Week03｜课时 5｜故障自愈与补数：Replay / Backfill / Runbook 怎么把链路拉回正轨

Table of contents

Week03 的收官，不是“系统终于跑通”，而是故障发生后你知道怎样把链路拉回正轨

这一讲不是在讲完整灾备平台，也不是把所有恢复能力都写成自动化已实现。

先解决更现实的问题：

当 ingest 链路出现缺口、乱序、失败窗口或错误写入时，你有没有一套可复用的 replay / backfill / runbook 基线。

进入实验 回看课时 4 返回 Week03 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

到了 Week03 的最后一课，你已经能看到一条最小 ingest 基线的轮廓了：

- Week02 定义了输入准入边界
- Week03 前四课把这些边界推进成了 batch / incremental / asset flow
- repo 里已经有 seed_loader.py、ticket_ingest.py、doc_ingest.py、assets.py 和 contract tests 这些最小基线[1][2]

[1] OmniSupport Copilot README 已经明确给出 Docker-only / Docker-first 的学生基线，并把 seed_loader dry-run 与 pytest tests/contract/ -v 作为 Week01–Week03 可直接复用的工程入口。OmniSupport Copilot README

[2] 当前 pipelines/ingestion/assets.py 已经定义了 seed_manifests、raw_doc_assets、raw_ticket_events 和 ingest_all_job，并在文件头与注释里明确说明 Week03 起会接入真实采集器、写 MinIO 与 PostgreSQL 元数据。assets.py

但真实系统到了这一步还没有“稳定”它只是**刚刚开始具备可运行性**。真正决定系统能不能长期跑下去的，是下面这类问题：

- 某次 ingest 失败后，应该直接重试，还是改用 replay?
- 补数（backfill）到底是在修历史空洞，还是在重新计算旧分区?
- 某条 source 出了问题，为什么不是把全链路一起重跑?
- 出现缺口后，谁来判断边界、谁来批准、谁来执行、谁来记录?

所以这节课不是再多讲一个工具，而是要让你建立一个更接近生产现场的判断：

故障恢复不是“把任务重新跑一遍”，而是有边界、有依据、有记录的工程决策。

参考学习时间（45–55 分钟）

如果你只阅读正文，大约需要 30–35 分钟；如果你跟着本课一起整理 replay_backfill_strategy_v1.md、ingestion_runbook_v1.md 并做一轮最小 recovery drill，建议预留 45–55 分钟。

学完这一讲，你应该能做到什么

1. 解释 retry / replay / backfill / rerun 之间的区别。
2. 理解为什么 Week03 现在就要把 recovery thinking 和 runbook 带进 repo。
3. 能结合当前 OmniSupport Copilot 基线，说清楚 recovery 最少依赖哪些追踪锚点。
4. 能在仓库里写出一份可用的《采集链路 Runbook》v1。
5. 能完成一次最小 recovery drill，并把过程记录成 recovery_drill_report.md。

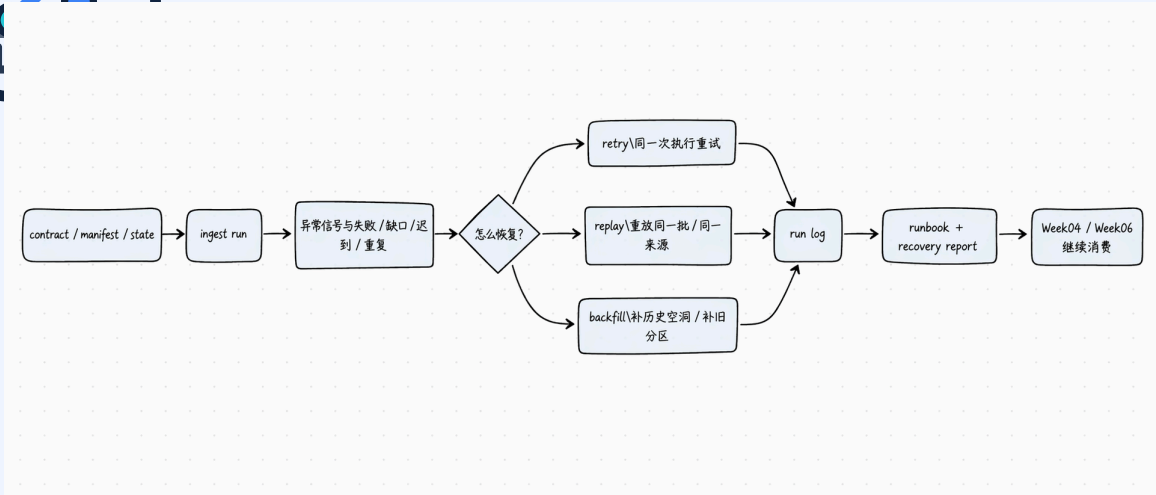
本课产出

完成这一讲后，你至少应该产出 3 个工件：

- runbooks/ingestion_runbook_v1.md
- docs/blueprints/week03/replay_backfill_strategy_v1.md
- reports/week03/recovery_drill_report.md

> 注意：这节课的重点不是再新造一套 replay 平台，而是把 **恢复规则、决策边界和操作顺序** 先收口成团队可执行资产。

先看一张总图



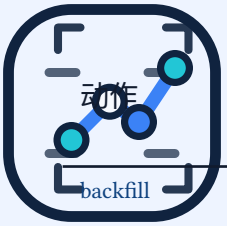
这张图里最重要的不是 3 个英文单词，而是中间那个判断节点：

怎么恢复，不应该靠拍脑袋，而要靠 contract、manifest、state、run log 一起做决策。

1. 先把 5 个最容易混淆的动作拆开

很多系统到了出故障时才发现，团队内部其实连术语都没统一。 所以先把 5 个动作拆开。

动作	它真正解决什么	什么时候最适合用	最容易被误用成什么
retry	同一次执行里的瞬时失败	网络抖动、短暂锁冲突、可安全重试的写入	被误当成一切恢复手段
rerun	把同一个步骤或同一个 job 再跑一次	逻辑没变，只是希望重新执行	被误以为已经等于 replay
replay	重新消费同一批次 / 同一来源 / 同一段变更	想验证幂等、重走 ingest、重建中间结果	被误用来补历史缺口
restore	把系统或数据恢复到某个已知可用状态	状态库损坏、对象存储污染、已知好版本需要快速回退	被误当成 replay / backfill 的替代品



动作	它真正解决什么	什么时候最适合用	最容易被误用成什么
backfill	为历史空洞或过往分区补数据 / 重算	历史日期缺失、旧逻辑修复后要补算历史	被误写成“重新全量跑一次”

Dagster 官方文档里把 backfill 定义得很清楚： 它是为**缺失的分区**或**需要更新的已有记录**重新 materialize；而 partitions 本身是把数据和计算切成逻辑片段，用来支持 incremental processing。[3][4]

restore 为什么要单独看

restore 和 replay / backfill 最大的差别在于：

- replay / backfill 仍然是在 **重新计算或重新消费**
- restore 更像 **把系统先拉回某个已知可用快照或备份点**

如果对象存储、状态表或下游表已经被错误写入污染，直接 replay 往往不够。 这时你先要判断：

- 是不是先 restore 到上一个可信状态
- 再决定后续要不要 replay 某批输入
- 或者再做 targeted backfill 补历史空洞

这句话先记住

- retry 更像执行级恢复
- replay 更像输入级重放
- restore 更像系统级回退到已知好状态
- backfill 更像历史空洞修复
- rerun 是更泛的“再执行一次”

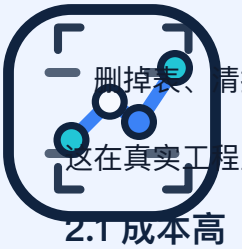
如果这 5 个词不先分清，后面 runbook 一定会写乱。

2. 为什么“出故障就全量重跑”不是一个成熟答案

在课堂 demo 里，最简单的恢复动作通常是：

[3] Dagster 官方文档说明，backfilling 是为缺失分区或需要更新的已有记录重新 materialize，可以针对全部或部分 partitions 执行。Dagster Docs | Backfilling Data

[4] Dagster 官方文档说明，partitioning 是把数据切成逻辑片段的技术，用于支持 incremental processing。Dagster Docs | Partitioning Assets



删掉表、清掉目录、再全量跑一次。

这在真实工程里往往不是一个好答案。原因有 4 个：

2.1 成本高

有些源体量不大，但有些源一旦规模起来，全量重跑会让你：

- 重新读取大批无变化数据
- 重新写入对象存储和数据库
- 重新触发下游 parse / index / eval
- 拉长恢复时间

2.2 风险大

全量重跑常常意味着：

- 覆盖掉你想保留的旧结果
- 把边界不清的 source 再次放大
- 引入新的重复或错序风险

2.3 可解释性差

如果你最后只说“我把它全跑了一遍”，那后面很难回答：

- 到底是哪一段有问题
- 为什么这次只修了这里
- 影响面多大
- 是否真的补全了空洞

2.4 不利于形成 runbook

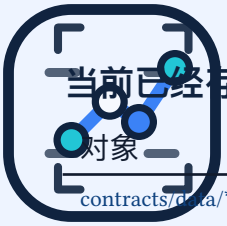
真正可交付的 runbook 一定会问：

- 触发条件是什么
- 操作边界是什么
- 用哪个命令 / 哪个参数 / 哪个 manifest
- 结果怎么验收

而“全量重跑”通常很难写成精确的恢复流程。

3. 当前 repo 里已经具备哪些恢复前提

这节课不需要你再造第二套平台。 相反，你要先认清当前 repo 已经有了什么，哪些恢复能力可以从今天开始建立。



当前已经存在的恢复前提

对象	当前 repo 已有的价值
contracts/data/*.json	定义输入边界与必需字段，是恢复前的契约基线
data/seed_manifests/*.json	提供 source 的最小声明边界，是 scoped replay 的自然入口
pipelines.ingestion.seed_loader	可以从 manifest 目录执行最小 ingest baseline，是恢复 drill 的起点[5]
pipelines/ingestion/assets.py	已经把 ingest 组织成 asset graph，是后续 partition / backfill 的天然挂点[6]
tests/contract/test_json_schemas.py	已经能先验证 contract / manifest 的结构合法性，避免“坏配置也进入恢复流程”

当前还没有 fully automated 的部分

也要诚实说清楚，Week03 现在还没有 fully automated 的：

- 通用 replay service
- 自动 checkpoint / state manager
- 一键 backfill engine
- recovery policy engine

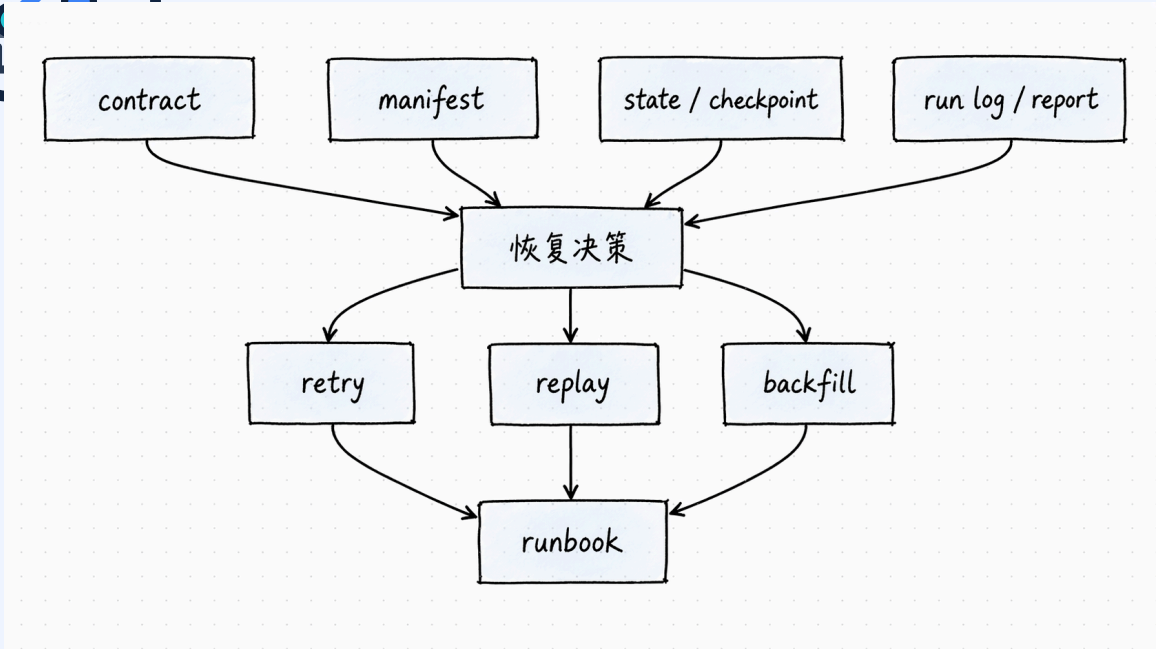
这不是缺点，而是课程分阶段实现的正常状态。

> ℹ 课程边界判断
>
> Week03 的目标不是“本地一次性做完所有恢复自动化”， 而是先把 **恢复决策、恢复边界、恢复记录、恢复手册** 变成可持续演化的工程对象。

[5] OmniSupport Copilot README 已经明确给出 Docker-only / Docker-first 的学生基线，并把 seed_loader dry-run 与 pytest tests/contract/ -v 作为 Week01–Week03 可直接复用的工程入口。OmniSupport Copilot README

[6] 当前 pipelines/ingestion/assets.py 已经定义了 seed_manifests、raw_doc_assets、raw_ticket_events 和 ingest_all_job，并在文件头与注释里明确说明 Week03 起会接入真实采集器、写 MinIO 与 PostgreSQL 元数据。assets.py

4. 再看一张图：恢复真正依赖哪些锚点



这一张图里，最容易被忽略的是：

manifest

没有 manifest，你就很难界定： – 这次到底恢复哪一类 source – 恢复哪个窗口 – 恢复哪个来源清单

state / checkpoint

没有 state，你就很难知道： – 上次成功到了哪里 – 本次 replay 从哪里开始 – backfill 是否已经覆盖了缺口

run log / report

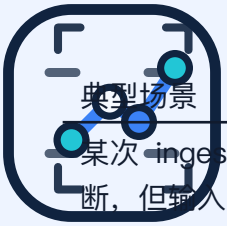
没有 run log，你就很难回答： – 这次为什么失败 – 失败在 contract / data / system 哪一层 – 后来是 retry 还是 replay – 恢复是否真的成功

所以这节课不是“写 Runbook 文档课”，而是要让你看到：

Runbook 只是表面，真正的恢复能力来自前面这些锚点是否被工程化。

5. 什么时候该 replay，什么时候该 backfill

这一段最适合你直接拿去做团队内部的判断表。



典型场景	更合理的动作	为什么
某次 ingest 因网络抖动中断，但输入边界和时间窗口没变	retry / rerun	更像执行层失败，不需要重新解释输入边界
某个 manifest 已知合法，但想重新走同一批数据看是否幂等	replay	目标是重放同一来源 / 同一批次
某个日期分区或历史窗口压根没入湖	backfill	目标是补历史空洞，而不是重放同一批
contract 变了，需要重新计算历史结果	backfill	这是逻辑变更后的历史重算
某次 replay 之后仍然发现少数据	先查 state / run log，再决定是否转 backfill	避免一上来把问题扩大化

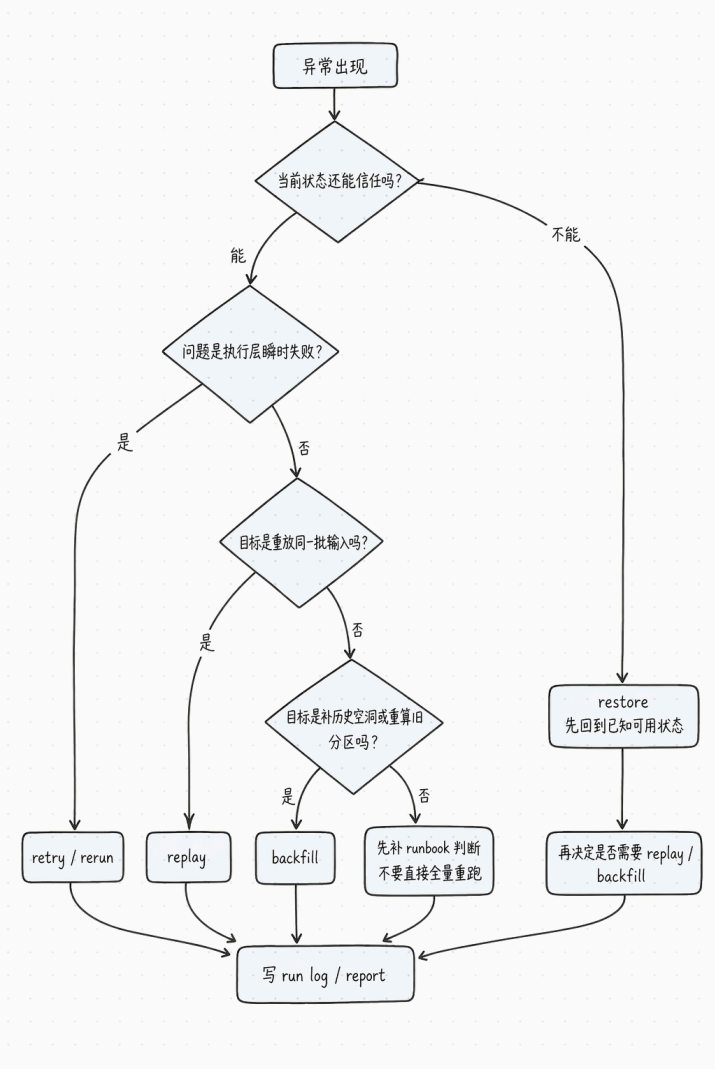
Dagster 文档也正是按这个思路来组织：partitions 用于把数据切成可管理的逻辑片段，而 backfill 则针对这些片段做历史补齐或重算。默认一个 backfill 覆盖 N 个分区时会提交 N 个 runs，也可以通过 backfill policy 采用更批量化的策略，不同策略在开销、隔离性和资源利用之间有取舍。[7][8][9]

[7] Dagster 官方文档说明，partitioning 是把数据切成逻辑片段的技术，用于支持 incremental processing。Dagster Docs | Partitioning Assets

[8] Dagster 官方文档说明，backfilling 是为缺失分区或需要更新的已有记录重新 materialize，可以针对全部或部分 partitions 执行。Dagster Docs | Backfilling Data

[9] Dagster 官方文档给出了多种 backfill strategy，对比 one-run-per-partition、batched 和 single-run 在开销、隔离性和资源利用上的差异。Dagster Docs | Partition Backfill Strategies

再用一张决策树把恢复路径收紧



这张图最重要的不是把所有动作背下来，而是先守住一个顺序：

1. **先判断当前状态还能不能信任**
2. 再判断这是执行层失败、输入重放，还是历史补齐问题
3. 最后才决定动作，而不是一上来就整链路重跑

6. 这节课最重要的实践：做一次最小 recovery drill

这次实践不会让你实现一个完整 replay service。 它会做两件更重要的事：

1. 用当前 repo 已有基线做一次**有边界的恢复演练**
2. 把恢复决策写进**可执行的 Runbook**

6.1 你要准备的 3 个文件

24 课 starter pack 里只提供这 3 个模板：

- runbooks/ingestion_runbook_v1.md
- docs/blueprints/week03/replay_backfill_strategy_v1.md
- reports/week03/recovery_drill_report.md

6.2 你的 recovery drill 最少要包含什么

建议按下面顺序做：

第一步：先确认当前 baseline 可跑

```
docker compose --env-file infra/env/.env.local -f infra/docker-compose.yml up -d --build

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

docker compose --profile tools --env-file infra/env/.env.local -f infra/docker-compose.yml run --rm devbox \
  python -m pipelines.ingestion.seed_loader --manifest-dir data/seed_manifests
```

这一段的作用不是“重新跑一次实验”，而是给恢复演练确认一个干净的起点。[10]

第二步：人为定义一个恢复场景

建议你从下面 3 类里选 1 类：

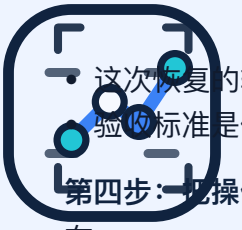
恢复场景	你要练什么
某个 manifest 暂时跳过了一个 source	学会 scoped replay
某个历史批次需要补回	学会 backfill 边界定义
某次变更后 contract 结构仍合法，但历史逻辑需要重算	学会逻辑变更后的 recovery 说明

第三步：写恢复决策，而不是直接动手跑

在 docs/blueprints/week03/replay_backfill_strategy_v1.md 中至少写清：

- 场景描述
- 当前症状
- 目标动作是 retry / replay / backfill 中哪一个
- 为什么不是另外两个

[10] OmniSupport Copilot README 已经明确给出 Docker-only / Docker-first 的学生基线，并把 seed_loader dry-run 与 pytest tests/contract/ -v 作为 Week01–Week03 可直接复用的工程入口。OmniSupport Copilot README



- 这次恢复的输入边界是什么
- 验收标准是什么

第四步：把操作写成 Runbook

在 runbooks/ingestion_runbook_v1.md 中至少写清：

- 触发条件
- 前置检查
- 执行命令
- 风险提醒
- 验证方式
- 回滚 / 人工升级路径

第五步：写一份 recovery drill 报告

在 reports/week03/recovery_drill_report.md 中至少记录：

- 本次 drill 采用什么恢复动作
- 为什么这么选
- 执行了哪些命令
- 预期与实际是否一致
- 还有什么自动化能力尚未具备

6.3 这次练习真正训练的不是脚本，而是恢复判断

这一步最容易被误解成“只是在写文档”。

其实不是。

你现在做的是把下面这些事情一次性绑定起来：

- contract boundary
- manifest scope
- ingest baseline
- Dagster asset / partition thinking
- recovery decision
- runbook literacy

这正是后面 Week04 / Week06 / Week12 会持续消费的能力。

7. 为什么 Runbook 不是“最后才写的运维文档”

很多团队会把 Runbook 理解成：

上线前最后补一份文档。

这个理解在数据系统里很容易出问题。

真正成熟的 Runbook 应该从你第一次做 recovery drill 时就开始出现。原因有 3 个：

7.1 它会逼你把术语讲清楚

如果你写不清： – retry 和 replay 的区别 – replay 和 backfill 的边界 – 谁批准恢复 – 恢复成功的验收标准

说明系统里这些概念本来就没统一。

7.2 它会逼你暴露缺失的系统能力

如果你写 Runbook 时发现： – 没有 state – 没有 batch_id – 没有 manifest scope – 没有 run log

这不是 Runbook 写不好，而是系统能力还没长出来。

7.3 它会直接缩短未来的故障恢复时间

等 Week12 做 tracing、Week14 做治理时，你会发现：

Runbook 越早写，后面系统越容易形成“定位 –> 决策 –> 恢复 –> 验证”的闭环。

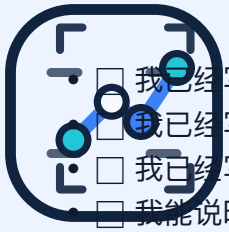
8. 这节课最重要的 8 个判断

1. retry / rerun / replay / restore / backfill 不是同义词。
2. recovery 是工程决策，不是“重新跑一下”的冲动。
3. 没有 contract / manifest / state / run log，就很难做可解释恢复。
4. replay 更像重放同一批输入，backfill 更像补历史空洞或重算历史分区。
5. Week03 现在讲 recovery，不是为了马上把平台做完，而是为了先把恢复边界讲清楚。
6. Dagster 里的 partition / backfill 思维，现在就该进入学生心智，而不是等 Week06 再第一次听到。
7. 当前课程主线最合理的目标是：把 recovery 决策和 runbook 写扎实，而不是假装本地已经 fully automated。
8. 一份好的《采集链路 Runbook》本身就是 Week03 最重要的交付物之一。

9. 自检清单

学完这一讲后，你至少应该能勾掉下面这些项：

- ☐ 我能说清 retry / rerun / replay / restore / backfill 的区别
- ☐ 我知道当前 repo 已经有哪些恢复前提，哪些还没有 fully automated
- ☐ 我能解释为什么恢复决策要依赖 contract / manifest / state / run log



- [ ] 我已经写出 replay_backfill_strategy_v1.md
- [ ] 我已经写出 ingestion_runbook_v1.md
- [ ] 我已经写出 recovery_drill_report.md
- [ ] 我能说明这节课如何接到 Week04 和 Week06

10. 课后最小行动

如果你今天时间有限，至少先完成下面两步：

1. 把 runbooks/ingestion_runbook_v1.md 模板补完整
2. 选一个场景，在 replay_backfill_strategy_v1.md 中明确写出：
   - 这次为什么选 replay 或 backfill
   - 为什么不选另一个
   - 成功验收标准是什么

只要你把这两件事做完，Week03 的恢复思维就已经真正立住了。

延伸阅读

- Dagster / Partitions and Backfills
- Dagster / Backfilling Data
- Dagster / Partition Backfill Strategies
- OmniSupport Copilot README
- OmniSupport Copilot pipelines/ingestion/assets.py
- OmniSupport Copilot pipelines/definitions.py