# Week04｜课时 1｜为什么 AI 数据工程需要“有记忆的表”，而不只是能查的表

## Table of contents



## 先把“可查”升级成“可回看、可解释、可绑定 release”的状态记忆

AI 项目里真正危险的不是“表不存在”，而是你有数据、有索引、有答案，却说不清它们到底对应哪一版状态。


进入课时 2 返回 Week04 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

Week03 已经解决了数据怎样进入系统、失败后怎样 replay / backfill、state 和 report 怎样帮助你恢复。

但这些还没有回答一个更难的问题：

> **未来某一天，当答案变了、坏案例出现了、评测漂了时，你怎么证明这是哪一版数据状态造成的？**

这一讲要建立的核心判断是：

**AI 数据工程需要的不只是“能查的表”，而是“有记忆的表”。**

## 参考学习时间

建议按一节标准课安排：先读完正文，再把 Week03 ingest baseline 和 Week04 snapshot / release 绑定关系写进 lakehouse_foundation_v1.md。

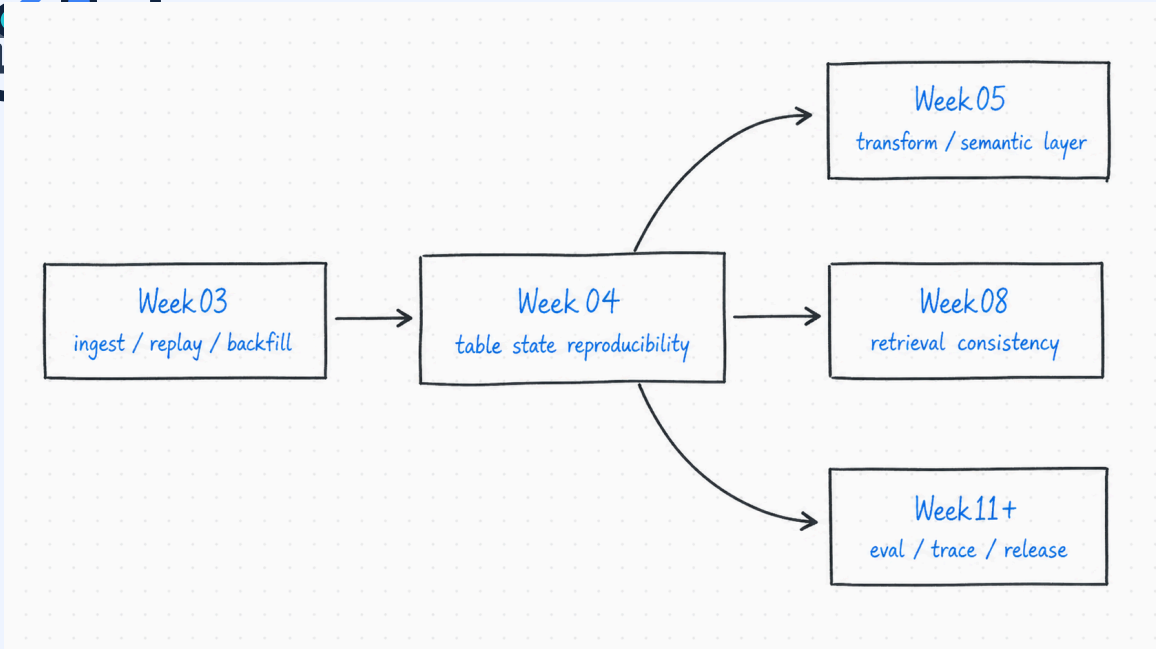
## 学完这一讲，你应该能做到什么

1. 解释为什么 raw bucket、Postgres 当前表和向量索引都不能替代 snapshot-able table。
2. 区分“可查”与“可回看、可解释、可绑定 release”。
3. 用真实事故复盘语言说明 Week04 为什么重要。
4. 写出一份 Week04 目标、边界与状态记忆问题清单。

## 本课产出

- docs/blueprints/week04/lakehouse_foundation_v1.md
- docs/blueprints/week04/state_memory_questions_v1.md

## 先看一张总图




这张图要表达的重点是：

- Week03 回答“数据怎样稳定进来”
- Week04 回答“进来之后怎样保留状态记忆”
- 后面的 semantic / retrieval / eval，都需要站在这层状态记忆之上

## 1. 一个真实风格的 bad case

想象 OmniSupport Copilot 已经在 Northstar Edge Gateway 的支持团队里试运行。周一上午，一位企业客户问：

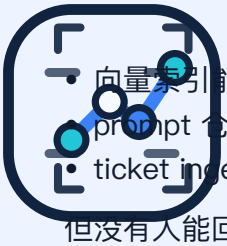
> “升级到固件 5.8.2 之后，网关为什么频繁掉线？要不要回滚？”

Copilot 给出的答案很完整：先检查 PoE 供电，再确认 DHCP 租约，最后建议在维护窗口回滚到 5.7.9。客服同学把答案贴给客户，客户也接受了这个排障路径。

周三下午，同一个客户、同一个设备型号、同一个固件问题，Copilot 的回答变了：它开始建议直接替换硬件，并且没有再提回滚到 5.7.9。业务方问工程团队：“为什么昨天答 A，今天答 B？到底哪一个答案可信？”

这时团队发现，所有系统看起来都“能查”：

- raw bucket 里能找到旧版和新版固件文档；
- PostgreSQL 当前表能查到这条工单和关联设备；



- 向量索引能检索到相关 chunk；
- prompt 仓库也能看到最近一次改动；
- ticket ingest 和 doc ingest 的日志都没有明显报错。

但没有人能回答 4 个关键问题：

1. 周一那次回答到底基于哪一版文档资产？
2. 周三索引重建时消费的是哪一批文档与工单状态？
3. 评测分数变化发生在数据变更之前，还是索引重建之后？
4. 如果要复现周一回答，应该回到哪个稳定的数据状态？

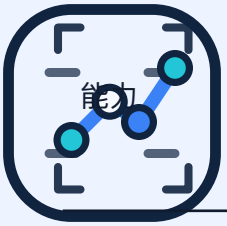
这不是“模型突然坏了”这么简单，也不是“数据能查到”就等于系统可解释。没有 snapshot 锚点时，坏案例复盘会变成猜谜：每个团队都只能拿自己手里的日志解释一段链路，却没有一个统一对象能证明“当时的数据状态是什么”。

这就是 Week04 要解决的核心问题：让进入 Lakehouse 的 Bronze / Silver 表具备可提交、可回看、可绑定后续索引与评测的数据状态记忆。

> ! 本课核心判断
>
> 如果系统不能回答“这个答案基于哪一个数据状态”，坏案例复盘就会变成猜测会。

## 2. 四类常见数据对象能回答什么

| 能力 | 只靠 raw bucket | 只靠 PostgreSQL 当前表 | 只靠 pgvector / 向量索引 | Iceberg snapshot-able table |
|---|---|---|---|---|
| 能不能查当前数据 | 能看到原始文件 | 能查当前业务行 | 能检索当前 chunk | 能查当前表状态 |
| 能不能回看历史状态 | 只能靠文件名、备份和人工约定 | 当前表天然覆盖旧状态，除非另建审计表 | 索引常被重建，旧索引状态不一定可复现 | 可以通过 snapshot / history 回到旧表状态 |
| 能不能绑定评测 / release | 很难表达“这批文件组成哪版表” | 可以记录当前数据，但难绑定历史表版本 | 可以记录索引版本，但不是源数据状态锚点 | 可以把 eval run、index release、data release 绑定到 snapshot |



| 能力 | 只靠 raw bucket | 只靠 PostgreSQL 当前表 | 只靠 pgvector / 向量索引 | Iceberg snapshot-able table |
|---|---|---|---|---|
| 能不能解释答案漂移 | 能说明原文是否存在 | 能说明当前业务状态 | 能说明当前检索结果 | 能说明答案消费的是哪版 Bronze / Silver |
| 适合承担什么角色 | 原始材料文件柜 | 当前业务视图 | 检索索引 | 有提交历史的数据账本 |

这里不是说 raw bucket、Postgres、向量索引不重要。它们的问题是：

**它们各自只能回答一段链路的问题，不能统一回答“这份数据当时长什么样”。**

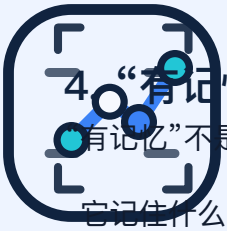
可以这样理解：

- raw bucket 像文件柜，保留原始材料，但很难表达“表状态”。
- PostgreSQL 当前表像正在使用的业务视图，适合服务当前查询，但不天然表达历史版本。
- 向量库像检索索引，负责召回，不应该被当成源数据状态锚点。
- Iceberg table 像有提交历史的数据账本，适合把表状态和后续 release / eval / index 绑定起来。

## 3. “可查”与“可回看”的差异

| 能力 | 只可查的系统 | 有状态记忆的系统 |
|---|---|---|
| 当前查询 | 能查当前数据 | 能查当前数据 |
| 历史回看 | 依赖备份、日志、人工推断 | 通过 snapshot / history 回到旧状态 |
| 评测绑定 | 很难说明分数对应哪版数据 | 可以把 eval run 绑定到 table snapshot |
| 索引重建 | 通常只记录“重建过” | 可以记录索引来自哪个数据状态 |
| release 复盘 | 只能看代码版本和部署时间 | 可以同时看代码、数据、索引、评测状态 |

所以 Week04 的价值不是“学会一个表格式”，而是把 AI 系统的复盘能力从口头判断推进到可验证证据。



## 4. “有记忆的表”到底记住什么

“有记忆”不是一句营销词。放到 Week04，它至少包含 5 类状态证据：

| 它记住什么 | 为什么对 AI 数据工程重要 |
| --- | --- |
| 每次提交后的表状态 | 评测、索引、发布可以绑定到明确 snapshot |
| 当前版本由哪些数据文件构成 | 可以复盘某次回答到底消费了哪些数据 |
| 历史 snapshot 之间的关系 | 可以解释某次写入、追加或演进发生在什么位置 |
| schema / partition 等元数据演进 | 字段变化、分区策略变化不再只靠口头描述 |
| 可以被 time travel / baseline / release 引用的状态锚点 | 后续 Week05、Week08、Week11 才有稳定输入 |

它不等于：

- 自动解决所有数据质量问题；
- 自动让 RAG 回答变好；
- 自动替代 Week02 的 data contract；
- 自动替代评测与 tracing；
- 自动完成治理、回滚、发布审批全流程。

> ⚠ 不要把 Iceberg 当成“数据质量自动修复器”
>
> Iceberg 给你的是表状态记忆和演进边界。数据质量、权限语义、指标口径和发布策略，仍然需要后续课程继续建立。

## 5. 团队复盘时必须问的 8 个问题

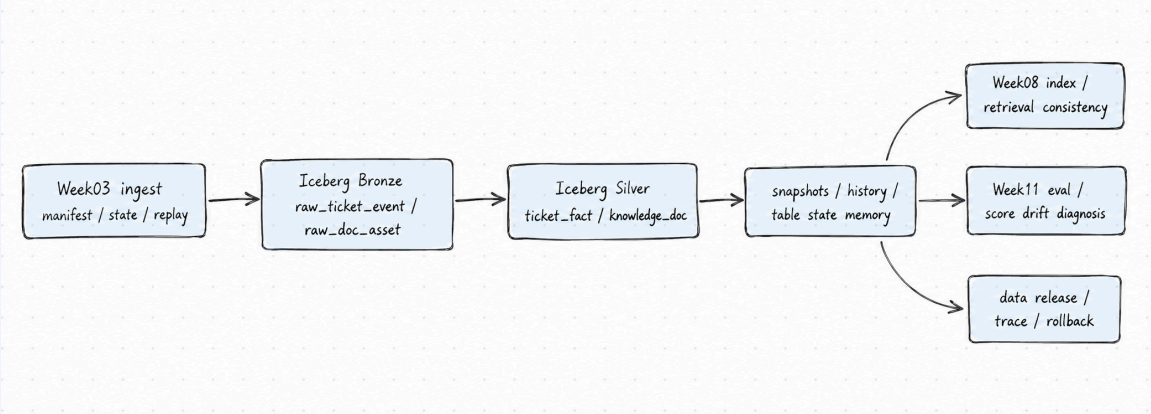
当答案变了、评测掉了、索引异常了，团队不应该先争论“是不是模型问题”，而应该先问：

1. 这次回答基于哪一版 raw 文档?
2. 对应哪一次 ticket / doc ingest batch?
3. 对应哪一个 Iceberg snapshot?
4. 对应哪一版索引?
5. 对应哪一版 Prompt?
6. 评测分数变化发生在哪个数据状态之后?
7. 业务字段或权限字段是否在这期间发生变化?
8. 如果需要回看或回滚，我们能定位到哪个稳定状态?

这些问题就是 state_memory_questions_v1.md 的主体。

Week04 只先解决其中一部分：**数据状态锚点**。Prompt、索引、评测、release 会在后续周次继续接上。

## 6. 从 ingest 到有记忆的数据系统




图里的实线是 Week04 要站住的主线：把 Week03 的 ingest 结果落成 Bronze / Silver 表，并让这些表具备 snapshot、history、schema evolution 和 baseline 证据。

虚线是后续周次会消费的能力：Week08 的索引应该知道自己来自哪版文档表；评测应该知道分数绑定到哪版数据；release 也应该能把代码、索引、数据状态放到同一条证据链里。对小白来说，这张图的价值是把 Iceberg 从“一个湖仓组件”重新放回 AI 系统复盘链路里：它不是为了显得技术栈高级，而是为了让后续每次回答、检索、评测都有可追溯的数据锚点。

## 7. 小白常见误区

| 误区 | 纠偏 |
| --- | --- |
| Iceberg 是大数据平台，和 AI / RAG 无关 | RAG 的索引和评测都需要知道消费了哪版数据，表状态锚点直接影响可复现性 |
| 有对象存储就等于有 Lakehouse | 对象存储只是底座；缺少表格式、metadata、snapshot，就仍然只是文件堆 |
| 有向量库就等于有可复现数据状态 | 向量库是索引，不是源数据状态账本 |
| time travel 就是复制很多份表 | time travel 依赖 snapshot 与 metadata，不是每次复制整表 |
| Week04 做完就完成治理和发布回滚 | Week04 只建立最小状态记忆层，完整治理和 release 会在后续课程继续扩展 |

## 8. Week04 不让你立刻上生产 Iceberg

> ℹ 边界判断
>
> 本周不是让你马上把生产系统迁到 Iceberg，而是先建立 AI 数据状态锚点：snapshot、history、schema evolution、baseline。

本周最小闭环只要求你能解释并演示：

- 表状态可以被提交；
- 历史状态可以被查看；
- 旧状态可以被回看；
- 字段可以在边界内演进；
- 当前 baseline 可以被记录。

## 9. 回到 OmniSupport Copilot 当前 repo

本周内容要对齐当前项目里已经指向 Lakehouse 的对象：

- pipelines/lakehouse/iceberg_schemas.py
- pipelines/lakehouse/assets.py
- infra/docker-compose.yml
- pyproject.toml

如果这些对象还只是 stub，你也不要在课程页面里伪造运行结果。正确做法是：

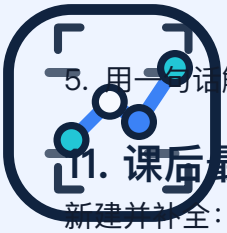
> 待 Week4 项目代码落地后，同步 pipelines.lakehouse.* 的真实命令。

## 10. 本课自检清单

- 我能解释 Week03 与 Week04 的边界：一个是 ingest correctness，一个是 table state reproducibility。
- 我能说清 raw bucket、Postgres 当前表、向量索引各自缺少什么状态记忆能力。
- 我能写出 8 个团队复盘会真正会问的数据状态问题。
- 我没有把 Week04 误写成 dbt、RAG、治理或发布周。

再做 5 道自测：

1. 为什么“能查到当前数据”不等于“能复现当时回答”？
2. raw bucket、PostgreSQL 当前表、向量索引分别应该承担什么角色？
3. 如果 Week08 索引重建后答案变了，Week04 应该提供哪类证据？
4. “有记忆的表”不能自动解决哪些问题？



5. 用一句话解释 table state reproducibility。

## 11. 课后最小行动

新建并补全：

```
docs/blueprints/week04/state_memory_questions_v1.md
```

建议包含 3 段：

1. 当前项目最需要绑定数据状态的 3 类场景；
2. 每类场景需要回答的 snapshot / release / eval 问题；
3. 本周最小 Iceberg 闭环需要先证明哪些能力。

再补一个 5 句话练习：

> 用 5 句话解释“为什么 OmniSupport Copilot 需要有记忆的表”。

要求每句话分别覆盖：业务坏案例、当前对象不足、Iceberg snapshot 价值、后续 RAG / eval 消费、Week04 边界。

## 延伸阅读

- Apache Iceberg / Reliability[1]
- Apache Iceberg / Maintenance – Expire snapshots[2]
- Apache Iceberg / Evolution[3]

[1] Iceberg 官方文档把 valid snapshots、current snapshot、atomic metadata swap、serializable isolation、reliable reads、version history 和 rollback 放在同一组可靠性能力里。这正是本课所说“有记忆的表”的底层依据。Apache Iceberg | Reliability

[2] Iceberg 官方文档说明，每次写入都会创建新的 snapshot，snapshot 可以用于 time travel 或 rollback；但旧 snapshot 也需要被合理过期以控制 metadata 体积。Apache Iceberg | Maintenance

[3] Iceberg 官方文档将 schema evolution、partition evolution 和 sort order evolution 作为 table evolution 的核心能力；本课只把它作为“状态记忆为什么能支撑后续演进”的入口，不展开实现细节。Apache Iceberg | Evolution