它是 Week02 进入持续工程链的第一步
Data Contract 讲完就结束了	下节 manifest 和 seed loader 会继续消费它

11. 小结

这一讲最重要的，不是多记几个 contract 术语，而是把下面这条链看清：

asset inventory -> metadata / PII -> Data Contract -> contract tests -> manifest -> seed loader dry-run

到这里，Week02 的输入控制面才第一次具备了工程可执行性。

12. 课后最小行动

做完这一讲，请至少完成下面三件事：

- ☐ 用 5 层结构重新检查一次四类 JSON contract
- ☐ 在 tests/contract/fixtures/week02/ 下准备最小样例
- ☐ 跑一次 pytest tests/contract/ -v，并记录一条 conditional 与一条 breaking 变更判断

13. 下一讲衔接

下一讲不是再讲一次 contract，而是回答：

这些 contract 怎样真正开始驱动一次 ingest 批次的准入、隔离和运行证据。

这也是 Week02 为什么必须在 Week03 之前把 contract 收口好的原因。