AIT-NATIVE / 实用产品指南

用户手册

与版本绑定的安装、概念、实用工作流、精简 CLI 指南、集成、运维与故障排查，对应 ait-native 1.1.1。

版本	文档
1.1.1 / 1.1.1	修订 4
内容	范围
44 个导读章节	安装、工作流、 集成、运维

ait-native.dev/technical/
Published 2026-09-02

目录

点击页码可在 PDF 内跳转。每一章也链接到它确切的在线页面。

开始

技术文档

ait-native 概念、工作流、命令、集成与运维的版本化入口。

适用人群 开发者与运维人员

https://ait-native.dev/technical/v1.1.1/

挑能回答问题的最短那一页

这份文档是基础 文档 页的技术篇，固定对应 ait-native 1.1.1，把日常使用、agent 负责的工作流命令、集成 API 和远程操作分开来讲。

- 要给一个仓库接上 AIT，先看 上手指南。
- 遇到看不懂的 AIT 记录或标识符，去 核心概念 补背景。
- 打开 系统架构与权威边界， 看本地编写、远程执行和原生嵌入这三条路怎么接到同一个工作流内核，又各自不共享权威。
- 想查 1.1.1 的每一条公开命令，用 完整 CLI 参考； 只想看某个工作流下的有限例子，翻更短的那几页面向工作流的 CLI 文档。
- 要看冻结的 token、经过时间、置信区间和协议边界，读 性能评测。
- 只有当 server 和 runner 确实能解决具体的协作需求时，才打开 远程基础设施。
- 要实现或审计固定记录、载荷、索引、恢复或格式转换，查 完整的 Binary DB v0 schema 这一页。

运作模型

用户在仓库里初始化一次，然后描述想要的结果。coding agent 读生成的 仓库指示，把受管控的工作收敛到某个确切的 sprint 条目，在绑定的 工作区里实现，记录一个 Snapshot，最后走完配置好的收尾路径。

```
request and constraints
  -> exact sprint item
  -> Task-bound implementation
  -> repository checks
  -> immutable Snapshot
  -> configured closeout
```

本地工作流不需要跑起一个 server。远程权威和 CI 都属于明确另加的 部分。

版本边界

本节每一页都绑定到公开的 1.1.1 源码 tag。版本面板会链到核对这份公开文档时用到的那份源码、Release、分发契约，以及 CLI 解析器。

开发主线的行为可能比这几页新。如果某个版本的命令和这份文档说的 对不上，请以你实际安装的那个版本的文档为准。

接下来看哪里

- 功能工作流
- 回归工作流
- 性能评测
- 分发与发布
- 故障排查

开始

快速开始

安装 1.1.1、初始化一个仓库、提出需求，然后查看记录下来的结果。

适用人群 开发者

https://ait-native.dev/technical/v1.1.1/getting-started/

1. 安装当前版本

从 快速上手 里挑一条验证过的 1.1.1 安装路径。 初始化仓库之前，装好的原生命令必须报出预期的产品版本。

```
ait --version
```

PyPI、npm、Homebrew、apt 和 GitHub 上的构建产物，打包外壳各不相同， 但通向同一套仓库工作流。server 在明确配置并启动之前，一直是不运行 的。

2. 初始化一个仓库

在 coding agent 要改的那个仓库里执行初始化。

```
cd <your-repository>
ait init
```

AIT 会建立仓库权威，并在 `AGENTS.md` 里生成或刷新当前生效的工作流区块。它不替你选编程语言、框架、构建命令或测试命令。

3. 照常提需求

直接描述想要的结果和约束，不用把活儿翻译成 AIT 的一套记账术语。

```
■■■■■■■■■■■■■■■■■■
■■■■ API ■■■■■■■■■■■■■■■
```

coding agent 按仓库里生成的指示走。对于受管控的工作，这套指示包括 sprint 条目、Task、绑定的工作区、仓库检查、Snapshot，以及配置好的 收尾。

4. 查看结果

需要直接证据的时候，用这些只读命令。

```
ait status --json
ait diff --stat
ait queue summary
```

收尾成功后工作区是空的，这很正常。完成的 Task 和 Snapshot 保留着 记录下来的结果。

下一步

- 核心概念
- `ait init`
- `ait status`
- 功能工作流

开始

核心概念

理解那些把意图、隔离的工作、不可变状态、证据与收尾串起来的仓库对象。

适用人群 开发者与集成方

https://ait-native.dev/technical/v1.1.1/concepts/

Repository 与 Line

Repository 是某个工作目录在本地的 AIT 权威。Line 是一个有 名字、可移动的指针，指向已被接纳的 Snapshot。`main` 是常见的初始 Line；每个 Task 会拿到自己的功能 Line 和绑定的工作区。

Plan 条目

Plan 记录带版本的 Markdown 沿革。在 sprint 模式下，一个稳定的 `[plan-ref: ...]` 标识整份文档，而带有确切 `[ref: ...]` 的那个未 勾选清单项，就是可以开成 Task 的单位。

```
# Password Visibility [plan-ref: login/password-visibility/root]

- [ ] Add the accessible control and focused tests. [ref: login/password-visibility/implement]
```

条目写的是要达成的结果，不是让你顺手做几件互不相干的改动。

Task、Change 与工作区

Task 是绑定 Plan 意图的有界工作单元，Change 是它可供评审的 改动沿革。Task 绑定的工作区把实现和目标 Line 隔开，直到结果被 接纳为止。

启动 Task 的那条命令会打印出确切的工作区路径。agent 必须在那里 改代码，不能在仓库的规范根目录里改。

Snapshot

Snapshot 是不可变的仓库状态，带作者信息和父级历史。它记录的是
完成后的代码状态；它不会为了把文档沿革藏进代码历史，就顺带塞进一处 无关的 Markdown Plan 改动。

Patchset 与证据

在有远程支撑的工作流里，Patchset 是某个 Change 被选中的评审
候选。CI、attestation、评审和策略这几类证据，都在最终收尾之前挂到这个候选 上。

纯本地的工作不需要远程基础设施。但它配置的接纳策略照样决定了完成 之前必须验证哪些东西。

收尾

收尾会推进配置好的目标 Line，完成 Task，清理绑定的工作区，并执行 仓库的 sprint
清单策略。远程工作可能要在代码收尾之后单独再做一步： 更新最后那个 Markdown 复选框，并同步 Plan。

相关参考

接着看 并行 Task 隔离， 或者打开 `ait plan`、 `ait task`、 `ait snapshot` 和 `ait workflow` 查具体的命令说明。

开始

系统架构与授权边界

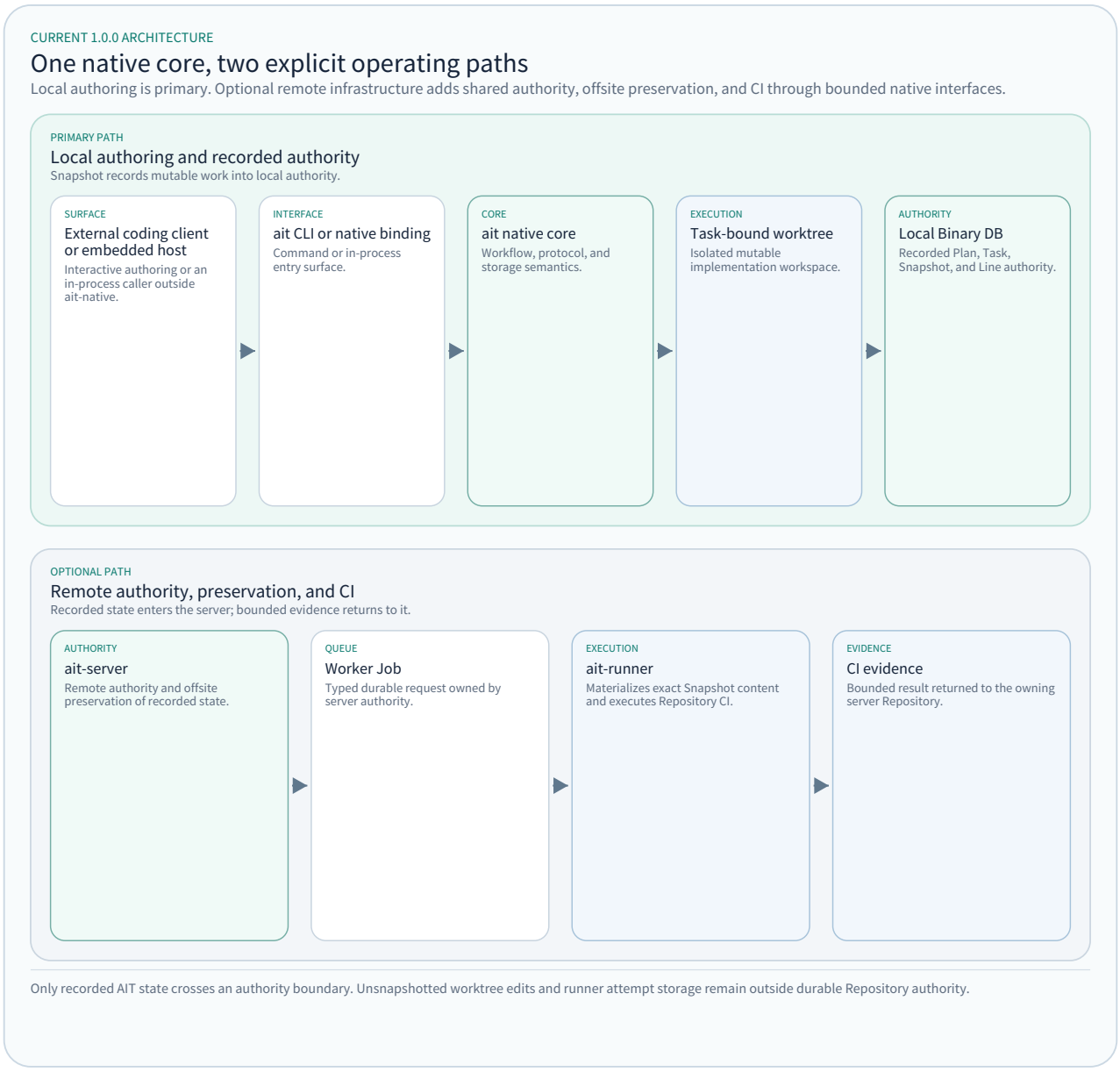
看清外部 coding client、原生绑定、ait-native 内核、Task worktree、Binary DB、服务器与 runner 如何跨本地与可选远程授权连接。

适用人群 开发者、集成方与运维人员

https://ait-native.dev/technical/v1.1.1/system-architecture/

按边界来读这套系统

ait-native 只有一个原生工作流内核，外加两条明确的运行路径。常走的 是本地这条：外部 coding client或嵌入式宿主调用原生接口，实现发生在 Task 绑定的 worktree 里，再由一个 Snapshot 把结果记进本地的 Binary DB 权威。远程基础设施是可选的，而且只能通过明确的操作，去 扩展已经记录下来的状态。



组件与权威对照

组件边界	它拥有什么，又不拥有什么
外部 coding client与 Python/Node 宿主	负责 ait-native 之外的交互式或嵌入式使用体验。它们调用 AIT 接口，但不会取代原生的工作流或存储语义。
`ait` CLI、原生绑定与原生内核	负责命令接纳、工作流状态迁移、协议行为，以及对已记录权威的访问。Python 和 Node 是在进程内进入同一个内核，而不是另实现一套工作流。
Task 绑定的 worktree 与本地 Binary DB	worktree 是一个隔离的、可变的实现工作区。Binary DB 拥有记录下来的 Plan、Task、Change、Snapshot、Line 和相关本地状态。Snapshot 就是可变编辑与持久 AIT 历史之间的那条界线。
`ait-server`	拥有已注册的远程 Repository 权威、共享的远程工作流状态、带类型的 Worker Job、证据，以及对送达服务端的记录状态做异地保存。它不保存没进过 Snapshot 的本地改动。
`ait-runner`	认领一个兼容的 Worker Job，物化出确切的 Snapshot 内容，在隔离的一次尝试中执行 Repository 声明的 CI 入口，然后返回有界的证据。它永远不会变成 Repository 权威。

主要的本地编写路径

理解需求、写出实现，这些仍然是 coding client的事。AIT 给这份工作 提供的是稳定的仓库身份，和一条隔离的变更边界：

1. 客户端通过 CLI 或受支持的原生绑定调用 `ait`。
2. 原生内核解析出确切的 Repository、Plan 条目、Task、Change 和 Task 绑定的 worktree。
3. 在创建 Snapshot 之前，代码改动一直是那个 worktree 里的可变 状态。
4. Snapshot 和工作流记录进入本地 Binary DB 权威；配置好的收尾路径 随后就能基于这份记录状态做判断。

多个 agent 之间互不干扰，因为每个活动的 Task 各占一个 worktree 和 一把命令锁。`AIT_RAM` 可以加快放置速度，但它不改变权威边界：没进 Snapshot 的脏文件属于工作区状态，已记录的 Snapshot 才是可恢复的 AIT 状态。参见 并行 Task 隔离。

可选的远程权威与 CI 路径

本地记录下来的状态，只有通过明确配置的远程操作才会到达 `ait-server`。一旦收下，服务端就拥有远程的 Repository 副本、共享的 工作流状态、证据和 Worker Job 队列。这样就能远程管理，也能对送达 服务端的那份确切状态做异地恢复。

要跑 CI，服务端会提交一个带类型的 Worker Job。`ait-runner` 认领这个 job，把它引用的 Snapshot 物化到隔离的尝试存储里，运行 Repository 声明的入口，然后返回有界的结果。服务端记录由此产生的证据；runner 的尝试目录用完即弃，不构成第二个权威。

远程保存不等于自动故障切换，也没法还原那些从未被捕获、从未传出去的 脏文件或未跟踪文件。参见 远程基础设施 和 `ait-runner`：原生 CI 执行面。

一个内核，通过原生嵌入复用

`ait-python` 和 `ait-node` 在进程内加载已接纳的原生运行时。它们给 应用提供面向业务的入口，但不会另起一个工作流引擎，也不会重新定义 仓库权威。嵌入式应用同样得选定确切的 Repository 上下文，并遵守和 CLI 一样的 Task、Snapshot、远程和策略边界。

这张图有意只画到当前 1.1.1 的产品边界。Release 家族的组件、版本 证据和包的构成，仍然由 分发与发布 定义。

开始

性能评测

查看冻结的 AIT 与 Git worktree 各 100 个 session 结果、置信区间、验收披露与并发边界。

适用人群 开发者、评估人员与技术决策者

https://ait-native.dev/technical/v1.1.1/benchmark/

已发布结果

冻结的 1.1.0 campaign 在五个游戏开发 workload 上比较了 AIT Task worktree 与 Git worktree。分析纳入 100 个 AIT session 和 100 个 Git session；每个 workload 有 20 组配对，两种 treatment 都是 100/100 通过验收。

主要结果给每个 workload 相同权重：

- workload 中位数 token 节省：34.95%；
- bootstrap 95% 置信区间：27.85% 至 39.77%；以及
- workload 中位数经过时间节省：21.04%。

Provider token 汇总总量是描述性交叉检查，不是主要估计量：AIT 使用 `46,300,272` token，Git 使用 `70,140,925` token；在本次 campaign 中， AIT 的 token 少 33.99%。

各 workload 结果

Workload	AIT token 中位数	Git token 中位数	Token 节省	95% 置信区间	配对
GD-01	213,481.0	317,517.8	32.77%	22.07-41.92%	20/20
GD-02	266,652.6	428,522.2	37.77%	27.59-45.66%	20/20
GD-03	376,821.3	496,364.8	24.08%	12.41-34.32%	20/20
GD-04	580,373.9	915,416.8	36.60%	25.68-45.84%	20/20
GD-05	877,684.7	1,349,224.8	34.95%	22.36-44.82%	20/20

冻结分析中，每个 workload 的区间都保持在零以上。

协议边界

两种 treatment 使用相同的 GPT-5.6 Sol 模型和 maximum reasoning、相同的 冻结 fixture、相同的任务 prompt 和功能验收检查、全新 session，以及 provider 报告的 token 计数。Session 采用线性执行；本次 campaign 没有测量高并发吞吐量或扩展性，也不保证每个仓库、模型或任务都有相同节省。

验收披露

Campaign 为 200 个纳入分析的观察值执行了 201 个 session。原始 AIT GD-05 尝试没有保留 `soundEnabled: 0`，因此未通过冻结的功能检查。该失败 保留在 run ledger 中，按冻结验收规则排除，并在相同模型与 fixture pin 下补跑一次。替补通过，并成为纳入分析的观察值。

Task-driven 并发是另一项能力

AIT 以 Task 作为 agent 工作单元。独立 Task 各自拥有 Change、功能 Line 和 worktree，因此命令可以跨多个 agent session 并发执行。对共享目标 Line 的接纳会重新校验并串行进行。这能防止过期写入目标，但不会让多个执行者同时写同一个 worktree 变得安全。

命令与权威边界见并行 Task 隔离。 这项产品能力不属于上面的线性 benchmark 结果。

冻结证据

- 可读摘要

- 机器可读结果
- Session 运行明细
- 发布校验和

工作流

功能工作流

跟着一个有界的功能需求，从 sprint 条目一路走完实现、验证与收尾。

适用人群 开发者与 coding agent

https://ait-native.dev/technical/v1.1.1/workflows/feature/

场景

功能这个例子跟交互式 Demo 演的是同一件事：加一个可访问的密码可见性 控件，同时保住登录 API 和键盘行为。

需求由用户提出。仓库工作流由 coding agent 跑完。

圈定范围

agent 写一张 sprint card，上面只放一个确切的、可开成 Task 的条目。

```
# Password Visibility [plan-ref: login/password-visibility/root]

- [ ] Add the accessible control and focused tests. [ref: login/password-visibility/implement]
```

然后从这个确切条目启动绑定的 Task。

```
ait task start \
  --from docs/sprints/password_visibility.md#login/password-visibility/implement \
  --intent "Add accessible password visibility control"
```

输出会给出 Task、Change 和工作区路径。

在绑定的工作区里实现

agent 进到打印出来的那个路径，改控件、改对应的测试，再跑仓库自己那 套验证。AIT
不猜框架，也不替你编一条测试命令。

```
src/LoginForm.tsx
tests/LoginForm.test.tsx
3 tests passed
```

记录结果

检查通过之后，agent 把代码状态记下来。

```
ait snapshot create --message "Add accessible password visibility control"
```

有远程支撑的工作接着用引导式的就绪命令，直到选中的 Patchset 拿齐 CI、
attestation、评审和策略这几类证据，再用评审者持有的 finish 入口完成原子 远程收尾。

```
ait workflow ready <change-id> --apply
ait workflow finish <change-id> --apply
```

收尾并核对

agent 用仓库生成指示里给的那条 Task 命令收尾，然后核对 Task、工作区、 Line 和 sprint 条目是否都已关闭。

```
ait task finish <task-or-change-id>
ait task audit <task-id>
```

这个例子已经创建最终 Snapshot，所以 `task finish` 复用干净的 Line head。 如果本地工作仍是 dirty，就用 `ait task finish <task-or-change-id> --message <message>`，让同一个操作先创建 最终 Snapshot，再完成收尾。

想看同一套流程跑起来，用交互式 Demo；命令细节打开 `ait task` 参考。

工作流

回归工作流

定位出问题的修订，隔离修复，补上回归测试，并完成新的 Task。

适用人群 开发者与 coding agent

https://ait-native.dev/technical/v1.1.1/workflows/regression/

场景

现有的密码控件对键盘触发不再正确响应。修复既要保住原本预期的行为， 还要补一个能在出问题那次修订上失败的测试。

定位出问题的状态

在决定怎么修之前，先看一眼相关的那一行，或者一段有界的范围。

```
ait blame src/LoginForm.tsx --line 84
```

结果会把这行源码和它对应的 Snapshot、以及能拿到的 Plan 沿革串起来。
这些证据回答的是行为什么时候进的历史；怎么修，它不替你定。

新建一个修复 Task

给这次回归记一条新的 sprint 条目，别去重开已经完成的工作。

```
# Password Keyboard Regression [plan-ref: login/password-keyboard-regression/root]

- [ ] Restore native keyboard activation and lock it with a regression test. [ref:
  login/password-keyboard-regression/repair]
```

```
ait task start \
  --from docs/sprints/password_keyboard_regression.md#login/password-keyboard-regression/repair
    \
  --intent "Restore password-control keyboard activation"
```

证明修好了

agent 只动划定范围内的那点行为，加上回归测试，然后在 Task 工作区里跑 仓库的检查。

```
4 tests passed
```

它会记一个新的 Snapshot，后面走的就绪和收尾路径，跟其他受管控的改动 一模一样。

```
ait snapshot create --message "Restore password-control keyboard activation"
ait task finish <task-or-change-id>
```

兜底规则

如果 blame 给出的证据不全，或者修着修着超出了说好的范围，就停下来， 重新划 Task
的范围。别覆盖工作区里不相干的改动，也别把一个已完成的 Task 当成还在进行的工作重新解读。

参见 `ait blame` 和 故障排查。

CLI · 使用与查看

ait init

创建或重新初始化 AIT 仓库权威与生效的 agent 指令。

适用人群 用户

https://ait-native.dev/technical/v1.1.1/cli/init/

角色

一般由用户在 coding agent 要改的那个仓库里跑一次。

用途

`ait init` 会补上缺失的本地仓库权威，并在 `AGENTS.md` 里生成或刷新当前 生效的 AIT 工作流区块。重新初始化会保留已有的有效权威，而不是把它换掉。

语法

```
ait init
ait init --json
```

值得留意的可选输入包括：初始仓库名、默认 Line、策略档案、作者模式，以及针对已有结构不完整时的有界修复开关。头一次用，一个选项都不用加。

示例

```
cd ~/src/example-app
ait init
```

跑成功之后，打开 `AGENTS.md` 或者执行 `ait config show`，看看生效的工作流 路由是什么。初始化不会启动 `ait-server`，也不会去推断编程语言或验证命令。

输出

文本输出会总结建好的或保留下来的仓库，外加一些指引。JSON 是给自动化用 的、稳定的机器可读结果。

失败与恢复

如果目录里的 AIT 状态是坏的或者只有一半，先留着这个报错，把已有的 `.ait` 结构看清楚，再考虑用修复选项。绝不能为了让初始化跑通就把权威换掉。

相关参考

- 快速上手
- `ait config`
- `ait status`

CLI · 使用与查看

ait config

查看生效的工作流路由，并应用明确的仓库配置改动。

适用人群 用户与运维

https://ait-native.dev/technical/v1.1.1/cli/config/

角色

一般由用户或 coding agent 读取。只有当仓库负责人已经选定了新行为， 才应该去改配置。

用途

`ait config` 报告当前生效的仓库路由，并应用明确的工作流预设或有界的覆盖 项。改动一旦生效，也会顺带刷新 `AGENTS.md` 里生成的 AIT 区块。

命令这一层刻意做得比落盘文件的 schema 窄。支持的根字段、嵌套字段、 归属、缺省规则和 worktree 叠加层，全都去看 附录：`.ait/config.json`。

语法

```
ait config show
ait config show --json
ait config set --workflow-mode solo_local
ait config set --sprint on
```

有远程支撑的个人工作流，对应的主预设是 `solo_remote`。先挑一个主预设， 再考虑加窄范围的覆盖项。

示例

```
ait config show --json
ait config set --workflow-mode solo_local
ait config show
```

写完之后看一眼生成的指引。打印出来的 task、Plan 和收尾命令，就是这个仓库 专属的那条路线。

输出

文本输出展示的是对做决定最有用的那些生效值。JSON 给出完整的、机器可读的 配置投影。

失败与恢复

改掉一个默认值，并不会把已有的 Task、Change、Plan 绑定或工作区搬到新作用域下。先按已记录的约定把现有沿革走完或恢复好，再去指望新默认值生效。

相关参考

- 附录：`.ait/config.json`
- `ait init`
- `ait workflow`
- 远程基础设施

CLI · 使用与查看

ait status

查看当前 Line、仓库状态、工作区状态、远程以及 worktree 健康状况。

适用人群 用户与 coding agent

https://ait-native.dev/technical/v1.1.1/cli/status/

角色

用户或 coding agent 都能放心跑，用来第一眼摸清仓库情况。

用途

`ait status` 汇总当前的 Line、头部 Snapshot、工作区状态、配置好的远程、 仓库标识，以及 worktree 的健康度。就产品沿革而言，它是只读的。

语法

```
ait status
ait status --json
```

示例

```
ait status --json
```

当 coding agent 需要判断现在待的是规范根目录还是 Task 绑定的工作区时， 用 JSON。人想快速看一眼，用那个紧凑的文本视图。

输出

常见字段包括当前 Line、默认远程、头部 Snapshot、工作区改动数量、仓库名 和 worktree 卫生状况。运行健康这类细节变了，不会因此产生新的 Snapshot。

失败与恢复

如果仓库发现失败，先确认你想要的仓库根目录在哪；只有在确实没有有效权威 时才跑 `ait init`。如果 status 报工作区是脏的，先看 `ait diff`，再去启动或 恢复 Task。

相关参考

- `ait diff`
- `ait queue`
- 故障排查

CLI · 使用与查看

ait diff

用有界的路径过滤条件，比较当前工作区与当前 Line head。

适用人群 用户与 coding agent

https://ait-native.dev/technical/v1.1.1/cli/diff/

角色

在做 Snapshot、修复或恢复的决定之前，用户或 coding agent 都能放心跑。

用途

`ait diff` 拿当前工作区跟当前 Line 的头部作比较。它不支持挑任意的 Snapshot 区间；要比不可变的 Snapshot，用 `ait snapshot diff`。

语法

```
ait diff
ait diff --stat
ait diff --name-only
ait diff [<path>...]
```

路径参数是精确的字面过滤，不是 shell 那种通配模式。`--stat` 和 `--name-only` 是两种可选的摘要形式。

示例

```
ait diff src/LoginForm.tsx
```

结果会把路径分成已修改、已缺失和未跟踪几类，遇到大文件或二进制内容会 截断，而不是无节制地全打出来。

输出

普通文本视图重点给出改动的路径和有用的内容。当自动化需要精确的分类时， `--json` 就是那道机器边界。sprint Markdown 走的是 Plan 沿革，不会作为 代码工作区的内容出现。

失败与恢复

如果某个本该在的文件不见了，先看 Task 工作区和 Line，别急着还原什么。 更不要为了凑出一个干净的 diff，就覆盖掉不相干的改动。

相关参考

- `ait status`
- `ait snapshot`
- 故障排查

CLI · 使用与查看

ait blame

在修复之前，把某个文件或行区间追溯到对应的 Snapshot 与 Plan 修订。

适用人群 用户与 coding agent

https://ait-native.dev/technical/v1.1.1/cli/blame/

角色

由 coding agent 或用户在给一个已定位的回归挑修复方案之前运行。

用途

`ait blame` 把一个路径、或者一段有界的行范围，归到对应的 Snapshot 和能 拿到的 Plan 修订沿革上。它把代码状态和记录下来的改动历史连起来，不动 工作区。

语法

```
ait blame <path>
ait blame <path> --line <number>
ait blame <path> --start <line> --end <line>
```

示例

```
ait blame src/LoginForm.tsx --line 84
```

能回答问题的最小范围就够了。还不知道相关区域在哪的时候，整文件查一遍 才有意义。

输出

结果会指出对应的 Snapshot 和有界的归属证据。如果那个源码状态连着带版本 的 Markdown 沿革，Plan 信息也会一并给出。

失败与恢复

归属查不到，那是一条证据，不是让你猜的许可。先确认路径、Line 和导入的 历史；然后把修复记成一个新的、有界的 Task。

相关参考

- 回归工作流
- `ait snapshot`
- `ait task`

CLI · 使用与查看

ait doctor

在不改动仓库权威的前提下，找到定向的诊断入口。

适用人群 用户与运维

https://ait-native.dev/technical/v1.1.1/cli/doctor/

角色

当常规的 status 证据指向存储、运行时、Plan 权威或远程依赖的问题时，由 用户或运维使用。

用途

`ait doctor` 是一组聚焦诊断的命名空间。可用的诊断主题是随版本变的，所以
先看你装的这版命令的帮助，别照抄一条不相干的维护命令。

语法

```
ait doctor --help
```

worktree 专用的诊断在 `ait worktree doctor` 下面，跟顶层的 doctor 主题是 分开的。

示例

```
ait status --json
ait doctor --help
ait worktree doctor --refresh --json
```

只挑跟 status 结论对得上的那条诊断。诊断不该变成一串碰运气的修复动作。

输出

doctor 的输出会报告它查了哪块边界、拿到什么证据，以及有界的下一步动作。 需要升级处理时，把这些留着。

失败与恢复

不要仅仅因为 doctor 报了个陈旧或不完整的状态，就去跑破坏性的清理。先把
确切的归属、标识符和路径弄清楚，再用针对那种情况给出的恢复命令。

相关参考

- `ait status`
- 故障排查

CLI · AGENT 工作流

ait plan

记录带版本的 Markdown 沿革，并在配置的范围内同步确切的 sprint 条目。

> 适用人群 coding agent 与维护者

https://ait-native.dev/technical/v1.1.1/cli/plan/

角色

通常由 coding agent 运行，作用范围严格照仓库生成的指示里打印的 那一个来。

用途

`ait plan` 记录带版本的 Markdown 沿革。在 sprint 模式下，某一个确切 的未勾选清单项会成为 Task 的绑定对象；普通文档同步一下就好，不必开 成 Task。

语法

```
ait plan sync <markdown-file-or-dir> --local
ait plan sync <markdown-file-or-dir> --remote origin
```

只用与当前生效的仓库配置相符的那个范围。首次执行的 `ait task start --from` 负责同步它所绑定的那个 sprint 条目的确切 文件。

示例

```
# Password Visibility [plan-ref: login/password-visibility/root]

- [ ] Add the accessible control and focused tests. [ref: login/password-visibility/implement]
```

```
ait plan sync docs/architecture.md --local
```

输出

同步会报出 Plan 的标识、修订、执行的动作和发布范围。Plan 的查看类 命令只读取条目和修订，不会改动它们。

失败与恢复

不要把 Plan ID 复制到 `task start` 里，也不要把 Markdown 复选框的 改动藏进代码 Snapshot。远程收尾把清单项的更新单独留出来时，直接改 那个绑定的确切条目，再在配置好的远程范围里同步这张 card。

相关参考

- 核心概念
- `ait task`
- 功能工作流

CLI · AGENT 工作流

ait task

启动绑定 Plan 的工作，查看就绪状态，并按配置好的范围完成 Task。

> 适用人群 coding agent 与维护者

https://ait-native.dev/technical/v1.1.1/cli/task/

角色

通常由 coding agent 运行。用户只说清想要的结果和约束，不用自己 去推 Task 的那套记账。

用途

`ait task` 把 sprint 意图、Change 沿革、一条功能 Line、一个受管的 工作区和最终收尾绑在一起。在 sprint 模式下，要从某一个确切的、能开 成 Task 的 Markdown 条目起步。

语法

```
ait task start --from <sprint-card>#<exact-ref> --intent <intent>
ait task audit <task-id>
ait task finish <task-or-change-id>
ait task finish <task-or-change-id> --message <message>
```

示例

```
ait task start \
  --from docs/sprints/password_visibility.md#login/password-visibility/implement \
  --intent "Add accessible password visibility control"
```

最后的输出里有 Task、Change 和确切的 `cd` 命令。所有代码工作都在 打印出来的那个工作区里做。

审计与收尾

`ait task audit` 读取 Task 的就绪状态和收尾状态。`ait task finish` 按 记录下来的工作流范围执行。Dirty 的本地工作必须带 `--message`，由该操作 创建最终 Snapshot；干净的本地工作省略这个选项并复用当前 Line head。 远程 Task 拒绝 `--message`，而且必须已有选定并完成就绪证据的 Patchset。

公开收尾入口使用 `ait task finish`，评审方使用 `ait workflow finish`。`Land` 只保留为内部原子操作及其持久记录的名称， 不是公开命令。

失败与恢复

启动只成功了一半时，留好打印出来的标识符，先查看 Task 再重试。工作 区不见了，就用 worktree 的恢复入口；除非原来的沿革已经明确处理掉， 否则不要为同一个未勾选的 sprint 条目再建第二个 Task。

相关参考

- 并行 Task 隔离
- `ait plan`
- `ait snapshot`
- `ait workflow`

CLI · AGENT 工作流

ait snapshot

记录不可变的仓库状态，并查看 Snapshot 内容或父级沿革。

适用人群 coding agent 与维护者

https://ait-native.dev/technical/v1.1.1/cli/snapshot/

角色

通常由 coding agent 在有界的实现和仓库检查都做完之后创建。查看 Snapshot 对用户来说是安全的。

用途

`ait snapshot` 记录不可变的仓库状态，并提供内容、比较、祖先和恢复 这几类查询。即使后来的 Change 或收尾目标挪了位置，Snapshot 也保留 它创建时所在的 Line 标识。

语法

```
ait snapshot create --message <message>
ait snapshot show <snapshot-id> --files
ait snapshot ancestry <snapshot-id> --ancestors
```

比较 Snapshot 要用两个不可变的标识符：

```
ait snapshot diff <old-snapshot-id> <new-snapshot-id>
```

示例

```
ait snapshot create --message "Add accessible password visibility control"
```

验证通过后，在 Task 工作区里运行这条命令。普通的 Snapshot 记录的是 代码状态；单独写的 sprint Markdown 仍然归 Plan 沿革管。

输出

create 返回新的 Snapshot 和它的父级。show 可以带上完整的文件清单；
祖先查询默认是有上限的，除非你明确要求输出全部。

失败与恢复

工作区不干净或者放错了地方，先用 `ait status` 和 `ait diff` 查一遍
再重试。回退和重放各自是独立的显式入口；替换工作区内容之前，先预览 一下它们的效果。

相关参考

- `ait diff`
- `ait task`
- 核心概念

CLI · AGENT 工作流

ait queue

查看进行中的工作、评审、远程与 worktree 健康状况的精简清单。

适用人群 用户与 coding agent

https://ait-native.dev/technical/v1.1.1/cli/queue/

角色

想摸清本地和远程正在进行的工作时，用户或 coding agent 用它都是 安全的。

用途

`ait queue` 是一份只读清单，由 Task、Change、工作区、worktree、评审和配置好的远程状态汇总而成。它自己并不维护一条独立的生命周期队列。

语法

```
ait queue summary
ait queue summary --remote origin --json
```

示例

```
ait queue summary
```

紧凑的文本输出会报出共享的 Task、就绪的候选、评审收件箱、本地草稿、 工作区改动和 worktree 的整洁情况。

输出

用 `--remote <name>` 把选定的远程权威也算进来，用 `--json` 拿到完整的机器可读投影。任何“就绪”标记都只是帮你定位的线索；真要动手改之前， 先走 `ait task audit` 或者带引导的 workflow 入口。

失败与恢复

远程读取报错时，旁边的本地清单可能照样是有效的。别把这个错误藏起来， 去查配置的远程和 server 是否健康，也别把缺失的远程条目当成已经完成 的工作。

相关参考

- `ait status`
- `ait task`
- `ait workflow`

CLI · AGENT 工作流

ait workflow

对有界改动分类，准备远程就绪证据，并查看收尾状态。

适用人群 coding agent 与运维

https://ait-native.dev/technical/v1.1.1/cli/workflow/

角色

通常由 coding agent 运行。运维人员也可以用同样的决策入口去查看 某个远程关卡，但不能绕过它。

用途

`ait workflow` 给有界的改动分类，准备远程 Patchset 的就绪状态，还能
查看或续上收尾。生效的范围和确切的命令，都来自仓库生成的那份指示。

语法

```
ait workflow guide [topic]
ait workflow reconcile --dry-run --json
ait workflow ready <change-id> --apply
ait workflow finish <change-id> --apply
```

`workflow ready` 和 `workflow finish` 是纯文本的决策入口。不要给它们 加 `--json`。

引导与对账

`workflow guide` 打印一份受支持的操作手册。`workflow reconcile` 会 跨对象盘点
Task、Change、Line、worktree、Land 和 Plan 绑定的状态。 它的默认形式和显式的 `--dry-run` 都不会改动 Plan
状态；只有对账动作 已经过评审时才用 `--apply`。

远程就绪

`workflow ready --apply` 可以创建或同步选定的 Patchset，并推进必需 的
CI、attestation、评审和策略状态。某个关卡需要单独处理时，它会停 下来，给出确切的下一步动作。

显式的远程评审者收尾入口是：

```
ait workflow finish <change-id> --apply
```

它先检查选定 Patchset 的评审和策略要求，再安全地原子 finish。配置好的
本地收尾，或就绪证据已经完整的远程收尾，也可以用：

```
ait task finish <task-or-change-id>
```

失败与恢复

不要拿本地检查去顶替必需的远程 CI，不要加上不支持的输出 flag，也不
要为了躲开阻塞就换一个权威。等对应的证据变了，再重新跑一遍决策入口， 并且保持同一个 Change ID。

相关参考

- `ait task`
- `ait queue`
- 远程基础设施

基础设施与集成

并行 Task 隔离

通过隔离的 worktree、AIT_RAM 扇出、不可变证据和原子 Land，并行跑多个 coding agent 的 Task。

适用人群 开发者、coding agent 与运维

https://ait-native.dev/technical/v1.1.1/workflows/parallel-task-isolation/

运作模型

AIT 允许并行编写、串行接纳。多个 coding agent 可以同时干活，但 它们不共用一个可变的 checkout，也不会直接往 `main` 上写。

每个受管的请求都会拿到自己的 Task、Change、功能 Line 和绑定的 worktree。agent 在这条沿革里记录不可变的 Snapshot。只有当某个候选 确切的 Patchset 攒齐了要求的证据，并且目标头又被重新核对过之后，它 才会被接纳进目标 Line。

这几层分工很重要：

- Task 绑定的 worktree 加上命令锁，避免工作区操作互相撞车。
- 不可变的 Snapshot 保证被评审过的那个修订版，不会在证据底下被换掉；
- rebase 和目标头核对，避免一个过期的候选覆盖掉更新的工作；以及
- 原子的 Land，避免共享目标 Line 变成"谁最后写谁赢"。

AIT 不会让两个 agent 编辑同一个 worktree 变得安全。高并行度来自开出 更多隔离的 Task worktree，而不是把一个目录共享得更狠。

高并发命令扇出

不同 Task 的独立 agent 可以同时下达并运行 `ait task start`、查看、 Snapshot、测试与 readiness 命令。每条命令都解析自己的 Task worktree 和 沿革，因此大型队列不需要共享一个全局可变 checkout。仓库命令锁保护 AIT 权威更新，但不会把多个执行者对同一 worktree 普通文件的无协调写入变安全。

指向同一条 Line 的 finish 刻意不做吞吐竞赛。每个候选都会重新校验当前 目标头，按需要 rebase 或停止，然后原子推进目标。这就是产品高并发 Task 支持的边界：独立编写与验证，对共享权威的接纳则串行进行。

一个 Task 独占一个可变 checkout

对象	在并行执行里的职责
Sprint 条目	给一个 Task 一个明确的目标结果和确切的 `[ref]`。
Task	拥有这个有界的工作单元和收尾状态。
Change	拥有这个 Task 可变的评审沿革。
功能 Line	让 Task 的头和目标 Line 分开。
绑定的 worktree	给一个 agent 一份物理上分开、可写的 checkout。
Snapshot	把那份 checkout 冻结成一个不可变的修订版。

对象	在并行执行里的职责
Patchset	选中由 CI、评审、attestation 和策略来评估的那个确切 Snapshot。
Land	重新校验，并原子地推进目标 Line。

`ait task start` 会创建并绑定这些工作身份。它的输出里会打出确切的 `cd` 命令。agent 必须先进到那个路径，再改代码。

```
ait task start \
  --from docs/sprints/login.md#login/password-visibility \
  --intent "Add accessible password visibility"
```

第二个 agent 起的是另一个未完成条目，拿到的是另一套 Task、Change、 Line 和 worktree。

```
ait task start \
  --from docs/sprints/login.md#login/session-regression \
  --intent "Repair session renewal regression"
```

别把同一个 sprint 条目派给两个 agent。1.1.1 会在建 Task 之前校验这个 条目是未完成且可开成 Task 的，但它并不声称某个 Plan 条目有一把完整 的、跨客户端的、服务端持有的唯一性锁。并行的 Task，应该从各不相同的确切条目引用起步。

两个 agent，一条目标 Line

假设两个 Task 都从 `main` 的 Snapshot `S0` 分出来：

```
main @ S0
|
+-- Task A -> Change A -> feature Line A -> worktree A
|              `-> Snapshot A -> Patchset A
|
`-- Task B -> Change B -> feature Line B -> worktree B
               `-> Snapshot B -> Patchset B
```

两个 agent 各自在打印出来的路径里并行实现和测试。

```
# Agent A, inside worktree A
ait snapshot create --message "Add password visibility control"
ait workflow ready <change-a> --apply

# Agent B, inside worktree B
ait snapshot create --message "Repair session renewal regression"
ait workflow ready <change-b> --apply
```

假设 Task A 先 finish，把 `main` 从 `S0` 推到了 `S1`。Task B 不会拿一个 基于 `S0` 的候选去覆盖 `S1`。

```
Task A: main S0 -> S1

Task B readiness:
  base S0 != current main S1
  |
  +-- clean rebase -> refresh Snapshot/Patchset evidence -> eligible Land
  |
  `-- conflict -> stop with main unchanged -> explicit resolution required
```

对一个干净的绑定 worktree，引导式的就绪流程可以在发布下一个候选之前， 先把它 rebase 到当前目标上。rebase 出来的候选是新内容，必须为那个 确切选定的 Patchset 带上证据；之前那次 CI 通过，不会拿来给 rebase 之后的修订版当证明。

rebase 冲突时，AIT 会记录下暂停状态，目标 Line 保持不动。查看之后， 明确选择继续还是中止：

```
ait worktree status --json
ait worktree rebase --continue
# or
ait worktree rebase --abort
```

冲突解完之后，按当前工作流输出的指引，把结果修订版记录下来并准备好。 `ait task finish` 消费的是一个已经就绪的选定 Patchset。如果在接纳之前 目标头又变了，这个操作会直接失败，而不是把中间那次结果盖过去。

为什么 AIT_RAM 是这套策略的一部分

文件系统隔离是对的，但每个 Task 都要把整个仓库还原到磁盘上，代价可能 不小。`AIT_RAM` 策略把受管的 Task worktree 放在一个验证过的内存文件 系统上，并维护一份可复用的只读 `main-seed`，让扇出更快。

AIT_RAM 是执行层和缓存层。它不是 AIT 权威，不是备份，也不是 ait-server 恢复的替代品。Task、Line、Snapshot 以及其他仓库权威，仍然留在持久的仓库状态或配置好的服务端里。放进易失内存的，只有物化出来的 checkout。

```
Persistent repository
|
+-- .ait/                          Task, Line, Snapshot authority
+-- .ait-worktree-links/task-a --+
`-- .ait-worktree-links/task-b --+---- stable local aliases
                                 |
/Volumes/AIT_RAM/                |    memory-backed execution
`-- .ait-repos/<repository-key>/ |
    |                            |
    +-- main-seed                |    read-only main Snapshot
    +-- task-a <-----------------+    writable Agent A checkout
    `-- task-b <-----------------+    writable Agent B checkout
```

Repository key 是从权威仓库路径推出来的。它把共享的内存根切分开，这样即使不同仓库的 Task worktree 名字很像，也不会撞到一起。默认的稳定别名根是仓库里的 .ait-worktree-links。

别猜路径，直接查解析出来的结果：

```
ait config show --json
ait worktree list --json
```

配置投影会报出 task_worktree.memory_root、推导出来的 ephemeral_root、别名根，以及 main-seed 的 RAM 预算（如果有）。

1.1.1 实际生效的内存默认值

这些默认值故意写死；1.1.1 不会按主机内存的百分比去推。

控制项	内置值	效果
macOS 受管 RAM 卷	8 GiB（8,589,934,592 字节，即 16,777,216 个 512 字节扇区）	内置 /Volumes/AIT_RAM 镜像的容量。
Linux 或 Windows 的 RAM 容量	AIT 不做默认分配	AIT 用的是已经挂好的 tmpfs/ramfs 或 DRIVE_RAMDISK 的容量。
仓库的 main_seed_ram_max_bytes	未设置（null）	不对 RAM 资格做 Snapshot 大小的截断。这不代表物理内存是无限的。
Task 放置的空闲空间阈值	无	1.1.1 没有任何字节或百分比阈值，会自动把新 Task 挪到 SSD。
ait doctor memory-root	只读	这个诊断从不挂载、置备、修复或创建任何根。

用 ait config show --json 和 ait doctor memory-root --json 把存下来的仓库策略跟主机当前的容量区分开。前者报的是带类型的内存根选择和仓库本地的 seed 上限；后者报的是请求的容量、总字节、可用字节，以及内置的零字节校验下限，每个值还附带来源。

内存根的校验与置备

在真正依赖 RAM 放置之前，先证明配置的这个根确实是内存支撑的，而且可写：

```
ait doctor memory-root
```

这个诊断永远是只读的。受管的 Task 放置走的是另一条路：在解析新的 worktree 时，它在 macOS 上可以置备配置的或内置的 RAM 卷。如果这次尝试拿不出一个可用的根，放置就继续走持久磁盘的回退路径。一个只是名字叫 AIT_RAM 的路径，不能当作"这块存储是内存支撑的"的证明。

平台	需要的证明	置备行为
macOS	确切的 `/Volumes/<name>` 挂载点，由一个可写的 `hdiutil` RAM 镜像支撑，且 APFS 卷身份符合预期。	新 Task 的放置可以置备带类型的或内置的卷；诊断从来不会。
Linux	一个已存在的挂载点，文件系统类型是 `tmpfs` 或 `ramfs`。	AIT 只校验，不创建系统挂载。
Windows	一个已存在的根，被报告为 `DRIVE_RAMDISK`。	AIT 只校验，不创建 RAM 磁盘。

仓库配置用的是这几个带类型的字段：

字段	用途
`task_worktree.memory_root.kind`	平台证明：`macos_ram_volume`、`linux_memory_root` 或 `windows_ramdisk`。
`task_worktree.memory_root.root`	确切的内存根绝对路径。
`task_worktree.memory_root.volume_name`	macOS RAM 卷标签；Linux 和 Windows 不用。
`task_worktree.memory_root.sector_count`	macOS `ram://` 的正整数扇区数。省略时用 `16,777,216` 个扇区。
`task_worktree.main_seed_ram_max_bytes`	可选的非负值，是默认 Line 的 Snapshot 大小上限，用来判定 RAM 资格。

已经检测到受支持的根时，`ait init` 会把它记下来。在 macOS 上，即使 仓库没存内存根，新 Task 的放置照样可以去试内置的 `/Volumes/AIT_RAM` 规格。这份 Task worktree 契约没有环境变量可以覆盖。

`main-seed` 扇出

`main-seed` 是默认 Line 头上某一个确切 Snapshot 的只读物化结果。它是 缓存，不是一个可变的共享 checkout。

一个 Task 从当前默认 Line 分出来时，引导过程走这条路径：

1. 解析出验证过的内存根，以及该 Repository 专属的 worktree 根。
2. 核对 `main-seed` 与确切的默认 Line Snapshot 是否一致。
3. seed 缺失、过期或无效时，从 Snapshot 权威刷新或重建。
4. 把 seed 拷进新的 Task 路径。
5. 只把拷出来的那棵 Task 树设为可写，并挂上它的 Task、Change 和功能 Line 元数据。
6. 把稳定的别名暴露出来，打印给 agent。

逐文件拷贝的优先顺序是：

1. macOS 的 `clonefile`；
2. Linux 的 `reflink`；
3. 写时复制用不了时，退回普通文件拷贝。

写时复制让你在 Task 启动时不必付出一份完整独立的物理拷贝。只要 agent 改了某个文件，各个 worktree 在逻辑上照样是彼此独立的。如果拷贝 seed 以安全的方式失败了，引导过程会把拷了一半的目标删掉，直接从功能 Line 的 Snapshot 还原。Task 该隔离还是隔离；变的只是那条加速路径。

让 seed 保持最新

成功 land 到默认 Line 之后，AIT 会把 `main-seed` 对齐到 land 下来的 那个 Snapshot。刷新路径会：

- 用一把 Repository 和 Line 专属的刷新锁；
- 拿候选 worktree 跟 land 下来的 Snapshot 做校验；
- 可以把已完成的绑定 worktree 提上去，也可以从 Snapshot 权威重建；

- 准备一份暂存 seed，以及一份可回退的上一版 seed 备份；
- 把装好的 seed 设为只读；以及
- 在原子目录切换之前和切换过程中，重新核对目标 Line 的世代号。

如果刷新还在准备的时候 `main` 又往前走了，AIT 会拒绝装那份过期的 seed。缓存刷新失败，不会伪造出一个成功的 seed 状态。它会被单独报出来， 免得把 land 下来的代码权威和这份用完可弃的加速缓存混为一谈。

容量与磁盘回退

RAM 上的并行度必须明确设上限。默认情况下 `task_worktree.main_seed_ram_max_bytes` 是未设置的，所以 1.1.1 不会 自动按 Snapshot 大小截断。运维可以设定允许走 main-seed 支撑的 RAM 放置的默认 Line Snapshot 大小：

```
ait config set \
  --task-worktree-main-seed-ram-max-bytes <bytes>

ait config unset task-worktree-main-seed-ram-max-bytes
```

每个新 Task 的放置判定是这样的：

```
configured seed limit exists AND Snapshot bytes > limit
  -> Repository-internal persistent disk
else if a supported RAM root can be selected and its Repository directory created
  -> RAM
else
  -> Repository-internal persistent disk
```

比较用的是严格大于。正好等于配置上限的 Snapshot，仍然有 RAM 资格。 超出上限时，AIT 会记下 `main_seed_ram_budget_exceeded`，并选用持久 Repository 根（通常是 SSD）下的 `.ait-worktree/<worktree-name>`。用内置的未设置值时，这个按大小触发 的切换是关掉的。

当所有符合条件的内存候选都不可用或者用不了时，AIT 同样会选那个持久 根。这包括：Linux 上没检测到 `tmpfs`/`ramfs`、Windows 上没有 `DRIVE_RAMDISK`、macOS 上 RAM 卷不可用或者尝试失败、配置的内存/临时 路径在它声明的根之下不被接受，或者创建 Repository 专属的 worktree 目录失败。

那个只读的内存根诊断，会报出一个内置的最小值零字节。1.1.1 不会拿这个 值当作 Task 放置的自动开关，也没有隐藏的低内存百分比阈值。需要留一份 余量的运维，得靠主机监控来落实，并在文件系统被吃光之前停止接纳新的 Task。

放置不会把一个已有的 Task 中途从 RAM 挪到磁盘。如果在选定 RAM 路径 之后拷贝 `main-seed` 失败，AIT 会先在同一个路径上直接试 Snapshot 还原；那是物化的回退，不是切到 SSD。如果选中的文件系统完不成这次还原，或者之后被写满了，命令会失败，运维得保住已记录的 Snapshot、解决容量 问题，再在一个合格的根上重建或重新开始工作。

回退改变的是性能，不是 Task 身份、Snapshot 语义或 Land 的安全性。查 Task/worktree 输出里的 `root_source`、`ephemeral_enabled`、 `target_path` 和 `fallback_reason`。`root_source` 是 `repo_internal_fallback` 且 `ephemeral_enabled: false`，就证明放在了 持久根上；别因为看到一个稳定别名，就推断它一定指向 RAM。

开始大规模扇出之前，先看看当前容量和活着的 worktree：

```
ait doctor memory-root --json
ait worktree doctor --refresh --json
ait queue summary
```

持久性与恢复边界

状态	RAM 卷丢了还在吗?	恢复来源
Task、Change 和 Line 权威	在	持久的本地权威，或者配置好的 `ait-server`。
已记录的 Snapshot 内容	在	持久的 Snapshot 存储；发布过的话，还有远程内容权威。
`main-seed`	不要求	从当前默认 Line 的 Snapshot 重建。
干净的 Task worktree	不要求	从它记录的功能 Line 头重建。
还没进 Snapshot 的脏改动	不在	没有任何自动恢复保证。

运维规矩就一条：工作必须扛得住易失执行存储丢失时，就在语义完整的检查 点上创建 Snapshot。`AIT_RAM` 加速的是 worktree 的物化；它不会让没进 Snapshot 的文件变得持久。

意外卸载或重启之后，先诊断，别急着删元数据：

```
ait doctor memory-root
ait worktree doctor --refresh --json
ait worktree recreate <worktree-name> --dry-run
ait worktree recreate <worktree-name>
```

如果注册表的恢复必须从一个已知的 Task 和 Change 绑定开始，用 `ait worktree recover-task --help` 展示的那套明确的恢复入口。恢复绝不 会仅仅因为某个老目录还在，就把已完成或已取消的权威重新打开。

清理与操作规矩

Task 成功收尾时，会在确认被接纳的 Snapshot 仍然是头之后，删掉绑定的 worktree 并归档它的 Task 功能 Line。这样既释放了易失容量，又不会删掉 Snapshot 历史。

对中断之后留下的残余，先预演一遍带归属判断的清理：

```
ait worktree cleanup-candidates --json
ait worktree cleanup --dry-run
ait worktree prune-stale --dry-run
```

同时开很多 agent 时，守住这几条：

- 每个 agent 都给一个不同的 sprint 条目和 Task。
- 只进那个 Task 打印出来的 `cd` 路径。
- 永远别把仓库的规范根目录当成共享编辑目录。
- 在依赖一个易失 worktree 之前，先记下不可变的检查点。
- 让就绪流程去刷新过期的候选；绝不要绕过基线头的冲突。
- 把 CI 当成某一个选定 Patchset 的证据，而不是某个 Task 名字的通行证。
- 通过 AIT 来 Land，这样目标头核对和原子切换才一直有效。
- 删 worktree 之前，先诊断、先预演清理。

这套东西不保证什么

worktree 隔离挡住的是文件系统层面的互相覆盖。它证明不了两处文字上不 冲突的改动，行为上也兼容。两个 agent 可以改不同的文件，却动了同一个 运行时契约。仓库自己的回归测试、评审和策略，照样少不了。

AIT 也不提供无限的 RAM 并发。仓库大小、脏增量、工具缓存、测试进程和 runner 容量，仍然决定了安全的扇出规模。`AIT_RAM` 让这份成本变得可见、 可控；它没有免掉容量规划这件事。

相关参考

- `ait task`
- `ait snapshot`

- `ait workflow`
- `ait worktree` 完整参考
- `ait doctor` 完整参考

基础设施与集成

远程基础设施

部署明确的 ait-server 与 ait-runner 边界，用于远程权威、异地恢复和仓库自有的 CI。

适用人群 运维

https://ait-native.dev/technical/v1.1.1/remote-infrastructure/

需要了再开远程权威

常规的本地工作流不需要服务端。远程基础设施带来的是三项相互独立的 能力，前提是它们真能解决具体问题：

- 通过共享、持久的工作流权威做远程管理；
- 把已记录的 Snapshot 和 Line 状态异地保存下来，用于恢复；以及
- 通过带类型的 Worker Job 执行仓库自己的 CI。

它同时也把认证、TLS、网络暴露面、数据根备份、升级和兼容性这些责任
压到运维身上。完整的数据流、保护边界和恢复步骤，读 `ait-server`：远程权威与恢复。

组件边界

- `ait-server` 拥有仓库注册表、共享的工作流状态、策略、证据和 Worker Job 队列。
- `ait-runner` 认领兼容的带类型 job，物化出确切的 Snapshot 内容， 执行仓库声明的 CI 入口，返回有界的结果。

runner 不会变成仓库权威，也不会替你挑某种语言专属的构建系统。

在本地试用 1.1.1 服务端

除非已经有一条经过评审的安全入口，否则把评估用的容器绑到 loopback。

```
docker network create ait-native
docker volume create ait-native-data
docker run --detach \
  --name ait-server \
  --network ait-native \
  --publish 127.0.0.1:8088:8088 \
  --restart unless-stopped \
  --volume ait-native-data:/var/lib/ait \
  ghcr.io/weita2026/ait-server:1.1.1

curl --fail http://127.0.0.1:8088/healthz
```

公开的容器索引覆盖 Linux `amd64` 和 `arm64`。

注册并配置仓库

本地仓库还没有数字 Repository 权威时，`remote add` 会注册一个，然后 把这个 remote
存下来。如果已经配了仓库索引，它就去核验那个确切的 权威，而不是再分配一个。

```
ait remote add origin <server-url> --default
ait remote list --json
ait config show --json
ait repo show --remote origin --json
```

别把别的仓库的索引抄过来。这个索引是远程权威路由的一部分。光注册 不会传任何
Snapshot；要做出一份可恢复的异地副本，得走配置好的远程 工作流，或者明确执行一次 `ait push`。

Runner 契约

runner 接收某个已注册仓库的版本兼容 job，执行它确切的 `ci/run.sh` 或 `ci/run.ps1`
入口。源码根目录、尝试目录、worker 身份和仓库索引 都是运维自己选的值。

尝试用的存储要隔开，投递失败要盯着，终态结果之后还得确认尝试目录 自己那份数据已经清干净。

专门的 `ait-runner`：原生 CI 执行面 一章，展开讲了 1.1.1 的确切命令、Worker Job 生命周期、带类型的契约、物化上限、租约、证据、部署和排查。

就绪与收尾

远程工作会发布一个选定的 Patchset，并在 Task 最终收尾之前记录下 CI、attestation、评审和策略这几类完成的证据。

```
ait workflow ready <change-id> --apply
ait workflow finish <change-id> --apply
```

某道关卡还没过时，就绪与 finish 命令会报出确切的下一步动作。选定的远程 Patchset 已经就绪时，也可以走配置好的 Task 级入口 `ait task finish <task-or-change-id>`。

恢复这件事是明确的

`ait-server` 只能作为送达过它的那部分状态的恢复源。脏文件、未跟踪文件以及其他没被记录下来的工作区文件，都在这条边界之外；而且一台 服务端并不提供自动故障切换。把确切的 Repository 索引留在恢复清单 里，单独保护好服务端数据根，并按 `ait-server` 那一章里带版本的 恢复流程演练一遍。

基础设施与集成

ait-server：远程权威与恢复

了解 ait-server 的远程工作流权威、Snapshot 保存、灾难恢复、runner 边界与运维职责。

适用人群 Repository 所有者与运维

https://ait-native.dev/technical/v1.1.1/ait-server/

为什么要跑 ait-server

本地的 AIT 仓库没有服务端照样能用。下面这几件事里只要有一件是必须 的，就该加上 `ait-server`：

- 远程工作流管理： 把 Repository、Plan、Task、Change、Patchset、 证据、策略、评审、Land 和 Worker Job 的状态，统一放在一个远程权威 之下。
- 异地保存与灾难恢复： 把已记录的 Snapshot 内容和 Line 头放到 另一台机器或另一个故障域，之后再用它们重建本地 AIT 权威、把源码 文件物化出来。
- 仓库自己的 CI： 接纳带类型的 job，由兼容的 `ait-runner` 通过 仓库声明的 `ci/run.sh` 或 `ci/run.ps1` 执行。

这几件事共用同一台服务端，但彼此是分开的。服务端连得上，不等于本地
每个文件都能恢复；有一份异地副本，本身也不等于有了自动高可用或故障 切换。

保护是从一个 AIT Snapshot 开始的。工作区里那些脏的、未跟踪的、
被忽略的，或者出于别的原因没被记录的文件，不会因为仓库注册过了 就自己跑到服务端上去。

服务端拥有的权威

一个服务端数据根，拥有一份全局的、只追加的 Repository 注册表。每个 注册过的 Repository 都会拿到一个数字 `repository_index`；选中它的 权威靠的是这个索引，不是显示名。Repository 的名字是可以重复的。

每一份数字 Repository 权威，都拥有它自己的远程侧：

- Snapshot 元数据、内容包、祖先关系和 Line 头；
- Plan、Task、Change、Patchset、attestation、评审、策略和 Land 记录；
- 仓库范围内的 Worker Job 和 CI 结果状态；以及
- 校验并排序这些操作所需的协议状态。

那些远程记录，服务端说了算。它不拥有开发者当前的工作目录，不替仓库 挑测试框架，也变不出从来没被记录过的文件。

在仓库的恢复清单里，把确切的 `repository_index` 留好。它是路由元
数据，不算密钥，但一丢，重建客户端时就没有安全办法认出已有的那份 权威了。索引绝对不要靠猜，也别拿另一个 Repository 的顶上。

注册并核验一个 Repository

在一个初始化过的仓库里，把服务端加成 remote：

```
ait remote add origin <server-url> --default
ait remote list --json
ait config show --json
ait repo show --remote origin --json
```

还没配 Repository 索引时，`remote add` 会注册一份新的数字权威，并把
返回的索引存下来。已经配了确切索引时，这条命令会去那台服务端核验 那份权威，对不上就直接失败。

注册不会传任何 Snapshot 或工作区内容。它只是把后续工作流、push、 pull 和恢复操作所需的地址和权威定下来。

远程工作流管理

对一个远程范围的 Task 来说，本地的 Task worktree 仍然是编写的边界。 创建完 Snapshot 之后，就绪流程会把选定的 Patchset，以及远程 Change 所要求的 Snapshot 证据发布出去：

```
ait workflow ready <change-id> --apply
ait workflow finish <change-id> --apply
```

CI、attestation、评审、策略和 Land 的状态顺序，由服务端排。最终 finish 消费的是那个已就绪的选定 Patchset，并推进远程的目标 Line。 `workflow ready` 和 `workflow finish` 都是纯文本决策界面；某道关卡还挂着时，照它们打印的确切下一步动作走。配置好的 Task 级等价入口是 `ait task finish <task-or-change-id>`。

运维可以在不改动的前提下查看已注册的权威和它的执行状态：

```
ait repo show --remote origin --json
ait repo jobs --remote origin --json
ait repo ci-capabilities --remote origin --json
```

远程工作流的发布，同时也是它所记录 Snapshot 内容的一份异地副本。 一个发布了但还没 land 的 Patchset，只是工作流证据；它不代表远程的 目标 Line 已经往前走了。

明确地保住一条 Line

如果目的是把一条已经记录下来的 Line 保住，而且跟受管的远程 Task 无关，那 `ait push` 就是那条直接的同步路径：

```
ait status --json
ait snapshot create --message "Record the recovery point"
ait push --remote origin --line main
```

push 会上传选定的 Line 头和必需的 Snapshot 祖先，然后推进那条远程 Line。它不会捕获工作区里的脏文件，不会创建 Snapshot，不会发布 Patchset，不会合并分叉的历史，也不会执行 Task finish。每条需要自己 恢复点的 Line 都得 push；注册本身不会启动什么自动同步计划。

一条 Line 最近一次成功 land 或 push 出去的远程 Line 头，就是它的可恢复点。用你能接受的最大数据丢失窗口来定发布节奏，然后去核验服务端、真的演练一次恢复，而不是把"注册成功了"当成备份证据。

保护边界

服务端能保住的是：

- 由 push 或远程工作流发布上传的 Snapshot 内容和祖先关系；
- 成功推进过的远程 Line 头；
- 已发布的远程工作流记录，以及它们接纳的证据；以及
- 由那份权威存下来的、仓库范围内的 CI 和 Worker Job 状态。

服务端恢复不了的是：

- 脏的、未跟踪的、被忽略的，或者没进过 Snapshot 的工作区内容；
- 一直留在本地、从没 push 或发布出去的 Snapshot 和 Line 移动；
- 在记录下那个恢复点之前就被删掉的文件；
- 密钥、凭据、编辑器状态、外部服务，以及任何不属于已记录 Snapshot 的依赖；或者
- 服务端本身丢了之后的服务端数据根--除非运维另外在别处保护过这份 数据根。

正因为有这条边界，异地保存才必须是有意为之的动作：创建 Snapshot、把它传出去、确认远程状态，并留好用来定位它的 Repository 索引。

本地权威完好时的还原

`.ait` 和 remote 配置都完好时，把远程 Line 拉下来，再把它的头物化 到一个干净的工作区里：

```
ait status --json
ait diff --stat
ait pull --remote origin --line main --restore
```

pull 会先导入远程的 Snapshot 祖先，再去判定本地头和远程头的关系。 远程更靠前就快进，本地历史更靠前就不动这条 Line，没让它处理的分叉 它会拒绝。`--restore` 会用选定的、拉下来的那个头替换掉工作区。工作区是脏的就会被拒，除非明确带上 `--force`；做这个选择之前，先把只 存在于本地、还活着的文件保好。

本地权威丢失或损坏时的恢复

原来的 `.ait` 权威没了或者信不过时，另开一个干净的恢复目录。损坏的那个目录留着做诊断，别删。先把可信的、存好的根路由字段作为本地权威 的一部分恢复回来：确切的 `repository_index`、`default_remote` 和 `remotes` 映射。1.1.1 没有公开的 `repository_index` 设置命令，而且 `ait remote add` 是注册操作，不能拿来安全地顶替那份恢复清单。

```
# Run after the exact saved routing fields are restored.
ait config show --json
ait repo show --remote origin --json

ait remote recover-head --remote origin
ait remote recover-head --remote origin --apply
ait pull --remote origin --line main --restore
```

第一条 `recover-head` 是预演。`--apply` 会下载服务端逻辑 `main` 头 经过校验的 Snapshot 祖先，并在一个校验过的本地 Binary 权威世代里把 它重建出来。1.1.1 的 `recover-head` 没有 `--line`，也不支持多条 Line。最后那条 pull 把 `main` 物化成源码文件。恢复不会把服务端的 远程工作流账本克隆成一份单独的本地工作流账本；远程的 Task、Change、Patchset 和 job 记录，仍然从它们的服务端权威那里查。

只要出现这几种情况就停下来：权威的响应和存下来的索引对不上、预演报出了预期之外的头、校验没过，或者工作区里还有必须保住的文件。如果确切的路由字段恢复不出来，也停。别为了绕过一个对不上，就去注册一份替代权威。完整的退役归档走的是另一套生命周期： `ait repo restore --remote origin` 会创建出一份带新 Repository 索引 的还原权威。

Runner 与 CI 的边界

`ait-runner` 是执行器，不是持久的 Repository 权威。它认领一个兼容的 带类型 Worker Job，把确切的 Snapshot 内容物化到隔离的尝试存储里，跑 仓库自己写的入口，返回有界的结果。准入、租约、job 状态和结果接纳， 都归服务端。

runner 不会替你挑某种语言专属的构建系统，不会变成 Repository 权威的第二份副本，也不会把它尝试工作区里的任意文件留下来。尝试目录要保持 隔离，投递失败的结果要盯着，终态 job 之后要把尝试目录自己的数据清掉。

1.1.1 的确切命令、job 生命周期、请求与结果契约、物化上限、租约行为 和部署检查，见 `ait-runner`：原生 CI 执行面。

保护服务端数据根

用异地服务端来保护客户端，前提是服务端权威本身还活着。容器示例把它 放在挂到 `/var/lib/ait` 的持久卷里；安装版的用户模式和服务模式，用 的是各自配置的数据根。运维必须保护的，就是这一整个根。

`ait-server` 不会去排第二个副本，不会自己复制数据根，不会选主，也不会自动故障切换。请在另一个故障域用独立的备份或存储快照系统。抓一份应用一致的副本：要么停掉写入，要么用能提供同等一致性保证的存储；别把随手拷贝的、正在被写的单个权威文件当成可恢复的东西。

服务端版本或不可变镜像摘要、数据根备份、确切的 Repository 索引清单、 远程 URL，以及单独管理的访问凭据--恢复演练要用的这些都得留着。还原 出来的副本，在对外提供服务之前先隔离验证：

```
ait-server probe --data <restored-data-root> --defer-ci-admission
```

probe 失败就意味着这个还原出来的根不能启用。`/healthz` 返回成功只能证明正在跑的这个进程是健康的；它证明不了备份够新、够全或者能恢复。

运维清单

- 默认把服务绑到 loopback，或者绑到一条评审过的私有入口上。
- 在受信任的入口和主机边界上终结 TLS，并强制做调用方校验、网络管控、 监控和容量限制。
- 把兼容的 1.1.1 客户端、服务端和 runner 制品都钉死版本。
- 每个 Repository 的数字索引，都记在客户端数据根之外的地方。
- 为每条需要保护的 Line 定好并核验 Snapshot 的发布节奏。
- 把完整的服务端数据根备份到另一个故障域。
- 拿一份隔离的副本，把 `probe`、`recover-head` 和 `pull --restore` 演练一遍。
- 盯着服务端健康度、失败的 job、runner 兼容性和存储增长。

1.1.1 的每条 HTTP 路由、请求契约、响应和错误边界，见完整的 `ait-server` REST API 参考。 远程基础设施 讲容器边界，`ait-runner`： 原生 CI 执行面 讲完整的 runner 运维契约， 排查 讲通用的证据保全 规则。

基础设施与集成

ait-runner：原生 CI 执行面

通过确切的 Worker Job 认领、Snapshot 物化、隔离的执行尝试、有界证据、租约和部署控制，运维 1.1.1 原生 runner。

> 适用人群 CI 运维、Repository 所有者与集成方

https://ait-native.dev/technical/v1.1.1/ait-runner/

ait-runner 是干什么的

`ait-runner` 是原生的执行面，跑的是 `ait-server` 接纳下来的 CI 工作。 服务端拥有 Repository 权威、Worker Job 记录、租约、状态迁移和被接纳 的结果。runner 只拥有一次有界的执行尝试：认领兼容的工作，把确切的已记录源码重建出来，调用 Repository 的 CI 入口，返回证据，然后把自己 的尝试目录删掉。

CI 必须离开开发者的 worktree 执行时、不同机器服务不同 Repository 集合时，或者运维得重放某一对已知的 Worker Job 时，runner 就派上用场。 它不是 Repository 备份，不是策略引擎，不是独立于 `ait-server` 的调度器，也不是某种语言专属的构建系统。

1.1.1 里对外可认领的 Binary Worker Job 只有两类：

固定 kind	公开名称	执行目的
`7`	`patchset.ci`	为受管的就绪判定校验选定的 Patchset Snapshot。
`11`	`repo.ci`	针对某一个确切的已记录 Snapshot 跑 Repository CI。

持久的 job 身份是 `(repository_index, worker_job_index)` 这一对。诊断 或重放一个 Binary Worker Job 时，绝对不要拿 Repository 名字来顶这 一对值。

四种使用场景

常驻的执行服务

想让一个 worker 持续轮询、维持租约、执行兼容的 job、投递终态结果时， 把 `serve` 放在服务管理器下面跑：

```
ait-runner serve \
  --server https://ait.example.internal \
  --worker-id runner-taipei-01 \
  --attempt-root /srv/ait/runner-attempts
```

`--worker-id` 是必填的运维身份，不是环境层面的设置。服务端 URL、源码根和尝试根同样是显式的命令参数，这样起出来的进程自己就把运行边界讲 清楚了。

按 Repository 划分的 worker 池

重复给 `--repository-index`，就能把一个服务实例限制在确切的一组 Repository 权威上。重复的索引会被归一：

```
ait-runner serve \
  --server https://ait.example.internal \
  --worker-id runner-linux-arm64-02 \
  --repository-index 3 \
  --repository-index 8 \
  --attempt-root /srv/ait/runner-attempts
```

不加这个过滤，兼容队列就会在所有已注册的 Repository 之间挑。过滤器是用来表达一条有意划下的机群边界的，不是拿来猜哪个 Repository 拥有 某个出错 job 的。

只跑一个确切的 Worker Job

在受控的诊断或恢复过程里，用 `run-job` 处理一对已知的 Binary 值。它 只对那一对做认领、执行、心跳和收尾：

```
ait-runner run-job \
  --server https://ait.example.internal \
  --repository-index 3 \
  --worker-job-index 42 \
  --attempt-root /srv/ait/runner-attempts
```

服务端必须声明当前的 `ait.runner.native-job.v3` 契约。这一对不可认领、 租约凭证无效，或者请求和认领到的 Repository 对不上，命令都会失败。

本地请求校验

用 `execute` 跑一个带类型的本地目录请求，不认领任何服务端 job。这对 校验 runner 边界本身很有用；它不会创建也不会更新 Worker Job：

```
{
  "contract": "ait.runner.native-job.v3",
  "label": "local-repository-ci",
  "source": {"kind": "local_directory", "path": "."},
  "command": {
    "argv": ["./ci/run"],
    "working_directory": ".",
    "environment": {}
  },
  "timeout_ms": 900000,
  "suite_id": "repository-ci"
}
```

```
ait-runner execute \
  --request request.json \
  --source-root /srv/ait/sources \
  --attempt-root /srv/ait/runner-attempts
```

`--request -` 从标准输入读同样的 JSON，这也是默认行为。服务端下发的 `remote_snapshot` 请求，需要 `run-job` 或 `serve` 用的那个服务端物化 provider；独立跑的 `execute` 走的是本地源码路径。

1.1.1 完整命令面

```
ait-runner doctor --server <SERVER>

ait-runner execute \
  [--request <REQUEST>] \
  [--source-root <SOURCE_ROOT>] \
  [--attempt-root <ATTEMPT_ROOT>]

ait-runner run-job \
  --server <SERVER> \
  --repository-index <REPOSITORY_INDEX> \
  --worker-job-index <WORKER_JOB_INDEX> \
  [--source-root <SOURCE_ROOT>] \
  [--attempt-root <ATTEMPT_ROOT>]

ait-runner serve \
  --server <SERVER> \
  --worker-id <WORKER_ID> \
  [--repository-index <REPOSITORY_INDEX>]... \
  [--once] \
  [--poll-interval-ms <MILLISECONDS>] \
  [--heartbeat-interval-ms <MILLISECONDS>] \
  [--source-root <SOURCE_ROOT>] \
  [--attempt-root <ATTEMPT_ROOT>]
```

命令	确切行为
`doctor`	读取服务端的实时健康状态并协商 runner 契约，不认领任何 job。
`execute`	解析一个有界的 v3 请求，在隔离的一次尝试里执行它的本地源码。
`run-job`	认领并完成确切的一对 Binary Worker Job；需要当前的 v3 支持。
`serve`	持续轮询；带上 `--once` 时，在一次空闲检查、一次 job 投递或一次出错之后退出。

`execute`、`run-job` 和 `serve` 的 `--source-root` 默认是 `.`。不给 `--attempt-root` 就用平台临时目录下的 `ait-runner/attempts`。`serve` 默认轮询间隔 1,000 毫秒，请求的心跳间隔 30,000 毫秒。两个间隔都不能为零；实际生效的 Binary 心跳还会被当前租约进一步收紧。

端到端的 job 生命周期

1. 一个远程工作流或 Repository CI 请求，促使 `ait-server` 为某个 Repository 提交一个带类型的 Worker Job。
2. runner 检查服务端健康度，选出当前的 Binary runner 契约。`doctor` 到这一步就停了，什么都不认领。
3. `serve` 认领下一个兼容的 job，或者 `run-job` 认领确切的 `(repository_index, worker_job_index)` 这一对。服务端返回租约凭证、 尝试次数、固定的 job kind 和带类型的运行时请求。
4. 执行之前，runner 会校验凭证、job kind、Repository 索引、请求大小、 路径、参数、环境、平台和 CI 入口。
5. 对 `remote_snapshot`，它会把确切的 Snapshot Tree、Blob 包和锁定的 外部 Repository Snapshot 下载下来、验过，放进一次新的尝试里。
6. 逻辑上的 argv 选择器 `./ci/run`，在 Unix 上解析成 `ci/run.sh`，在 Windows 上解析成 `ci/run.ps1`。runner 用直接的 argv 启动它，不是 拼一条 shell 命令。
7. 进程跑着的时候，一条单独的心跳维持着服务端租约。超时会在证据定稿 之前，把它拥有的那个进程组终止掉。
8. runner 把 stdout 和 stderr 收成有界的证据，记下退出和物化的事实， 删掉尝试目录，再校验终态结果的大小。
9. runner 带着原来那份租约凭证提交 complete 或 fail。这次状态迁移还 能不能被接受，只有服务端说了算，被接纳的终态也由它存下来。

整个尝试期间，选定的 Snapshot 始终不可变。对一个远程 Binary job， runner 永远不会去执行开发者 worktree 里的脏内容。

请求契约

当前的请求 schema 是 `ait.runner.native-job.v3`。未知字段一律拒收。 它的顶层字段是：

字段	契约
`contract`	必填，必须正好是字符串 `ait.runner.native-job.v3`。
`label`	可选的非空标签，最多 256 字节。
`source`	必填，`local_directory` 或当前的 `remote_snapshot` source 对象。
`command`	必填的直接 argv 命令对象。
`timeout_ms`	可选；默认 900,000 毫秒，取值范围必须是 1 到 86,400,000 毫秒。
`suite_id`	可选的非空套件标签，最多 256 字节。

服务端下发的 source 里，`remote_snapshot` 带着确切的 `repository_index`、`repository_name`、`snapshot_id`，以及一张从外部 Repository 名字到数字索引的映射。请求里的 Repository 索引，必须和认领 到的那一对 Worker Job 对得上。command 对象要求有 `argv`，而 `working_directory` 默认是 `.`，`environment` 默认是空映射。

argv 的第一个值永远是逻辑选择器 `./ci/run`。后面的值会直接传给平台 入口。工作目录必须是相对路径，而且不能跑出物化出来的工作区。

物化与尝试隔离

每次执行都用一个名字唯一的 `attempt-*` 目录，里面装着工作区、下载 下来的包和临时日志。`serve` 会在它配置的尝试父目录上拿一把排他锁， 而且只回收确切属于本 runner 的陈旧尝试目录。清理时会处理只读文件，并且不会跟着符号链接跑到尝试目录之外。

远程物化在这些情况下会直接失败：清单格式不对、校验和对不上、记录未知 或过多、Tree 有环或过深、路径越界、source 或入口是符号链接，以及内容 超出声明的上限。锁定的外部 Repository，是从它们各自确切的 Snapshot 物化出来的，不是从某个现成的 checkout 里拿的。

CI 子进程会收到这几个由 runner 提供的值：

名称	含义
`AIT_RUNNER_ATTEMPT_ROOT`	当前这次 runner 所属尝试的确切根目录。
`AIT_RUNNER_WORKSPACE`	物化出来的主 Repository 工作区的确切路径。
`AIT_EXTERNAL_<NAME>_REPO_ROOT`	某个归一化的、锁定的外部 Repository 物化出来的确切根目录。

这些是注入到进程边界上的值，不是启动 runner 用的设置。runner 唯一 支持的凭据环境变量，是可选的 `AIT_SERVER_TOKEN`。服务端 URL 和 worker 身份仍然走 `--server` 和 `--worker-id`；`AIT_SERVER_URL` 和 `AIT_RUNNER_WORKER_ID` 不是输入。

有界的执行与证据

边界	1.1.1 上限
带类型的请求	1 MiB。
终态结果	JSON 编码之后 64 KiB。
超时	默认 15 分钟；最长 24 小时。
命令 argv	256 个值；每个 16 KiB；总共 128 KiB。
命令环境变量	256 条；键和值加起来共 256 KiB。
stdout 与 stderr 证据	每条流取 8 KiB 尾部。

每条流的记录里包含总字节数、所有捕获字节的 SHA-256、base64 编码的 尾部、尾部字节数，还有一个截断标记。完整的临时日志会随尝试目录一起 删掉，所以终态证据是有意做成有界的，不是一份藏起来的日志归档。

远程 Snapshot 导入还会强制这些上限：清单 64 MiB、物化文件 10,000,000 个、物化字节 512 GiB、单个包下载 16 GiB，以及 8 个并发的包下载/解码 操作。这些是安全天花板，不是推荐的 job 规模；运维给真实 worker 主机 定容量预期时，应该定得低得多。

结果与诊断契约

命令执行成功会写出一个 `ait.runner.native-result.v1` 对象。它的终态 `status` 是 `succeeded`、`command_failed` 或 `timed_out`，而 `tests_status` 只有在 `succeeded` 时才会是 `pass`。结果里包含套件 记录、退出码或信号、耗时、物化的文件数和字节数、有界的 stdout 与 stderr 证据，以及清理证据。

面向服务端的命令，会把被接纳的终态迁移包在 `ait.runner.delivery.v1` 里。其他对外的输出契约有：

契约	什么时候发出
`ait.runner.doctor.v1`	`doctor` 确认健康，并报出选定的 runner 契约。
`ait.runner.service.v1`	`serve --once` 没找到兼容的 job，返回 `idle`。
`ait.runner.serve-event.v1`	常驻的 `serve` 把一次有界的 job 失败报到 stderr，然后继续轮询。
`ait.runner.error.v1`	命令在能返回正常结果之前就失败了。

`--once` 是快速失败的，出了那一个结果就返回。常驻的 `serve` 在一次 有界的单 job 失败事件之后会继续跑。轮询连不上服务端时，常驻服务模式 会做有界的重连退避，最长约 30 秒。当前的 Binary 契约一旦选定，进程 就不会悄悄降级。心跳失败会让认领到的 job 失败；runner 不会把语义 含糊的心跳重试叠在一起。

部署

1.1.1 为 macOS、Linux/glibc 和 Windows 发布原生 runner 可执行文件， arm64 和 x86_64 都有。Linux 的 OCI 索引覆盖 `amd64` 和 `arm64`：

```
ghcr.io/weita2026/ait-runner:1.1.1
```

镜像入口是 `ait-runner`，工作目录是 `/workspace`，以数字 UID/GID 65532
运行。给尝试路径挂一个属于该身份、可写的卷；跑远程 Snapshot 的 worker 并不需要一份可变的本地 Repository checkout：

```
docker volume create ait-runner-attempts
docker run --rm \
  --network ait-native \
  --volume ait-runner-attempts:/var/lib/ait-runner \
  ghcr.io/weita2026/ait-runner:1.1.1 \
  serve \
  --server http://ait-server:8088 \
  --worker-id runner-container-01 \
  --repository-index 0 \
  --attempt-root /var/lib/ait-runner \
  --once
```

`AIT_SERVER_TOKEN` 用服务或容器的 secret 来传；别把 token 嵌进镜像、 命令、Repository 或者留存的日志里。上生产之前先核验发布的索引，然后 把不可变的 1.1.1 镜像摘要钉死。

安全与运维清单

- 每个 worker 都跑在专用的操作系统身份或容器下，文件系统和网络权限 给到最小。
- 收紧服务端入站，在受信任的边界上用带认证的 TLS，并且别让 `AIT_SERVER_TOKEN` 出现在 argv 里或尝试根下的文件里。
- 尝试用的存储放在 Repository 权威之外，也放在任何源码根之外。
- 别让两个同时跑的 `serve` 进程共用一个尝试根；排他租约会拒掉它。
- 把 `ci/run.sh` 和 `ci/run.ps1` 当成 Repository 经过评审的 CI 边界 来写。别指望 runner 自己猜出包管理器或测试命令。
- 盯着服务端的 Worker Job 状态、runner 的失败事件、租约丢失、磁盘 容量、清理证据和版本契约不匹配。
- 把 `command_failed` 当作仓库 CI 的证据，把 `timed_out` 当作执行 超限，把 `ait.runner.error.v1` 当作 runner 或请求边界的失败。

先做诊断，不要认领工作：

```
ait-runner --version
ait-runner doctor --server https://ait.example.internal
ait repo jobs --remote origin --json
ait repo ci-capabilities --remote origin --json
```

然后逐项对：确切的 Repository 和 Worker Job 索引、选定的 runner 契约、
尝试次数、租约时间、终态操作、流的摘要，以及清理证据。服务端那边的 job
证据要留好；清理了一半的尝试目录别留着，更别把它当成权威源码 再用一次。

HTTP 的认领、心跳、complete、fail、Snapshot 导入和包下载这些路由，见 ait-server REST API 参考。
服务端权威和异地恢复，见 ait-server：远程权威与恢复。 完整的公开环境变量清单在 附录：环境变量。

基础设施与集成

ait-server REST API 参考

调用每一个 1.1.1 ait-server HTTP 操作，包含与实现对齐的请求字段、响应、错误、恢复流程和安全边界。

适用人群 集成方、runner 作者与运维

https://ait-native.dev/technical/v1.1.1/ait-server/rest-api/

这份参考涵盖什么

这是 1.1.1 版 `ait-server` 发布版路由器对外暴露的完整 HTTP 接口面。 它包含 51 个路由模板、71 个 HTTP 方法绑定，以及把共享路由所 接受的动作后缀展开之后的 78 个可调用操作。

这套 API 是 JSON over HTTP，既有资源读写，也有 `:runCi`、`:close`、 `:submit` 这类显式的动作端点。这些动作端点是有意为之的；整个接口面 并不是严格统一的 CRUD API。

本章记录的是 `ait`、`ait-runner` 和恢复工具所使用的服务端接口。普通 用户应优先看 `ait` 命令指南，因为客户端会强制执行时序和兼容性检查，而裸 HTTP 调用方得自己实现 这些。

安全与传输边界

把 1.1.1 的监听器当成可信网络内的服务边界。不要把裸监听器暴露到公网。把它放在环回地址或经过审查的私有网络上，并在可信入口处终结 TLS、调用 方校验、请求限额和访问策略。默认监听器是 `127.0.0.1:8088`；要换绑定地址，请使用服务端命令的显式选项。

恢复令牌和 Worker Job 租约令牌属于运维能力，不能替代入口处的认证。 不要记录或泄露它们。

下面的示例中：

```
BASE=http://127.0.0.1:8088
REPOSITORY_INDEX=17
```

通用协议规则

寻址与标识符

- `{repository_index}` 和 `{worker_job_index}` 是规范的无符号十进制 `u32` 值。`0` 有效；`01`、正负号、空格以及超过 `4294967295` 的值 都会被拒绝。
- Repository 名称属于展示和发现用的数据。名称可以重复，也不决定权威 归属；决定权威的是数字形式的 Repository 索引。
- Task、Change、Patchset、Snapshot、Land、Plan 和 revision 标识符都是 不透明的公开标识符。请保持它们原样的拼写，必要时对路径段做 URL 编码。
- 以 `_at_s` 结尾的时间字段是无符号 Unix 秒。`created_at` 这类字段 是 RFC 3339 字符串。某个事件不存在时，投影里通常是 `null`，而不是 编造一个时间戳。

请求头与请求体限额

JSON 请求用 `Content-Type: application/json`，JSON 响应用 `Accept: application/json`。标准 JSON 提取器套用框架默认的 2 MiB 请求体上限。 以下批量路由把请求上限提高到 2 GiB：

- zstd 批量 plan、commit 和 pull-manifest 请求；
- zstd Object Pack 与 Tree Pack 传输路由；以及
- Repository 恢复文件上传。

裸 pack 和恢复文件的请求体要求确切的媒体类型：

请求体	必需的 Content-Type	响应类型
Object Pack	`application/vnd.ait.remote-sync.object-pack+zstd`	同一值
Tree Pack	`application/vnd.ait.remote-sync.tree-pack+zstd`	同一值
退役或恢复文件	`application/vnd.ait.remote-authority-file.v1`	同一值

空的 Object Pack 和 Tree Pack 上传会被拒绝。超过当前生效体积限额的 请求，在进入处理器之前就会被拒绝。

成功与错误响应

多数成功操作返回 `200 OK`，并带上该路由特有的 JSON 值。Repository 注册 在追加了新权威时返回 `201 Created`，在返回已匹配的既有注册时返回 `200 OK`。启动 Repository 恢复会话返回 `201 Created`。

服务端生成的应用错误使用这个稳定的响应体：

```
{"error":"human-readable diagnostic"}
```

状态码	含义
`200`	读取、幂等重放、更新、动作或二进制下载成功。
`201`	新建了 Repository 权威或恢复会话。
`400`	路径值、请求字段或状态迁移无效，或路由自行处理的 JSON 格式有误。
`404`	未知的路由、Repository、实体、pack、文件或动作后缀。
`405`	路由存在，但这个 HTTP 方法没有绑定。
`409`	权威冲突、生命周期冲突、Line head 过期、受治理的目标 Line 移动，或退役确认不匹配。

状态码	含义
`413`	请求体超过该路由当前生效的限额。
`415`	JSON 提取器或二进制路由拒绝了这个请求的媒体类型。
`422`	JSON 语法可以接受，但没法反序列化成带类型的请求。
`500`	权威操作损坏、不兼容、I/O 失败，或其他内部原因失败。
`503`	启动尚未完成，或某个权威正忙但可重试。应用生成的 `503` 错误会带上 `Retry-After: 1`。

不要把诊断文本当成长期有效的机器契约来解析。请按 HTTP 状态码和文档
写明的响应字段做分支；文本留给运维人员诊断用。

启动行为

监听器在全部服务权威打开完成之前就已经可用。激活之前：

- `GET /healthz` 返回 `503` 和 `{"ready":false,"status":"starting"}`；
- 其他所有路径返回 `503` 和 `{"error":"ait-server is starting","ready":false,"status":"starting"}`。

就绪路由器会原子地替换掉启动期代理。激活之后， `GET /healthz` 返回 `200`，且 `ready: true`。

Repository 生命周期准入

`GET`、`HEAD` 和 `OPTIONS` 请求可以通过生命周期准入。
`/v1/native/repository-authorities/{repository_index}/...` 下的写操作 通常要求 Repository 处于 active 状态。

退役状态查询/启动、退役清除、退役中止，以及 Worker Job 的 `:claim`、 `:heartbeat`、`:complete` 和 `:fail` 这几个排空动作，在 Repository 退役期间仍然可用。针对退役中或已清除的 Repository 的其他写操作会被拒绝。这样在途工作能排空，备份或恢复流程能走完，同时又不再接纳新的 工作流工作。

完整操作索引

握手返回的紧凑 `endpoints` 数组只是发现用的提示，不是完整的路由列表。 下面这些表才是 1.1.1 发布版的完整接口面。

服务发现

方法与路径	操作
`GET /healthz`	读取就绪状态和权威后端。
`GET /v1/handshake`	读取协议、包、CI、runner 和紧凑端点能力。
`GET /v1/native/capabilities`	读取运维层面的 Repository 与 Worker Job 契约。

Repository 注册表、备份、恢复与作业

方法与路径	操作
`GET /v1/native/repository-authorities`	列出或发现 Repository 权威。
`POST /v1/native/repository-authorities`	注册一个 Repository 权威。
`GET /v1/native/repository-authorities/{repository_index}`	读取一个 Repository 权威。
`POST /v1/native/repository-authorities/{repository_index}:runCi`	入队 Repository CI。
`GET /v1/native/repository-authorities/{repository_index}/retirement`	读取退役的排空/导出状态。
`POST /v1/native/repository-authorities/{repository_index}/retirement`	开始退役。

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/retirement/files/{file_path}`	下载清单中列出的一个权威文件。
`POST /v1/native/repository-authorities/{repository_index}/retirement/purge`	在收到确切的导出确认之后清除。
`POST /v1/native/repository-authorities/{repository_index}/retirement/abort`	把退役中的 Repository 恢复成 active 状态。
`POST /v1/native/repository-restores`	开始一次恢复会话。
`PUT /v1/native/repository-restores/{restore_token}/files/{file_path}`	上传清单中列出的一个恢复文件。
`POST /v1/native/repository-restores/{restore_token}/commit`	校验并激活恢复出来的 Repository。

方法与路径	操作
`POST /v1/native/worker-jobs:claim`	认领下一个匹配的外部 runner 作业。
`GET /v1/native/repository-authorities/{repository_index}/worker-jobs`	列出 Repository 的 Worker Job。
`GET /v1/native/repository-authorities/{repository_index}/worker-jobs/{worker_job_index}`	读取一个 Worker Job。
`POST /v1/native/repository-authorities/{repository_index}/worker-jobs/{worker_job_index}:claim`	认领指定的那一个作业。
`POST /v1/native/repository-authorities/{repository_index}/worker-jobs/{worker_job_index}:heartbeat`	延长活动中的租约。
`POST /v1/native/repository-authorities/{repository_index}/worker-jobs/{worker_job_index}:complete`	提交一次成功的尝试。

方法与路径	操作
`POST /v1/native/repository-authorities/{repository_index}/worker-jobs/{worker_job_index}:fail`	记录一次可重试或终态的尝试失败。

Line、Snapshot 与远程同步

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/lines`	列出 Line。
`GET /v1/native/repository-authorities/{repository_index}/lines/{line_name}`	读取一个 Line。
`PUT /v1/native/repository-authorities/{repository_index}/lines/{line_name}`	以比较并设置的方式创建或更新一个 Line。
`POST /v1/native/repository-authorities/{repository_index}/lines/{line_name}:close`	关闭一个 Line。
`POST /v1/native/repository-authorities/{repository_index}/snapshots:exists`	批量检测 Snapshot 是否存在。
`GET /v1/native/repository-authorities/{repository_index}/snapshots/{snapshot_id}`	导出 Snapshot 元数据，可选带上内容。

方法与路径	操作
`POST /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/plan`	区分出已有和缺失的 Snapshot 与 pack。
`POST /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/commit`	提交已上传的 pack 元数据、定位符、Snapshot，以及可选的 Line 更新。
`GET /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/import-manifests/{snapshot_id}`	取得一个 Snapshot 的导入闭包。
`POST /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/pull-manifests`	取得一个有界祖先范围的导入闭包。
`GET /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/object-packs/{pack_id}`	下载一个 Object Pack。
`PUT /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/object-packs/{pack_id}`	上传一个 Object Pack。

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/tree-packs/{pack_id}`	下载一个 Tree Pack。
`PUT /v1/native/repository-authorities/{repository_index}/remote-sync/zstd-bulk/tree-packs/{pack_id}`	上传一个 Tree Pack。

Task 与队列读模型

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/tasks`	列出 Task，最新的在前。
`POST /v1/native/repository-authorities/{repository_index}/tasks`	创建一个 Task，可选择绑定 Plan。
`POST /v1/native/repository-authorities/{repository_index}/task-start`	原子地创建/修订/选定 Plan 条目、Task 和第一个 Change。
`GET /v1/native/repository-authorities/{repository_index}/tasks/{task_ref}`	读取一个 Task。
`POST /v1/native/repository-authorities/{repository_index}/tasks/{task_ref}:close`	完成或放弃一个 Task。
`GET /v1/native/repository-authorities/{repository_index}/read/tasks/{task_id}/audit`	读取 Task 收尾证据。

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/read/queue-summary`	读取队列汇总投影。
`GET /v1/native/repository-authorities/{repository_index}/read/task-queue`	读取 Task 队列投影。
`GET /v1/native/repository-authorities/{repository_index}/read/reviewer-inbox`	读取分配给该 Repository 的评审工作。

Change、Patchset、评审、策略、CI 与 Land

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/changes`	列出 Change，最新的在前。
`POST /v1/native/repository-authorities/{repository_index}/changes`	在某个 Task 下创建一个 Change。
`GET /v1/native/repository-authorities/{repository_index}/changes/{change_ref}`	读取一个 Change。
`POST /v1/native/repository-authorities/{repository_index}/changes/{change_ref}:close`	归档、放弃或替代一个 Change。
`POST /v1/native/repository-authorities/{repository_index}/changes/{change_ref}:selectPatchset`	选定一个 Patchset。
`POST /v1/native/repository-authorities/{repository_index}/changes/{change_ref}:requestReview`	向一个评审组发起评审。

方法与路径	操作
`POST /v1/native/repository-authorities/{repository_index}/changes/{change_ref}:submit`	为一个 Change 提交直接 Land。
`GET /v1/native/repository-authorities/{repository_index}/changes/{change_ref}/patchsets`	按序号顺序列出 Patchset。
`POST /v1/native/repository-authorities/{repository_index}/changes/{change_ref}/patchsets`	发布一个 Patchset。
`GET /v1/native/repository-authorities/{repository_index}/changes/{change_ref}/reviews`	读取评审记录行和汇总计数。
`POST /v1/native/repository-authorities/{repository_index}/changes/{change_ref}/reviews`	记录一次评审动作。
`GET /v1/native/repository-authorities/{repository_index}/patchsets/{patchset_id}`	读取一个 Patchset 及其 CI 投影。

方法与路径	操作
`POST /v1/native/repository-authorities/{repository_index}/patchsets/{patchset_id}:runCi`	启动或复用 Patchset CI，必要时入队一个 Worker Job。
`POST /v1/native/repository-authorities/{repository_index}/patchsets/{patchset_id}:evaluatePolicy`	评估当前的 attestation、CI 和评审关卡。
`GET /v1/native/repository-authorities/{repository_index}/patchsets/{patchset_id}/attestation`	读取最新的 attestation。
`PUT /v1/native/repository-authorities/{repository_index}/patchsets/{patchset_id}/attestation`	追加一条新的 attestation。
`GET /v1/native/repository-authorities/{repository_index}/patchsets/{patchset_id}/policy`	读取最新的策略决定，或待评估状态的投影。
`GET /v1/native/repository-authorities/{repository_index}/read/patchsets/{patchset_id}/ci-status`	读取紧凑的 CI 就绪度以及近期作业。

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/lands/{submission_id}`	读取一次 Land 尝试。
`POST /v1/native/repository-authorities/{repository_index}/task-land`	原子地解析并 Land 一个 Task 或指定的 Change。
`POST /v1/native/repository-authorities/{repository_index}/history-promotion:prepare`	为受治理的远程 Land 准备已记录的本地 land 历史。

Sprint 与 Plan 权威

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/sprints`	列出 Plan，可按 artifact 路径过滤。
`POST /v1/native/repository-authorities/{repository_index}/sprints`	创建一个 Plan 和它的第一个 revision。
`GET /v1/native/repository-authorities/{repository_index}/sprints/{plan_id}`	读取 Plan 详情和 head revision。
`PATCH /v1/native/repository-authorities/{repository_index}/sprints/{plan_id}`	修改 Plan 状态。
`GET /v1/native/repository-authorities/{repository_index}/sprints/{plan_id}/revisions`	列出 Plan revision，最新的在前。
`POST /v1/native/repository-authorities/{repository_index}/sprints/{plan_id}/revisions`	追加一个 revision，可选带 head 比较并设置。

方法与路径	操作
`GET /v1/native/repository-authorities/{repository_index}/sprints/{plan_id}/revisions/{plan_revision_id}`	读取一个完整的 revision。
`GET /v1/native/repository-authorities/{repository_index}/sprints/{plan_id}/revisions/{plan_revision_id}/artifacts`	读取同一个完整 revision，并带上它打包后的 artifact 投影。
`PUT /v1/native/repository-authorities/{repository_index}/sprints/{plan_id}/revisions/{plan_revision_id}/artifacts`	预留路由；1.1.1 拒绝写入支撑性 artifact。
`POST /v1/native/repository-authorities/{repository_index}/sprint-plan-ids/by-contains`	查找匹配全部归一化词条的活动 Plan ID。
`POST /v1/native/repository-authorities/{repository_index}/sprint-task-linkage/resolve`	归一化并校验 Task 到 Plan 的关联。
`GET /v1/native/repository-authorities/{repository_index}/read/plans/candidate-inputs`	读取匹配的 Plan 及其关联的 Task 候选。

发现类端点

健康检查

```
curl --fail-with-body "$BASE/healthz"
```

就绪时的响应：

```
{
  "ready": true,
  "authority_backend": "binary_v0",
  "repository_identity": "repository_index",
  "operational_capabilities": {
    "contract": "ait.server.operational-capabilities.v1",
    "repository_identity": "binary-repository-index.v0",
    "worker_job_identity": "binary-worker-job-key.v0",
    "runner_contracts": ["ait.runner.native-job.v3"]
  }
}
```

把 HTTP `200` 加上 `ready: true` 当作进程已就绪。它并不能证明备份是 最新的，或者是可恢复的。

握手

```
curl --fail-with-body "$BASE/v1/handshake"
```

响应字段如下：

字段	含义
`ready`	就绪路由器的状态。
`contract_version`	agent/服务端的协议兼容版本。
`package_version`	正在运行的 `ait-server` 包版本。
`authority_backend`	1.1.1 中是 `binary_v0`。
`repository_identity`	`repository_index`。
`ci_capabilities`	远程同步特性和原生 runner 契约。

字段	含义
`operational_capabilities`	Repository 与 Worker Job 的身份契约。
`supported_async_job_types`	服务端已知的十种持久作业类型名。
`endpoints`	紧凑的发现子集；不是完整的操作索引。

原生 runner 能力声明的是 `ait.runner.native-job.v3`、结果契约 `ait.runner.native-result.v1`、队列契约 `ait.server.worker-job.service.v1`、远程 Snapshot 输入、直接参数 执行，以及平台对应的 `ci/run.sh` 或 `ci/run.ps1` 入口。

运维能力

`GET /v1/native/capabilities` 返回 `operational_capabilities` 对象， 不带更大的握手载荷。在动手写直连的 Worker Job 客户端之前先看它。

Repository 注册与 Repository CI

注册一个权威

```
curl --fail-with-body \
  -H 'Content-Type: application/json' \
  -d '{"repository_name":"ait-native-site","namespace":"w","policy_flags":0}' \
  "$BASE/v1/native/repository-authorities"
```

请求字段	是否必需	规则
`repository_name`	是	非空的 UTF-8 展示名；允许重复。
`namespace`	是	零个、一个或两个 ASCII 字母、数字、_ 或 -。服务端按确切的零填充字节存储。
`policy_flags`	是	无符号 u8。

未知字段会被拒绝。一个非空命名空间如果已被某个存活的 Repository 占用，只有在名称和策略标志也一致时才会走重放；否则注册就是冲突。

```
{
  "contract": "ait.server.repository-registration.v1",
  "created": true,
  "repository": {
    "repository_index": 17,
    "repository_name": "ait-native-site",
    "lifecycle_kind": 1,
    "namespace": "w",
    "policy_flags": 0,
    "created_at_s": 1786800000,
    "updated_at_s": 1786800000,
    "tombstoned": false
  }
}
```

把响应里的 `repository_index` 持久化下来。不要在多个权威共用同一个 展示名时，靠名称重新发现再猜一个。

发现并读取权威

```
curl --get --fail-with-body \
  --data-urlencode 'repository_name=ait-native-site' \
  "$BASE/v1/native/repository-authorities"

curl --fail-with-body \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX"
```

列表响应包含 `contract`、`repositories`、`count` 和一个 `discovery` 对象。`discovery.routing_authority` 是 `false`：按名称过滤是用来盘点 清单的，不是用来选路由的。单项响应把一个 `repository` 包在 `ait.server.repository-authority.v1` 里。

生命周期取值为 1 active、2 retiring、3 purged。已 purged 的条目 不出现在列表发现里。

入队 Repository CI

这个路由在 Repository 段上使用动作后缀：

```
curl --fail-with-body \
  -H 'Content-Type: application/json' \
  -d '{"snapshot_id":"SNP-EXAMPLE"}' \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX:runCi"
```

`snapshot_id` 是可选的。不填时，服务端选用逻辑 `main` 的当前 head。 两种途径都解析不到 Snapshot 时，请求失败。响应包含 `repository_index`、`snapshot_id`、物理的 `snapshot_index`、`queued`、 Worker Job 投影，以及 `delivery: "binary_worker_job"`。

Repository 退役、导出与恢复

退役是 API 层面针对一个完整数字 Repository 权威的备份/导出流程。它被 刻意拆成多段：先停掉新的写操作，让 Worker Job 排空，校验每个导出文件， 最后清除还要求一次确切的确认。

1. 开始并轮询退役

下面这些 POST 操作没有请求体：

```
curl --fail-with-body -X POST \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/retirement"

curl --fail-with-body \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/retirement"
```

响应内容为：

```
{
  "contract": "ait.server.repository-retirement.v1",
  "repository": {"repository_index":17,"lifecycle_kind":2},
  "drain": {"queued_worker_jobs":0,"running_worker_jobs":0},
  "ready_for_export": true,
  "manifest": {
    "schema": "ait.remote-export.v1",
    "state": "complete",
    "repo_name": "ait-native-site",
    "namespace": "w",
    "exported_at_s": 1786800100,
    "files": [
      {"path":"...","size":1234,"sha256":"..."}
    ]
  }
}
```

在排队中和运行中的 Worker Job 双双归零之前，`manifest` 一直是 `null`。 只有 `ready_for_export: true` 才是取用所列文件的许可。

2. 下载并校验每个文件

对 `manifest.files[]` 的每一条，把它的相对 `path` 做 URL 编码后取用：

```
curl --fail-with-body \
  -o authority-file.bin \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/retirement/files/<encoded-path>"
```

响应是裸的 `application/vnd.ait.remote-authority-file.v1`。请对照清单校验 `size` 和小写十六进制的 `sha256`。当前清单里没有的路径返回 `404`。

3. 只在导出已持久化之后才清除

把 整个未经改动的 manifest 对象 作为清除请求体 POST 过去：

```
curl --fail-with-body \
  -H 'Content-Type: application/json' \
  --data-binary @remote-export-manifest.json \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/retirement/purge"
```

服务端会重新计算它当前的完整导出，并要求完全相等。不完整的确认、变过的文件清单，或者作业尚未排空，都会被拒绝。成功时返回 `contract`、 `purged`、`already_purged` 和终态的 Repository 投影。重放同一份有效的 确认是幂等的。

中止退役

在清除之前，这个无请求体的操作会把 Repository 恢复成 active 状态：

```
curl --fail-with-body -X POST \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/retirement/abort"
```

响应会报告 `aborted: true` 和 `already_aborted`。已 purged 的权威无法 重新激活。

恢复一个已导出的权威

用校验过的清单，加上新注册表条目要用的 u8 策略标志，开一个会话：

```
curl --fail-with-body \
  -H 'Content-Type: application/json' \
  -d '{"manifest":<complete-manifest>,"policy_flags":0}' \
  "$BASE/v1/native/repository-restores"
```

带类型的请求只接受 `manifest` 和 `policy_flags` 两个字段。清单必须 满足：

字段	规则
`schema`	`ait.remote-export.v1`
`state`	`complete`
`repo_name`	非空的源 Repository 名称。
`namespace`	导出时的确切命名空间。它不能已被某个存活的 Repository 占用。
`exported_at_s`	以 Unix 秒表示的导出时间。
`files`	由 `{path, size, sha256}` 条目组成的确切数组。

`201` 响应返回一个随机的 32 个字符的十六进制 `restore_token`，以及 `state: "uploading"`。未提交的会话最多接纳 64 个。

按每个文件确切的相对路径和媒体类型逐个上传：

```
curl --fail-with-body -X PUT \
  -H 'Content-Type: application/vnd.ait.remote-authority-file.v1' \
  --data-binary @authority-file.bin \
  "$BASE/v1/native/repository-restores/$RESTORE_TOKEN/files/<encoded-path>"
```

每次上传都会对照清单里的大小和摘要做检查。然后用无请求体的方式提交：

```
curl --fail-with-body -X POST \
  "$BASE/v1/native/repository-restores/$RESTORE_TOKEN/commit"
```

提交会校验整个暂存归档，追加一条新的数字 Repository 条目，把恢复出来 的权威归一化到那个新索引上，并返回 `ait.server.repository-restore.v1`。在同一个存活会话上重放提交，会返回已记录的响应。恢复不保留旧的数字索引；响应里的新索引才是权威。

Worker Job 相关 API

服务端认识十种持久作业类型，但 1.1.1 的外部 runner 只能认领 `7`（`patchset.ci`）和 `11`（`repo.ci`）。其余类型属于服务端内部操作， 外部认领端点会拒绝它们。

Worker Job 状态是 `1` queued、`2` running、`3` succeeded 和 `4` failed。`(repository_index, worker_job_index)` 这一对就是公开身份。

列出与查看

```
curl --get --fail-with-body \
  --data 'state_kind=1' \
  --data 'limit=50' \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/worker-jobs"
```

`state_kind` 可选，取值必须在 `1..4`。`limit` 默认 `50`，必须在 `1` 到 `1000` 之间。结果按最新在前排列。每个作业投影包含标识符、kind/type、状态、重试状态、结果、尝试次数上限、错误类型、Patchset 或 Snapshot 引用、可用性、加锁与更新时间，以及墓碑状态。Patchset CI 作业 还会带上当前的紧凑 CI 投影。

读取指定的某个作业：

```
curl --fail-with-body \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/worker-jobs/42"
```

认领下一个或认领指定的

全局认领可以带一个可选的 Repository 过滤条件：

```
{
  "accepted_job_kinds": [7, 11],
  "repository_indexes": [17],
  "accepted_runtime_contracts": ["ait.runner.native-job.v3"]
}
```

把它发到 `POST /v1/native/worker-jobs:claim`。两个数组都不能有重复项。 `repository_indexes` 为空数组表示所有正在服务的 Repository。没有任何 匹配时，响应是 `200` 且 `claimed_job: null`。

要认领一个已经选好的作业，把下面这个确切的请求体 POST 到 `.../worker-jobs/{worker_job_index}:claim`：

```
{"accepted_runtime_contracts":["ait.runner.native-job.v3"]}
```

认领成功会返回 `attempt_count`、十六进制的 `lease_token`、 `lease_expires_at_s`、作业身份和 `runtime_request`。这个运行时请求会 选定一个远程 Snapshot，声明直接执行的 `ci/run` 参数向量，并设置超时。请严格按这个请求执行；不要拿当前工作区顶替。

心跳、完成与失败

心跳要求带上当前的尝试次数和租约，并且禁止带 `detail`：

```
{"attempt_count":1,"lease_token":"<hex-token>"}
```

在租约到期之前，把它 POST 到 `.../{worker_job_index}:heartbeat`。

完成必须带 `detail`。对 `repo.ci` 来说，固定的作业类型就足以完成持久 化。对 `patchset.ci`，`detail` 必须标明作业类型 `7`，并携带一份 `ait.runner.native-result.v1` 结果：

```
{
  "attempt_count": 1,
  "lease_token": "<hex-token>",
  "detail": {
    "job_kind": 7,
    "result": {
      "contract": "ait.runner.native-result.v1",
      "status": "succeeded",
      "tests_status": "pass",
      "suite_result_count": 1,
      "suite_results": [
        {"suite_id":"cargo_test","status":"pass","blocking":true}
      ]
    }
  }
}
```

终态结果的 `status` 是 `succeeded`、`command_failed` 或 `timed_out`； `tests_status` 是 `pass` 或 `fail`；每个 suite 的状态是 `pass` 或 `fail`。`suite_result_count` 必须等于数组长度。服务端会据此推导出紧凑 的 CI 证据，并原子地挂到选定的 Patchset 上。

失败必须带上错误信息和重试判断：

```
{
  "attempt_count": 1,
  "lease_token": "<hex-token>",
  "detail": {
    "retryable": true,
    "error": "runner lost its isolated attempt process"
  }
}
```

`error` 的取值上限是 4,096 个字符。只有 `retryable` 为 true 且还有剩余 尝试次数时，服务端才会重新入队。完成、失败和心跳都会拒绝过期的尝试次数或租约令牌。

Line 与 Snapshot

读取与更新 Line

`GET .../lines` 返回全部 Line 投影。`GET .../lines/{line_name}` 返回 一个投影，包含它的名称、状态和当前的 `head_snapshot_id`。

用下面这个请求体创建 Line，或对它做比较并设置：

```
{
  "head_snapshot_id": "SNP-NEW",
  "expected_head_snapshot_id": "SNP-OLD"
}
```

两个字段都是可选且可为 null 的。`expected_head_snapshot_id` 是比较并设置的守卫。直接的远程同步可以给一条空的默认 Line 做引导、重放它当前 的 head，或者移动另一条 Line。但它不能把已初始化的默认 `main` 移到 另一个 Snapshot；那会返回 `409` 和 `GOVERNED_TARGET_LINE_REQUIRES_LAND`，必须走 Land 完成。

关闭一条 Line，POST 到 `.../lines/{line_name}:close`：

```
{"status":"archived"}
```

省略时 `status` 默认为 `archived`。

检测 Snapshot 是否存在

```
curl --fail-with-body \
  -H 'Content-Type: application/json' \
  -d '{"snapshot_ids":["SNP-A","SNP-B"]}' \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/snapshots:exists"
```

响应会对给定的 Snapshot ID 做分类，而不传输它们的内容。

导出单个 Snapshot

```
curl --get --fail-with-body \
  --data 'include_content=true' \
  --data-urlencode 'path=src/main.rs' \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/snapshots/SNP-EXAMPLE"
```

`include_content` 默认为 `true`。`path` 可选，用来收窄导出的文件投影。 这是 JSON 形式的 Snapshot 导出，不是裸 pack 下载。

zstd 批量远程同步

安全的传输顺序是：先请求一个 plan，上传缺失的裸 pack，提交它们的清单 行和 Snapshot，然后按受治理 Line 的规则读取或更新 Line。拉取则是反向 过程：先请求一份清单，再下载它列出的 pack。

存在性 plan

```
{
  "snapshot_ids": ["SNP-HEAD"],
  "object_packs": ["OPK-A", {"pack_id":"OPK-B"}],
  "tree_packs": ["TPK-A"]
}
```

POST 到 `.../remote-sync/zstd-bulk/plan`。这三个数组都可以为空。pack 条目可以是 ID，也可以是带 `pack_id` 的清单对象。响应在下面这些字段里 保持请求顺序：

- `checked_snapshot_ids`；
- `present_snapshot_ids` 和 `missing_snapshot_ids`；
- `present_object_pack_ids` 和 `missing_object_pack_ids`；以及
- `present_tree_pack_ids` 和 `missing_tree_pack_ids`。

裸 pack 上传与下载

```
curl --fail-with-body -X PUT \
  -H 'Content-Type: application/vnd.ait.remote-sync.object-pack+zstd' \
  --data-binary @object-pack.zst \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/remote-sync/zstd-bulk/object-packs/O
    PK-A"
```

Tree Pack 传输走对应的 `tree-packs` 路径和媒体类型。路径里的 `pack_id` 必须是合法的单个路径段，而且要和上传的 pack 元数据一致。上传响应会 描述被接受的 pack；GET 返回确切的原始字节。

Commit

```
{
  "contract": "ait.remote_sync.zstd_bulk.commit.v1",
  "object_packs": [],
  "tree_packs": [],
  "blob_locators": [],
  "tree_locators": [],
  "snapshots": [],
  "line_update": null
}
```

每个清单字段都接受数组或 `null`；缺失的字段按空处理。pack、定位符和 Snapshot 对象就是 import 或 pull 清单原样给出的那些行。不要自行合成， 也不要丢掉它们的完整性字段。`line_update` 存在时，包含 `line_name`、 可为 null 的 `head_snapshot_id` 和可为 null 的 `expected_head_snapshot_id`。

响应会报告被更新插入和被跳过的 Object Pack、Tree Pack、Blob、Tree 和 Snapshot，另外还有 `remote_line`、`line_head_updated_after_ingest` 和 `raw_binary_upload: true`。已有的
不可变内容只有在与权威一致时才会被跳过；不一致就是错误，而不是覆盖。

import 与 pull 清单

`GET .../import-manifests/{snapshot_id}` 返回一个完整的闭包：

```
{
  "contract": "ait.remote_sync.zstd_bulk.import_manifest.v1",
  "repo_name": "ait-native-site",
  "snapshot_id": "SNP-HEAD",
  "snapshots": [],
  "object_packs": [],
  "tree_packs": [],
  "blob_locators": [],
  "tree_locators": [],
  "line_update": null
}
```

要取有界的祖先范围，POST：

```
{
  "contract": "ait.remote_sync.zstd_bulk.pull_manifest.request.v1",
  "head_snapshot_id": "SNP-HEAD",
  "have_snapshot_ids": ["SNP-BOUNDARY"]
}
```

`have_snapshot_ids` 不能有重复项，条目上限是 100,000。响应契约是 `ait.remote_sync.zstd_bulk.pull_manifest.v1`，并额外带上 `head_snapshot_id` 和 `boundary_snapshot_ids`。Snapshot 行按从所需的 祖先边界指向请求 head 的顺序排列；pack 清单则做过去重并按依赖排序。 下载返回的 pack ID，并把返回的那些行数组原封不动传给目标端的 commit。

Task 生命周期

创建与读取 Task

创建一个不绑定 Plan 的 Task：

```
{"title":"Correct release readback","intent":"Verify the published artifact"}
```

可选的 `task_id` 只有在等于服务端推导出的下一个 ID 时才被接受。可选的 Plan 关联要三个字段一起用：

```
{
  "title": "Correct release readback",
  "intent": "Verify the published artifact",
  "plan_id": "PR-12",
  "origin_plan_revision_id": "plan-revision:19",
  "plan_item_ref": "site/release/readback"
}
```

如果给了 `plan_id` 而没给 revision，路由会解析出当前的 head。如果给了 revision 而没给 Plan ID，它会在选定的 Repository 内搜索。条目必须存在 于那个 revision 中。`repo_id`、`repository_index` 这类由路由自己掌握 的字段会被拒绝。

Task 响应包含 `task_id`、Repository 身份、序号、标题、意图、Plan 关联 和漂移投影、状态，以及时间戳。状态是 `active`、`completed` 或 `abandoned`。

用 `GET .../tasks` 列出，用 `GET .../tasks/{task_ref}` 读取，关闭则 POST 到 `.../tasks/{task_ref}:close`：

```
{"status":"completed"}
```

关闭时接受的状态是 `completed`、`abandoned`、`canceled` 或 `cancelled`；后三个都会投影成 `abandoned`。

原子式 Task 启动

`POST .../task-start` 会创建或选定确切的 Plan revision 和条目，然后在 一次操作里追加 Task 和第一个 Change：

```
{
  "contract": "task-start-atomic/v1",
  "idempotency_key": "client-unique-key",
  "plan_item_ref": "site/release/readback",
  "plan": {
    "action": "existing",
    "plan_id": "PR-12",
    "plan_revision_id": "plan-revision:19"
  },
  "task": {
    "title": "Correct release readback",
    "intent": "Verify the published artifact"
  },
  "change": {
    "title": "Implement verified readback",
    "base_line": "main"
  }
}
```

`idempotency_key` 必填，上限 256 字节。`plan.action` 取值为：

- `existing`：需要 `plan_id` 和当前 head 的 `plan_revision_id`；

- `create`：需要在 `plan.payload` 下给出一份 Plan revision 载荷；或者
- `revise`：需要 `plan_id`、`plan.payload`，以及当前的 `expected_head_revision_id`，除非那个确切的 revision 已经存在。

Task 的 Plan 关联和 Change 的 `task_id` 由服务端推导；调用方不得自行提供这些推导字段。用同样的绑定和载荷重放，会返回已记录的原子结果。

Task 审计与队列

`GET .../read/tasks/{task_id}/audit?target_line=main` 返回该 Task、 它的全部 Change、目标 Line 的 head、未关闭和已 land 的 Change 计数， 以及一个结论：`task_completed`、`task_abandoned`、`ready_to_close`、`no_changes` 或 `in_progress`。

`GET .../read/queue-summary` 和 `GET .../read/task-queue` 接受可选的 `status=<value>`。前者返回工作流的聚合计数，后者返回 Task 队列的行。 `GET .../read/reviewer-inbox` 返回该 Repository 当前的评审工作。这些 都是派生出来的读模型：写入权威请用实体路由。

Change、Patchset、评审、策略、CI 与 Land

创建与关闭 Change

```
{
  "task_id": "T-0001",
  "title": "Implement verified readback",
  "base_line": "main",
  "fork_snapshot_id": "SNP-BASE"
}
```

`task_id`、`title` 和 `base_line` 是必填的。`fork_snapshot_id` 可选， 但如果给了，就必须等于创建时基础 Line 的 head。可选的 `change_id` 必须等于服务端推导出的完整身份或短身份。

Change 响应包含 `change_id`、完整的 `change_ref`、Task 和 Repository、 标题、基础 Line、fork Snapshot、状态、当前和选定的 Patchset、 land 目标，以及时间戳。状态可以是 `draft`、`active`、`review`、`landed`、`archived`、`superseded` 或 `abandoned`。

用下列状态之一关闭：

```
{"status":"archived"}
```

接受的取值是 `archived`、`superseded`、`abandoned`、`canceled` 和 `cancelled`。`landed` 会被拒绝，因为那个状态由成功的 Land 推导得出。

发布与选定 Patchset

POST 到 `.../changes/{change_ref}/patchsets`：

```
{
  "base_snapshot_id": "SNP-BASE",
  "revision_snapshot_id": "SNP-REVISION",
  "summary": "Add verified readback",
  "author_mode": "ai_with_human_review"
}
```

必填的作者模式取值是 `human_only`、`human_with_ai_assist`、 `ai_with_human_review` 或 `ai_only_experimental`。可选的 `patchset_id` 和 `patchset_number` 只有在与推导出的下一个身份一致时才被接受。用同样的 Change/base/revision 三元组重放，会返回已有的那个 Patchset。

Patchset 响应包含身份、Snapshot、摘要、作者与发布状态、撤回标志、改动文件的统计和路径、评估状态、创建时间、CI 运行序号，以及紧凑的 overall/tests/lint 证据。

选定 Patchset，POST 到 `.../changes/{change_ref}:selectPatchset`：

```
{"patchset_id":"T-0001/C-01/P-02"}
```

这个 Patchset 必须属于该 Change，而且不能已被撤回或作废。

发起与记录评审

一次只发起一个评审组：

```
{
  "patchset_id": "T-0001/C-01/P-02",
  "reviewer_groups": ["maintainers"],
  "note": "Please check release semantics"
}
```

把它发到 `.../changes/{change_ref}:requestReview`。可选的 `review_id` 只有在与推导出的身份一致时才被接受。

记录一次评审，POST 到 `.../changes/{change_ref}/reviews`：

```
{
  "patchset_id": "T-0001/C-01/P-02",
  "reviewer": "reviewer@example.invalid",
  "action": "approve",
  "comment": "Verified",
  "blocking": false
}
```

动作取值是 `request`、`comment`、`task_comment`、`code_review_summary`、`approve`、`task_approve`、`request_changes`、`task_request_changes`、 `defer`、`task_defer` 和 `dismiss`。`comment` 默认为空，`blocking` 默认为 false。这个 Patchset 必须是其 Change 当前选定的 Patchset。

评审列表响应包含 `reviews`，以及批准、阻塞、评论和总计这几个汇总计数。

Attestation 与策略

用 `PUT .../patchsets/{patchset_id}/attestation` 追加一条 attestation：

```
{
  "author_mode": "ai_with_human_review",
  "verification_state": "pass",
  "revoked": false,
  "require_tests_pass": true,
  "require_human_review": true,
  "require_lint_pass": true
}
```

`verification_state` 取值为 `unknown`、`pending`、`pass` 或 `fail`。 默认值是 `unknown`、`revoked: false`、`require_tests_pass: true`，人工 评审和 lint 这两项要求默认为 false。如果给了 `author_mode`，它必须与 Patchset 一致。调用方提供的 `detail.patchset_ci` 会被拒绝，因为 CI 的 完成归 Patchset 和 Worker Job 权威管。

`GET .../attestation` 返回最新的 attestation，没有时返回 `404`。它包含 各项要求标志、这条记录是否有 CI 支撑、作者模式，以及当前的 tests/lint 汇总。

`POST .../patchsets/{patchset_id}:evaluatePolicy` 接受一个空 JSON 对象， 并基于最新的 attestation、紧凑 CI 和实时的评审批准追加一条决定。 `GET .../policy` 返回那条决定和它的各项检查。首次评估之前，它返回：

```
{"patchset_id":"...","decision":"pending","checks":[]}
```

运行与查看 Patchset CI

用下面这个请求体启动 CI：

```
{"trigger":"manual_rerun"}
```

POST 到 `.../patchsets/{patchset_id}:runCi`。不给 trigger 时默认是 `manual_rerun`。`manual_rerun` 和 `base_stale_after_land_rerun` 会显式 推进这次运行；其他非空的 trigger 在有可用的既有终态证据时会复用它。新启动或仍在进行的运行通过 Worker Job 送达。响应会报告 `queued`、 `job`、`delivery`、`trigger` 和 Patchset 的 CI 投影。

查看用：

```
curl --get --fail-with-body \
  --data 'recent_limit=10' \
  --data 'projection=readiness' \
  "$BASE/v1/native/repository-authorities/$REPOSITORY_INDEX/read/patchsets/<patchset-id>/ci-stat
    us"
```

`recent_limit` 默认 `10`，必须在 `1` 到 `1000` 之间；唯一有名字的 投影是 `readiness`。
响应会报告可用性、运行序号、完成时间、状态和计数、是否存在可用的证据、 选定的/最新的作业，以及近期作业。1.1.1 的紧凑权威在这里不保留完整的 逐 suite 历史，所以这个读模型里的 `selected_suite_ids` 和 `suite_results` 是空的。

原子式 Task Land

优先用这个路由，而不是直接对 Change 用 `:submit`：

```
{
  "contract": "task-land-atomic/v1",
  "idempotency_key": "client-unique-land-key",
  "task_or_change_ref": "T-0001/C-01",
  "target_line": "main",
  "mode": "direct",
  "patchset_id": "T-0001/C-01/P-02",
  "expected_head_snapshot_id": "SNP-BASE"
}
```

`contract`、`idempotency_key`、`task_or_change_ref` 和 `mode` 是必填的。 key 的上限是 256 字节。只有当某个 Task 恰好只有一个可 land 的 Change 时，`task_or_change_ref` 才可以是 Task；否则请传确切的 Change。 `target_line` 默认取 Change 的基础 Line。mode 取值为 `direct`、`merge` 或 `ff-only`。`patchset_id` 可选，但必须是已选定的那个。比较并设置的 守卫可以叫 `expected_head_snapshot_id`，也可以叫 `expected_target_line_head`。

成功或重放会返回原子契约、重放标志、顶层的 Task、Change、Patchset、 目标和已 land 的 Snapshot 标识符，外加完整的 `task`、`change`、 `patchset`、`land` 投影，以及可选的 `history_promotion` 投影。

直接调用 `POST .../changes/{change_ref}:submit` 也接受这些 Land 字段，
但不要求原子契约。它更底层，也不提供同样的 Task 解析和幂等聚合结果。

用 `GET .../lands/{submission_id}` 读取任意一次尝试。Land 投影包含 选定的 Patchset、目标 Line、模式、状态、失败类型、结果、已 land 的 Snapshot、Line 动作，以及时间戳。失败类型包括基础过期、策略/评审/CI 被阻塞、冲突、目标更新失败和内部错误。

history promotion 的 prepare 操作

这个专用操作会把已记录的本地 land 历史，在受治理的远程 Land 之前转换 成远程工作流记录。顶层请求是：

```
{
  "contract": "history-promotion-prepare/v1",
  "idempotency_key": "client-unique-promotion-key",
  "target_line": "main",
  "base_snapshot_id": "SNP-BASE",
  "revision_snapshot_id": "SNP-FINAL",
  "author_mode": "ai_with_human_review",
  "summary": "Promote recorded local history",
  "entries": []
}
```

`entries` 必须包含 1 到 64 个条目。每个条目包含：

字段	含义
`local_task_id`、`local_change_id`、`local_change_ref`	原来的本地身份。
`expected_remote_task_id`、`expected_remote_change_ref`	可选的失败即中止的发布身份。
`task`	`{title, intent, plan_id?, origin_plan_revision_id?, plan_item_ref?}`。
`change`	`{title, base_line, fork_snapshot_id}`。
`pre_land_target_snapshot_id`、`landed_snapshot_id`、`landed_at_s`	已记录的本地 Land 边界。
`snapshots`	该边界对应的 1 到 64 个 `{snapshot_id, created_at_s}` 条目。

这个端点会在准备聚合 Patchset 之前，校验顺序、身份、Snapshot 祖先、 Plan 绑定和幂等性。裸调用方应当使用兼容的 `ait` 客户端生成的确切回执， 而不是手工拼这份载荷。

Sprint 与 Plan API

HTTP 路径用的是 `sprints`；响应对象用的是 Plan 标识符。一个 Plan revision 由一份 Markdown artifact 加上归一化的条目元数据支撑。

Plan revision 载荷

创建和修订共用这些字段：

字段	是否必需	含义
`title`	创建：是；修订：否	Plan 标题，或对 revision 标题的覆盖值。
`status`	否	`draft`、`active`、`archived` 或 `superseded`；默认是 `draft`。`active` 会按 draft/open 状态存储和投影。
`summary`	否	revision 摘要。
`artifact_path`	是	相对 Repository 的 Markdown artifact 路径。
`artifact_selector`	否	artifact 内部的稳定选择符。
`artifact_heading`	是	给人看的根标题。

字段	是否必需	含义
`items`	否	归一化的条目数组；默认为空。
`artifact_body`	否	用来打包 revision artifact 的确切 Markdown 正文。
`packed_artifact`	否	兼容客户端在用时给出的既有打包 artifact 描述符。
`expected_head_revision_id`	仅修订时可用，否	并发修订的比较并设置守卫。

每个 `items[]` 对象可以包含 `plan_item_ref`、条目文本、 `checkbox_state` 和 `heading_path`。没有可用引用的条目，无法被 Task 关联选中。

用 `POST .../sprints` 创建。调用方自行提供的 `plan_id` 会被拒绝，因为 它由服务端推导。用 `GET .../sprints` 列出；可选的 `artifact_path=<exact-path>` 用来收窄结果。用 `GET .../sprints/{plan_id}` 读取其中一个。

更新状态用：

```
        {"status":"archived"}
```

把共用的 revision 载荷 POST 到 `.../sprints/{plan_id}/revisions` 就能 追加一个 revision。列出或读取 revision 用对应的 GET 路由。

`/artifacts` 的 GET 路由返回的完整 revision 投影，和 revision 的 GET 路由一样。对应的 PUT 存在只是为了协议兼容，但 1.1.1 一律拒绝它，因为 单独写支撑性 artifact 不属于当前生效的紧凑 Plan 布局。请改为把 artifact 正文放进 Plan 的创建或修订操作里。

Plan 查找与 Task 关联辅助接口

查找包含全部给定归一化词条的活动 Plan：

```
{"contains_terms":["release","readback"]}
```

POST 到 `.../sprint-plan-ids/by-contains`。词条必须是 JSON 字符串数组； 归一化后为空集时返回空数组。

解析 Task 关联，把下面这些字段的任意组合 POST 过去：

```
{
  "plan_id": "PR-12",
  "origin_plan_revision_id": "plan-revision:19",
  "plan_item_ref": "site/release/readback"
}
```

既没有 Plan 也没有 revision 时，三个字段都返回 `null`。没有 Plan 关联 的条目会被拒绝。缺失的 Plan 或 revision 身份会在这个数字 Repository 内部解析，然后再校验归属关系和条目是否存在。

`GET .../read/plans/candidate-inputs?contains=release,readback` 会把 逗号分隔的词条归一化，返回匹配的活动 Plan 以及关联的 Task 候选。它是 给选择类界面用的读模型，不是写入端点。

端到端裸 HTTP 示例

这段精简过的流程展示了这些较底层的 API 是怎么串起来的。生产环境的 客户端必须保留每一个返回的标识符，并在遇到任何非成功状态时停下。

1. `GET /healthz`，直到 `200` 且 `ready: true`。
2. `GET /v1/handshake`；核对协议和 runner 契约。
3. 用 `POST /v1/native/repository-authorities` 注册一次；把数字形式的 Repository 索引持久化下来。
4. 用远程同步的 plan、裸 pack 上传和 commit 来发布不可变的 Snapshot 内容，同时不推进已初始化的受治理 `main` Line。
5. 创建，或者原子地启动绑定 Plan 的 Task 和 Change。
6. 发布并选定那个 revision Snapshot 已上传的 Patchset。
7. 加上 attestation 和所需的评审请求。
8. POST Patchset 的 `:runCi`；由一个 `ait-runner` 认领、心跳并完成由此 产生的 Worker Job。
9. POST Patchset 的 `:evaluatePolicy`；查看返回的决定。
10. POST `/task-land`，带上幂等键和期望的目标 head。
11. 读取 Land 和 Task 审计；只有成功的 Land 才会推进受治理的 `main`， 也才允许最终完成 Task。

日常使用请让 `ait workflow ready`、`ait-runner` 和 `ait task finish` 来 走这套流程。直接的 API 是留给兼容集成，以及需要显式控制传输层的运维 人员的。

运维核验清单

- 监听器处于私有网络，或者由可信的 TLS 入口保护。
- 健康检查和握手分开确认；不要把紧凑的端点列表当成完整的 API 目录。
- 每个请求都用保存下来的数字 Repository 索引来选路。
- 写操作的重试，在契约提供幂等键时使用幂等键，在 Line 可能移动的地方 使用比较并设置的 head。

- `503` 要尊重 `Retry-After`；`409` 要去查清楚，而不是闷头重试。
- 二进制媒体类型、字节限额、pack ID、文件大小和摘要都要严格校验。
- 退役导出已完整，并且独立持久化之后，才做清除。
- 恢复流程在隔离的服务器上演练过，并且记下了新分配的 Repository 索引。
- worker 租约令牌只留在内存或受保护的临时状态里，绝不拿来当通用的 API 认证。

关于部署、保存边界和灾难恢复流程，参见 `ait-server`：远程权威与恢复；关于服务端/runner 的运行边界，参见 ■■■■■■。

基础设施与集成

ait-agent：可选的传输管理器

了解可选的 1.1.1 传输管理器边界、它的四类 worker、生命周期命令，以及它与 coding client 和原生 worker 可执行文件的关系。

> 适用人群 Agent 运维、集成维护者与 Repository 所有者

https://ait-native.dev/technical/v1.1.1/ait-agent/

在 ait-native 里的角色

`ait-agent` 是个可选的原生管理器，管的是 Repository 范围内的 Telegram、LINE、Discord 和 Slack worker。它是独立的可执行文件，和 主 `ait` CLI 分开：1.1.1 没有 `ait agent` 这个顶层命名空间，所以要装和守着传输面，运维得直接用 `ait-agent`。

它不是主力的编码应用。日常的交互式工作，还是留在那个已经握着用户 对话和 Repository 会话的 coding client 里。只有当你确实需要一个 无界面的传输端点时，才装这个组件。

管理器与 worker 的边界

组件	职责
`ait-agent`	添加、列出、查看、启动、停止、重启、读日志、删除具名的 worker 配置。Telegram 还额外暴露一套 supervisor 生命周期。
`ait-agent-worker`	跑起选定的原生传输，验证入站请求，调用有界的回复 provider，把回复送出去。
Coding client	拥有主要的交互式编码体验，并决定什么时候执行获授权的 AIT 工作流命令。
`ait-server`	拥有可选的远程 Repository 权威、异地保存、恢复状态和 Worker Job。

启动一个传输 worker，本身不会创建 sprint card、Task、Snapshot、 Patchset，也不会 Land。这些仍然是由获授权的客户端明确执行的仓库 工作流操作。

公开的命令族

四种传输都暴露这几组生命周期命令：

```
add  list  status  start  stop  restart  logs  remove
```

Telegram 还多出一组：

```
supervisor status  supervisor start  supervisor run
supervisor stop    supervisor restart
```

只读的上手示例是这些：

```
ait-agent telegram list --json
ait-agent telegram status main --json
ait-agent telegram logs main --lines 100
ait-agent telegram supervisor status --json
```

add 系列命令收传输凭据和生命周期路径。各个传输要求的选项并不一样； 查自动生成的独立 `ait-agent` 参考， 别把 Telegram、LINE、Discord、Slack 之间的选项互相抄。

配置与密钥边界

管理器命令会在 `.ait/agent-workers.json` 里写入具名条目，并维护它们的运行时元数据。凭据必须当密钥对待：命令要传 token 时别让它留在 shell 历史里，收紧 worker 的环境和运行时文件，不做脱敏就别把日志或 配置副本发出去。

逐字段的清单说明、凭据优先级、传输默认值、回复 provider 和沙箱开关， 都写在 Agent-worker 配置。执行进程的契约和签名的一次性命令，写在 ait-agent-worker。

要不要用它

当一个受管的外部传输，确实能实打实帮运维够到某个 Repository 时， 才用 `ait-agent`。别只为了复刻一个 coding client 的界面，或者为了 把消息功能塞进核心架构，就把它加进来。本地 CLI、原生绑定、Task 隔离、Binary DB，加上可选的 server/runner 路径，没有这个管理器 照样是完整的。

基础设施与集成

ait-agent-worker：消息与应答运行时

通过它经过验证的命令面、四种传输、Codex 应答提供方、运行时准入和安全边界，运维可选的原生消息 worker。

适用人群 Agent 运维、集成维护者与 Repository 所有者

https://ait-native.dev/technical/v1.1.1/ait-agent-worker/

在 ait-native 里的角色

`ait-agent` 和 `ait-agent-worker` 是同一份 `ait-core` 源码权威里出来 的两个独立发布组件。`ait-agent` 管 worker 的配置和进程； `ait-agent-worker` 是不依赖 Python 的原生传输与回复执行运行时，服务 Telegram、LINE、Discord 和 Slack。

日常用 `ait` CLI 不需要这个 worker，它也不是另一个桌面或移动端的 coding client。交互式的编写体验，仍然由主 coding client 负责。 `ait-agent-worker` 从一个明确配置好的传输收下消息，把它绑到一个 Repository 上下文，拿到一份有界的回复，再顺着同一个传输送回去。

worker 不会自动去建 sprint card、开 Task、创建 Snapshot 或者 Land 一个 Change。只有当回复 provider 或者别的获授权客户端明确跑了对应的 `ait` 命令时，这些工作流变更才会发生。

适合拿来做什么

- 在一个 AIT Repository 上跑一个具名的消息机器人，还不用装 Python worker 运行时。
- 通过为某个服务实现的那种传输模式，接 Telegram、LINE、Discord 或 Slack 的请求，返回带 Repository 上下文的回复。
- 把一个原生的无界面 worker 放在受信任的服务守护进程或 HTTP 适配层 后面。
- 在把某个安装好的可执行文件放进生产之前，查清它编译进去的确切传输、 事件循环、平台和回退能力。
- 用 worker 自带的原生 Codex 回复 provider，或者为受控集成明确指定 一个自定义的本地回复程序。

远程 Repository 权威、异地保存、恢复、Patchset CI 和 runner 的 job 协调，这些该用 `ait-server`，不是 `ait-agent-worker`。worker 可以 解析出本地或远程的 Repository 目标，但它不是服务端的管理 API。

从消息到回复的执行路径

1. 进程先解析出 Repository 根目录、`.ait/config.json` 和 worker 清单。`AIT_AGENT_CONFIG_PATH` 可以指向另一个清单路径。
2. 清单被规范化，然后选中确切的 `<kind>/<name>` worker。字段非法、 worker 不存在、kind 和 name 对不上，都会在传输启动前就失败。
3. 解析凭据和运行时设置。生效的工作流模式决定了 Repository 目标是 本地的，还是走它配置好的默认 remote。
4. 请求的原生事件循环后端和从零开始编号的分片，会拿去和运行时准入 计划核对。后端或分片对不上就直接失败。
5. 传输完成认证或核验它自己那套入站校验，解析出确切的请求，套用 会话状态和去重状态，然后提交一个有界的回复 job。
6. 配置好的本地回复程序开跑。没有提供完整的自定义 provider 时， 原生 worker 会把自己的 `reply-provider` 子命令装成 provider，用 选定的模型、推理强度、沙箱和超时去调 Codex。
7. 传输把回复发出去或返回。失败走结构化的原生诊断，不会回退到 Python worker。

worker 的同步状态和 Codex 线程绑定，把传输侧的会话身份跟 AIT 的 工作流权威分了开。恢复一段会话可以复用它兼容的 Codex 线程，而 Task、Change、Snapshot 和 Land 的状态仍然归 AIT 所有。

可执行文件的完整命令面

命令	作用、输入和结果
`capabilities`	报出安装的平台、架构、socket 与进程控制后端、支持的传输、事件循环后端、默认后端，以及 Python 回退状态。默认输出文本；`--json` 输出 `ait.agent.worker.capabilities.v1`。
`run`	在配置规范化和运行时准入之后，跑起一个确切的具名 worker。它要求传输、worker 名、事件循环后端和分片。服务模式是长期运行的。Telegram 还允许通过 console 模式收下一次 stdin webhook 事务。
`slack-command`	执行一次签名过的 Slack 斜杠命令事务。它从 stdin 读取确切的表单 body，核验受信任 HTTP 边界传进来的请求元数据，然后写出一份脱敏的 JSON 结果。
`discord-interaction`	执行一次签名过的 Discord interaction 事务。它从 stdin 读取确切的 JSON body，核验受信任 HTTP 边界传进来的请求元数据，然后写出 Discord 的响应对象。
`reply-provider`	执行一次带版本的原生 Codex 回复 provider 请求。它从 stdin 读 JSON，写出 `ait.agent.gateway_reply_provider_response.v1` JSON。

1.1.1 的确切写法：

```
ait-agent-worker capabilities [--json]

ait-agent-worker run \
  --transport <telegram|discord|slack|line> \
  --worker <name> \
  --event-loop-backend <backend-reported-by-capabilities> \
  --shard <zero-based-index> \
  [--console-mode <service|webhook>]

ait-agent-worker slack-command [--worker <name>] \
  --signature <signature> --signature-timestamp <unix-seconds> < exact-slack-body
ait-agent-worker discord-interaction [--worker <name>] \
  --signature <signature> --signature-timestamp <timestamp> < exact-discord-body
ait-agent-worker reply-provider < provider-request.json
```

`run` 的 console 模式默认是 `service`。只有在 stdin 上喂一次 Telegram webhook 载荷时才用 `webhook`。Slack 和 Discord 那两个一次性 命令是适配层边界，不是不认证的网络监听器：调用方必须原样保住那份 raw body，并通过受信任的请求边界把匹配的签名和时间戳元数据一起传 进来。

启动之前，把两种形式的能力证据都记下来：

```
ait-agent-worker --version
ait-agent-worker capabilities
ait-agent-worker capabilities --json
```

别光凭操作系统名字去猜后端或传输。以安装好的二进制报出来的能力结果 为准，再把启动规划选定的后端和分片传给 `run`。

各个传输的行为

传输	长期运行的服务	有界的单次请求模式	认证与投递边界
Telegram	轮询，或者配置成 webhook 服务。	`run --console-mode webhook` 从 stdin 消费一次 webhook 载荷。	Bot token 是必须的。webhook 模式可以要求配置好的 webhook 密钥。回复支持消息合并、Markdown、解耦投递，以及可选的本地语音转文字设置。
LINE	在配置的 host、端口和路径上跑 HTTP 回调服务。	没有单独的公开一次性子命令。	Channel secret 用来验回调签名；channel access token 负责通过配置的 LINE API base URL 投递回复。
Discord	选定模式具备所需 bot 凭据时，跑 Gateway 服务。	`discord-interaction` 处理一次签名过的 interaction body。	应用身份决定选中哪个 worker。选定的 interaction 或 gateway 模式会要求公钥校验和 bot 投递凭据。
Slack	选定的 worker 有 app token 时，走 Socket Mode。	`slack-command` 处理一次签名过的斜杠命令表单 body。	签名密钥用来验 HTTP 命令。Socket 模式和 HTTP 模式各用各自需要的凭据；延迟投递的命令响应受这次命令事务的边界约束。

所有监听器默认都绑在 loopback 地址上。要把 HTTP 监听器放到受信任的
反向代理后面，得先拿确切选定的那个传输，把请求校验、TLS、body 大小、 超时和转发行为都测过再说。

原生 Codex 回复 provider

传输 worker 需要一个回复 provider；它们自己不带第二个编码模型。配置 里给了完整的 `local_reply.program` 加 `local_reply.args`，那就以它 为准。否则原生可执行文件就跑自己的 `reply-provider` 子命令，把自己 配成 provider。

原生 provider 接收带版本的 `ait.agent.gateway_reply_provider_request.v1` JSON 契约。请求里绑定 了一个 Repository 根目录、会话 key、surface、actor、输入载荷、可选的 上一个 provider 线程，以及回复设置。它为那个兼容的 Repository 和会话 启动或恢复一个 Codex 线程，抓取最终回复和有界的这一轮遥测数据， 再返回带版本的响应契约。

provider 会强制约束请求和进程输出的大小上限、一把线程锁、配置好的 超时，以及 `read-only`、`workspace-write`、`danger-full-access` 三个 沙箱取值之一。选哪个沙箱就是执行边界。它必须和这个 worker 在那个 Repository 里被允许做的事对得上；传输凭据不会额外给出文件系统权限 或者 AIT 权威。

准入、并发与关停

`run` 从不随便挑一个分片。它会规范化整份 worker 清单，把配置好的 worker
集合按选定的原生事件循环后端排一遍计划，再核对请求的那个 worker 确实属于传进来的分片。Telegram
还能额外限定预期的并发 worker 数和每个分片的 worker 数。

传输的回复工作走的是有界的原生 job 准入。容量满了，worker 会直接
报失败，而不是攒出没有上限的后台工作。服务在把已准入的工作排干之前， 会先停止接新
job；进程控制的具体行为，就是 `capabilities` 报出来的 那个原生后端。

失败与安全契约

运行期的失败以 `ait.agent.worker.error.v1` JSON 的形式写到 stderr。 这份诊断里包含一个稳定的 code、消息、数字退出码、有界的细节，还有 `python_worker_execution_allowed: false`。

退出码	含义
2	请求非法，包括输入格式不对，或者签名入站被拒。
3	worker、清单、凭据、后端或分片配置非法。
4	必需的原生运行时、传输、进程或输出边界不可用。

诊断只报凭据在不在，不报具体值。Slack 和 Discord 校验的是确切的 raw
body。一次性命令的结果会把密钥和投递定位信息脱敏。1.1.1 的传输、 签名、回复和状态相关工作，Python
回退一律关闭。

有意划下的界限

- `ait-agent-worker` 不是通用的桌面、移动或 Web 应用。
- 它不是持久的远程权威，也替代不了 `ait-server`。
- 收到一条消息，本身不会创建或完成任何 AIT 工作流对象。
- 一个传输 worker，每次进程调用只对应一个解析出来的 Repository 和 一个选中的具名 worker。
- 这些公开命令不会绕过传输签名、清单校验、运行时准入、Codex 沙箱 或者 AIT 的工作流策略。

配置索引

- 附录：Agent-worker 配置 展开讲 `.ait/agent-workers.json`、凭据优先级、每个传输字段、本地 回复设置、运行时路径和核验方法。
- 附录：环境变量 列出受支持的 worker 引导和凭据类环境输入。
- ait-server：远程权威与恢复 讲清楚另一套远程权威、保存和恢复的边界。

基础设施与集成

Python 集成

使用 ait-native PyPI 发行版里自带的、直接在进程内工作的 Python 绑定。

适用人群 Python 集成方

https://ait-native.dev/technical/v1.1.1/integrations/python/

包边界

PyPI 上的项目名是 `ait-native`。它的平台 wheel 里带着已接纳的原生 命令，以及来自同一个 1.1.1 兼容家族的、直接在进程内工作的 Python 绑定。

```
python -m pip install ait-native==1.1.1
```

这个绑定不会去下载运行时，不会拿 `ait` 子进程当 API 的传输通道， 也不会另做一套工作流实现。

直接调用 API

```
from ait_python import AgentClient, NativeRuntime

runtime = NativeRuntime()
print(runtime.binding_info())

agent = AgentClient(runtime)
print(agent.capabilities().supported_transports)
```

`NativeRuntime` 在进程内暴露带版本的原生导出。`AgentClient` 提供 带类型的入口，用来访问受支持的 agent 能力和 worker 操作。

仓库工作流

装上 Python 包并不会初始化仓库。在目标仓库里跑一次 `ait init`， 之后让 coding agent 按生成出来的说明走。

```
ait init
ait status --json
```

Python 项目走的 AIT 工作流和别人没有区别。测试、lint、构建和打包 命令，仍然是由仓库自己写好的输入。

版本核对

把绑定和别的组件搭在一起之前，先读一下绑定的元数据。自己重新编过 的绑定，或者对不上的内核，不会因为 Python 包能 import 就算是已接纳 的 1.1.1 家族。

家族边界和许可证边界见 分发与发布。

基础设施与集成

Node.js 集成

使用受支持的 npm 包提供的、直接的 Node-API JavaScript 与 TypeScript 接口。

适用人群 Node.js 集成方

https://ait-native.dev/technical/v1.1.1/integrations/node/

包边界

受支持的 npm 名字是 `@wa120/ait-native`。它带的是可移植的 JavaScript 和 TypeScript 门面层，并挑出确切的 1.1.1 平台 Node-API addon。

```
npm install @wa120/ait-native@1.1.1
```

这个包不打包也不启动 `ait-server`。它的 API 直接加载包自带的 addon，不拿子进程当绑定的传输通道。

直接调用 API

```
import { NativeRuntime } from "@wa120/ait-native";

const ait = new NativeRuntime();
console.log(ait.bindingInfo());
const status = ait.runCli(["status", "--json"]);
console.log(status);
```

包里带的 `ait` 命令，在进程内调的是同一个原生绑定。所以 JavaScript API 和这个命令共享同一套已接纳的 Rust 行为。

仓库工作流

Node.js 仓库的初始化方式和生成出来的仓库说明，跟其他受支持的仓库 完全一样。

```
npx --no-install ait init
npx --no-install ait status --json
```

AIT 不会去翻 `package.json` 来挑测试命令或者判断框架行为。要验证 什么，仍然由仓库自己明确说清楚。

平台包

按目标平台拆出来的 addon 包，是锁定版本的实现依赖，不是单独的产品。 应用应该依赖顶层那个受支持的包，让 npm 的常规解析去选出匹配的 addon。

BINARY DB V0 结构

Binary DB v0：格式与权威

查看 Binary DB v0 的文件格式、整数与偏移规则，以及存储根边界。

适用人群 内核实现者、集成方与运维

https://ait-native.dev/technical/v1.1.1/binary-db/

1.1.1 文档路由 · Binary DB v0 · `layout_id = 1`

本章是完整的活动 Binary DB v0 技术 schema 的入口。它说明每一条固定记录、 payload 和索引是如何寻址的，以及归哪个存储根所有。

未指定的行为一律保留并失败拒绝：实现不得自行发明扩展句柄编码，不得悄悄加宽 某个字段，也不得在已声明的 layout 转换之外选用新的规范 payload 编码。

全局文件格式

每个持久化的 Binary DB `.bin` 文件、以及每个可重建的 `.idx` 文件，都以一个 四字节头部开始：

```
BIN_HEADER_SIZE = 4
INDEX_HEADER_SIZE = 4

BinHeader / IndexHeader:
u32 layout_id = 1     # little-endian
```

文件路径决定记录种类、命名空间、权威作用域，以及该文件属于权威存储、payload 存储、对象数据还是可选缓存。头部不含 magic、记录数、校验和、权威名称、 generation 令牌，也不重复记录文件种类。

对于定长记录文件：

```
record_count  = (file_size - BIN_HEADER_SIZE) / record_size
record_offset = BIN_HEADER_SIZE + record_index * record_size
```

`file_size` 至少为四字节，且记录区必须能被所选记录大小整除。同一个文件绝不能 混放来自不同 layout 的记录。记录编解码器使用声明的字节偏移，不依赖 Rust 结构体布局、宿主对齐或隐式填充。

payload 偏移是 `u64` 文件地址，从所属 payload 文件的第 0 字节起算。第一个 payload 通常从偏移 `BIN_HEADER_SIZE` 开始。payload 与定长记录的整数字段使用 仓库固定的字节序。保留位和保留字段一律写零。

除非某字段是单独声明的有符号 Git identity 时间戳，否则每个活动的定长记录 `*_at_s` 字段都是小端无符号 `u64`，表示自 Unix 纪元起的整秒数。早于纪元的输入 失败拒绝。零保持该字段声明的"缺失"或"未知"这一确切含义。不得从时间戳精度推断任何顺序或唯一性。

`*_index_plus1 = 0` 表示缺失；非零值编码 `index + 1`，因为记录索引零是合法的。 不带 `plus1` 后缀的索引字段存的是直接的记录索引，合法地可以为零。

权威作用域

当权威根不同时，同一个文件名可能使用不同的记录 layout：

- 本地工作流权威拥有本地 Task、Change、Line、 Tag、Stash、Land、Land-target-Line、task-snapshot-link 和 Plan 状态。
- 一个 `ait-server` 仓库权威拥有远端 Task、Change、Line、Land、Land-target-Line、task-snapshot-link、Plan、Patchset、Worker Job 及其 ready/state 索引，以及 Attestation、Actor、Review、Policy 和 Waiver 状态。
- 一次 `ait-server` 安装拥有一个独立的服务端全局运行权威根，且仅用于 Repository 注册表及其命名空间查找索引。`repository_index` 指向该根的 `repository.bin`，并路由到某一个服务端 Repository 权威。Worker Job 的身份 只有作为 `(repository_index, worker_job_index)` 才有意义；第二个分量是被 路由到的那个 Repository 权威本地的永久物理记录索引。

- 共享内容权威拥有内容 Snapshot、有序 Snapshot-parent、Tree、归一化的 Tree-entry-range、Blob 以及 object-pack 元数据。
- 可选的本地 Git 互操作权威拥有 Git 导入/导出的仓库身份、generation、 不可变映射和可续跑的 checkpoint。它是一个逃生舱式的映射权威，不是 AIT 的主工作流权威或内容权威。
- 一个 task worktree 拥有可丢弃的 `.ait-worktree/cursor.bin` 状态。
- 可选的本地内容缓存拥有 manifest/file 查找加速。
- 可选的本地 remote-mirror 根可以镜像服务端的 snapshot link，同时保留服务端 索引。

不同权威根中数值相同的索引不是同一个身份。本地写入方绝不分配服务端索引。

BINARY DB V0 结构

Binary DB v0：工作流记录

展开 Task、Change、worktree、Line、Stash、Tag、Land 以及内联工作流链接记录的布局与不变量。

适用人群 内核实现者与工作流集成方

https://ait-native.dev/technical/v1.1.1/binary-db/workflow/

1.1.1 文档路由 · Binary DB v0 · `layout_id = 1`

这些 layout 是本地与远端工作流状态的定长记录和关联记录。阅读每个字节布局时，要连同后面的身份、变更、校验和遗留转换规则一起读；仅凭记录大小常量并不构成完整契约。

Task 记录

```
LOCAL_TASK_RECORD_SIZE = 64

LocalTaskRecord - task.bin:
u8  task_meta
u8  local_meta
u16 payload_len
u64 payload_offset
u32 origin_plan_revision_index_plus1
u32 plan_item_index_plus1
u32 published_remote_task_index
u64 created_at_s
u64 updated_at_s
u64 plan_linked_at_s
u64 published_at_s
u64 closed_at_s
```

```
REMOTE_TASK_RECORD_SIZE = 60

RemoteTaskRecord - task.bin:
u8  task_meta
u8  remote_meta
u16 payload_len
u64 payload_offset
u32 origin_plan_revision_index_plus1
u32 plan_item_index_plus1
u64 created_at_s
u64 updated_at_s
u64 plan_linked_at_s
u64 fetched_at_s
u64 closed_at_s
```

Task 的身份是推导出来的，不是存储的：

```
task_index = record_number
task_seq   = task_index + 1
task_id    = derive(authority_scope, repo_namespace, "T", task_seq)
```

Task 记录只追加，绝不能在物理上重排。遗留的 SQLite 导入可以对稀疏的遗留序列做一次压紧；它不得通过插入填充记录来保留空洞。

`closed_at_s` 是 `LocalTaskRecord` 和 `RemoteTaskRecord` 的固定尾部字段， 不是 payload，也不是单独的记录或文件。零表示：要么有效的 Task 状态不是终态， 要么离线遗留转换对某个终态 Task 没有拿到源的关闭时间。后一种情况下 Task 已 关闭，而它的历史关闭时间未知。非零值即关闭时间；当且仅当在 `TaskRecord.task_meta` 之下（对本地权威还要加上 `LocalTaskRecord.local_meta`）有效的 Task 状态为终态时，该值才有效。后续的 发布或关联更新可以改变 `updated_at_s`，但绝不替换关闭时间。

创建 Task 时写入 `closed_at_s = 0`。关闭操作先写入并 fsync `closed_at_s`， 之后才写入并 fsync 作为提交标记的终态状态位。原生写入方绝不会清除终态状态来 重新打开一个 Task。读取方把非终态 Task 上残留的非零时间视为缺失，恢复流程会把 它修复为零。原生 v0 写入方绝不能创建关闭时间为零的终态 Task。由于不存在迁移 来源位或字段，读取方、校验器和恢复流程把"终态加零"接受为未知的历史关闭时间，不把这种表示归类为损坏。

修正前的 layout-1 本地 40 字节和远端 36 字节 Task 记录只能作为离线转换输入。
转换器必须显式收到源记录大小，绝不能仅凭文件大小可整除性来推断。在独占的权威 锁之下，它用 44 字节或 40 字节记录把完整的 `task.bin` 重写到一个单独的文件，
并保留源权威提供的每一个关闭时间。对于遗留源格式可证明没有记录关闭时间的终态 Task，它写入 `closed_at_s = 0`，不得以创建时间、更新时间、转换时间或任何其他
推断出的时间戳来替代。定义了关闭时间但提供了非法值的源，仍然失败拒绝。转换器
校验整个重写后的文件，并在激活之前原子地替换旧文件。一个活动的 `task.bin` 绝不混用修正前和修正后的记录大小。

对于 bin 到 bin 的转换，源的 planning 状态 `unplanned` 和 `explicit_unplanned` 都编码为 `TaskRecord.task_meta` 位 0 未置位；`planned` 编码为位 0 置位。这就是 v0 完整的 planning 状态编码：v0 有意不保留这两种 unplanned 源写法之间的第三种区分。数值形式的 Plan revision 与 item 绑定会被 独立解析并保留。只有当文本形式的 `plan_section_ref` 或 `plan_drift_state` 缺失、或者能被精确推导时，才在不新增字段的情况下被接受：section ref 来自解析 出的 Plan Item 的 `heading_path_bytes`，而 drift 状态来自数值绑定以及当前的 Plan/Plan-revision 元数据。非推导得出的值或有歧义的值失败拒绝。

被接纳的源 Task 状态 `abandoned` 映射到 `TaskRecord.task_meta` 位 7 `canceled`。它不会新建另一个 Task 状态字段，也不会 从服务端源推断 `LocalTaskRecord.local_meta` 位 1。在单独列举的、确切的锁定源
门禁之下，遗留源写法 `canceled` 映射到同一个已有位，不映射到其他任何状态。
显式携带独立的本地放弃状态的本地源，按其既有语义保留该本地位。

对于从可证明没有更新时间或抓取时间的遗留 Remote Task 格式做的离线转换， 转换器相应地写入 `updated_at_s = 0` 或 `fetched_at_s = 0`。每个零表示未知的
历史时间，而不是转换时间。提供的合法时间被原样保留；定义了其中任一字段却提供 非法值的源格式失败拒绝。原生 v0 的 Remote Task 写入方继续写入它们实际的事件 时间，不使用这条遗留例外。

Change 记录

```
TASK_CHANGE_INDEX_RECORD_SIZE = 8

TaskChangeIndexRecord - task_change_index.bin:
u32 latest_change_index_plus1
u16 change_count
u8  next_change_ordinal
u8  reserved0
```

```
LOCAL_CHANGE_RECORD_SIZE = 68

LocalChangeRecord - change.bin:
u8  change_meta
u8  local_meta
u16 payload_len
u8  change_ordinal
u8  change_state
u16 reserved1
u64 payload_offset
u32 task_index
u32 previous_change_index_plus1
u32 fork_snapshot_index_plus1
u8  published_remote_change_ordinal_plus1
u8  reserved2
u16 reserved3
u64 created_at_s
u64 updated_at_s
u64 published_at_s
u32 base_line_index_plus1
u64 archived_at_s
```

```
REMOTE_CHANGE_RECORD_SIZE = 68

RemoteChangeRecord - change.bin:
u8  change_meta
u8  remote_meta
u16 payload_len
u8  change_ordinal
u8  change_state
u16 reserved1
u64 payload_offset
u32 task_index
u32 previous_change_index_plus1
u32 selected_patchset_index_plus1
u32 fork_snapshot_index_plus1
u64 created_at_s
u64 updated_at_s
u64 fetched_at_s
u32 base_line_index_plus1
u64 archived_at_s
```

Change 记录既不存放文本形式的 Change ID，也不存放文本形式的 Task ID。它的 身份是 `(authority_scope, task_index, change_ordinal)`。

已发布的 Local Change 的 Remote 身份，是由其所属 Local Task 的 `published_remote_task_index` 和 `published_remote_change_ordinal_plus1 - 1` 组成的二元组。Task 字段选定确切的 Remote Task ordinal；Change 字段在该 Task 内部选定 `C-01..C-64`。 没有任何 Local Change 存储或需要一个全局的 Remote `change_index`。

对于 Remote Change， `ChangePatchsetIndexRecord.latest_patchset_index_plus1` 是最新/当前的 Patchset，而 `selected_patchset_index_plus1` 是唯一持久化的、可变的 selected-Patchset 权威，供评审、就绪判定和 Land 使用。选定是 Change 的状态， 不是 Patchset 的不可变属性。零表示没有选定 Patchset。非零的 selected 指针必须 引用一个由该 Change 拥有的存活 Patchset。 源的 `current_patchset_number` 通过 latest 索引解析； 源的 `selected_patchset_number` 通过 selected 指针解析。两者可以不同，绝不能被合并成一个值。选定操作更新这个已有的指针，不改动任何 Patchset 记录。这只是对已有四字节槽位的一次语义重命名，不会移动该槽位，也不会移动 44 字节前缀中的任何字段。后来的内联生命周期扩展只追加两个已声明的尾部字段。

仅对被接纳的遗留服务端格式而言，一个已 land 且有一个或多个存活 Patchset 的 Change，可能提供 `current_patchset_number = 0`，尽管 latest/current 仍然可以 重建。只有当源的 selected 指针非零，且解析到该 Change 拥有的、`patch_ordinal` 最大的那个存活 Patchset 时，转换才接受这个零；此时它把那个 Patchset 写为 `ChangePatchsetIndexRecord.latest_patchset_index_plus1`。其他任何为零、缺失、 有歧义、非本 Change 拥有、被省略或互相矛盾的 latest 证据，都失败拒绝。selected 指针仍然独立存储，不会从这条例外推断出来。

`ChangeRecord.change_state` 位 0 是 `canceled`；位 1 到位 7 保留且为零。 canceled 位只有在 archived 生命周期下才有效。源的 Change 状态 `abandoned` 映射为 archived 生命周期、canceled 置位、以及已有的 superseded 位未置位。 它不会新建另一条 Change 记录或 payload。对于可证明没有抓取时间的遗留 Remote Change 格式，转换写入 `fetched_at_s = 0`；提供的合法值被保留，提供的非法值 失败拒绝。

`published_remote_change_ordinal_plus1 = 0` 表示缺失。值 `1..64` 编码 Remote Change 的 ordinal `0..63`，所以 `C-01` 存为 `1`，`C-64` 存为 `64`；值 `65..255` 非法。`reserved2` 和 `reserved3` 必须为零。当且仅当 ordinal 字段为零时，`LocalChangeRecord.local_meta` 位 0 才未置位。该位置位时，ordinal 字段和 `published_at_s` 都非零，并且所属的 Local Task 也必须置位它的 published 位。Remote 的 ordinal 可以与 Local 的 `change_ordinal` 不同；发布时存的是 Remote 权威返回的、确切的 Task 作用域 ordinal。

读取遗留的显式文本 `published_change_id` 的离线转换器，解析出确切的 `C-01..C-64` 后缀，并直接写入它的 ordinal 加一的值；它绝不伪造全局的 Remote Change 索引。按旧有的固定 `u32 published_remote_change_index` 解释来读取的转换器，必须显式选择那个源 schema， 并在写入本字段之前，把被引用的 Remote 记录解析到它所属的 Task 及 Task 作用域 的 ordinal。发布权威缺失、不匹配或有歧义时失败拒绝。

内联 Change 生命周期字段

`base_line_index_plus1` 和 `archived_at_s` 是每条 68 字节的 Local 或 Remote Change 记录中必需的固定字段。它们不是 payload，不是单独的记录，也不是单独的 文件。 `base_line_index_plus1` 是必需的，引用创建该 Change 时所依据的那条确切的 `LineRecord`。Change 绝不重复存放 base-Line 名称或文本形式的 Line ID。若 `fork_snapshot_index_plus1` 非零，它引用创建该 Change 时被观测为该 base Line 的 head 的那个确切的存活 Snapshot。Snapshot 的 `line_index_plus1` 是它不可变的 创作 Line 身份，而不是 Line-head 归属；当在另一条 Line 上创作的 Snapshot 成为 base Line 的 head 之后，它可能与 `base_line_index_plus1` 不同，并且不得被重写， 也不得拿来做相等比较。公开的 `forked_from_line` 值由数值形式的 base-Line 身份 推导而来。

当且仅当 Change 的生命周期为 archived 时，`archived_at_s` 才非零；例外是：从 可证明没有记录归档时间的遗留源格式转换而来的 archived Change，保留零作为未知 的历史归档时间。`ChangeRecord.change_meta` 位 7 在该 archived 生命周期内区分 出被取代的 Change，而 `ChangeRecord.change_state` 位 0 区分取消。归档和重新 打开各自重写并持久提交一条完整的 68 字节 Change 记录；生命周期位与 `archived_at_s` 一起变化，不涉及旁路文件排序，也不涉及记录内部的状态最后写入 协议。恢复流程拒绝非 archived 却带有非零归档时间的 Change。原生 v0 的归档写入 要求归档时间非零。由于没有新增迁移位，读取方、校验器和恢复流程把"archived 加零"接受为未知的遗留时间。公开的 `review` 状态是 active 生命周期加上 `review_pending`；`landed_at` 由成功的 Land 记录的 `updated_at_s` 推导，不会在 Change 上再存一份。

创建 Change 时，通过精确的 payload 比较来解析归一化的 base-Line 名称，把该 Line 当前的 head 观测为可选的 fork Snapshot，然后追加完整的 Change 提交记录。 对于已提交的 Change，恢复流程会拒绝：base Line 为零或缺失、Line 缺失或有歧义、 fork Snapshot 缺失或已打墓碑、生命周期与时间不匹配，或 superseded 位非法。 恢复流程不会把 fork Snapshot 的创作 Line 与 Change 的 base Line 做比较。

紧邻的上一版活动 layout-1 表示，即 44 字节的 `change.bin` 加上按 ordinal 对齐 的 8 字节 `change_lifecycle.bin`，只能作为离线转换输入。转换器必须显式收到那个 源表示，要求两个文件的记录数相同，保留每个 Change 完整的 44 字节前缀，追加 配对的确切 base-Line 索引和归档时间，校验得到的完整 52 字节文件，并在激活的 目标中省略 `change_lifecycle.bin`。它不得推断缺失的 base Line，也不得按任何 文本身份来对齐行。

更早的、旁路文件出现之前的 layout-1 generation，只有在它声明的源 schema 独立 提供确切的 `base_line` 时，才同样只能作为离线转换输入。转换把该值解析为权威 本地的 `line_index`，要求任何提供的 `forked_from_line` 都等于那条 base Line， 并保留确切引用的存活 fork Snapshot，不比较也不重写它的创作 Line。它保留提供的 合法源归档时间，只有当被接纳的源格式可证明没有归档时间字段时，才写入 `archived_at_s = 0`。提供的时间非法，或任何必需的非时间值不可得或有歧义，都会 在物化完整的 68 字节 Change 文件之前失败拒绝。

Worktree 游标

```
WORKTREE_CURSOR_RECORD_SIZE = 28

WorktreeCursorRecord - .ait-worktree/cursor.bin:
u8  cursor_meta
u8  reserved0
u16 reserved1
u32 task_index
u32 selected_change_index_plus1
u32 pending_pre_land_target_snapshot_index_plus1
u32 pending_landed_snapshot_index_plus1
u64 updated_at_s
```

该游标只在本地存在，是可丢弃的临时状态。它不是工作流历史，必须排除在内容 snapshot 之外。

Line 与 Stash 记录

```
LINE_RECORD_SIZE = 40

LineRecord - line.bin:
u8  line_meta
u8  reserved0
u16 line_name_len
u64 line_name_offset
u32 head_snapshot_index_plus1
u64 created_at_s
u64 updated_at_s
u64 archived_at_s
```

```
STASH_RECORD_SIZE = 8

StashRecord - stash.bin:
u8  stash_meta
u8  reserved0
u16 reserved1
u32 stash_snapshot_index
```

Stash 存储只在本地。`stash_snapshot_index` 必须引用一个种类为 `stash` 的内容 snapshot。删除 Stash 会给记录打上墓碑，绝不移动后面的索引。

本地 Tag 记录

```
TAG_RECORD_SIZE = 24

TagRecord - tag.bin:
u8  tag_meta
u8  reserved0
u16 payload_len
u64 payload_offset
u32 snapshot_index
u64 created_at_s
```

`tag_index` 是稳定的内部 Tag 身份，也就是 `TagRecord` 在 `tag.bin` 中从零开始 的 ordinal。公开的 Tag 身份仍然是它归一化的、非空的 UTF-8 名称。 `snapshot_index` 引用一个存活的内容 Snapshot。

创建 Tag 时先追加它带类型的 payload，再写它的固定记录。强制替换一个已有名称会 保留 `tag_index`，追加一份新的 payload，然后更新固定记录；替换过程被打断时， 可能只留下一份无人引用的 payload。删除会设置墓碑位，绝不删除被引用的 Snapshot， 也不移动 Tag 的 ordinal。重新创建同一个名称只会清除墓碑位，同时保留同一个 `tag_index`。

Land 记录

```
TASK_LAND_INDEX_RECORD_SIZE = 8

TaskLandIndexRecord - task_land_index.bin:
u32 latest_land_index_plus1
u16 land_count
u16 reserved0

CHANGE_LAND_INDEX_RECORD_SIZE = 8

ChangeLandIndexRecord - change_land_index.bin:
u32 latest_land_index_plus1
u16 land_count
u8  next_land_ordinal
u8  reserved0
```

```
LOCAL_LAND_RECORD_SIZE = 44

LocalLandRecord - land.bin:
u8  land_meta
u8  land_ordinal
u8  change_ordinal
u8  failure_kind
u32 change_index
u32 previous_task_land_index_plus1
u32 previous_change_land_index_plus1
u32 pre_land_target_snapshot_index_plus1
u32 landed_snapshot_index_plus1
u64 submitted_at_s
u64 updated_at_s
u32 target_line_index_plus1
```

```
SERVER_LAND_RECORD_SIZE = 48

ServerLandRecord - land.bin:
u8  land_meta
u8  land_ordinal
u8  change_ordinal
u8  failure_kind
u32 change_index
u32 patchset_index
u32 previous_task_land_index_plus1
u32 previous_change_land_index_plus1
u32 pre_land_target_snapshot_index_plus1
u32 landed_snapshot_index_plus1
u64 submitted_at_s
u64 updated_at_s
u32 target_line_index_plus1
```

本地 Land 权威就是存储的那对 snapshot。服务端 Land 权威还存储确切的被接受的 Patchset。保持同一个 `L-##` 的 Land 状态更新，采用状态最后写入的两阶段写。

`land_ordinal` 在 `(change_index, land_ordinal)` 内唯一。值 `0..63` 在完整的 所属 Change 身份之下渲染为 `L-01..L-64`。服务端 Land 的 `patchset_index` 是那次尝试所接受的不可变 Patchset；它不会让 Land 身份变成 Patchset 作用域的。因此一个 Change 可以同时保留针对某个 Patchset 的 `L-01` 和针对之后某个 Patchset 的 `L-02`。`ChangeLandIndexRecord` 负责下一个 ordinal 的分配，它的 latest/previous 链接 遵循 Change 作用域的 Land ordinal。`TaskLandIndexRecord` 和 `previous_task_land_index_plus1` 只保留物理上的 Task 清单。

Land 模式存放在 `land_meta` 的位 5 到位 6。源的 `direct`、`merge` 和 `ff-only` 映射到下面的固定编码；未知模式失败拒绝。源的结果对象不会变成 payload。目标 Line 身份以及 pre-land/landed 的 Snapshot 值来自固定的 Land 记录；被阻塞结果的 `blocker_class` 映射到 `failure_kind`。`BASE_STALE` 和 `POLICY_BLOCKED` 分别映射到 `base_stale` 和 `policy_blocked`。任何无法从该 固定记录精确推导出来的、无法识别的 blocker 或结果值，都失败拒绝。

原生 v0 的 Land 提交要求 `submitted_at_s` 非零。仅在离线遗留转换中，一个已 land 的 Change，如果其源权威记录了确切的 land 结果却从未记录提交时间，可以把 它第一个也是唯一一个成功的 Land 物化为：`land_ordinal = 0`、两个 previous-Land 链接均为零、`submitted_at_s = 0`，以及 `updated_at_s` 等于被保留的源 `landed_at`。零表示未知的历史提交时间；转换器不得用 Change 的创建/更新时间或转换时间来替代。当目标 Line、landed Snapshot、必要时的被接受 Patchset，或者 land 时间缺失或有歧义时，本规则不允许重建；原生 v0 写入方也绝不能创建提交时间 为零的新 Land。

内联 Land 目标 Line 权威

所属 Land 记录的 `target_line_index_plus1` 是必需的，引用提交 Land 时选定的那 条确切的目标 `LineRecord`。它属于固定 schema，不是 payload。Land 绝不重复存放 目标 Line 名称或文本形式的 Line ID。

存在时，`pre_land_target_snapshot_index_plus1` 引用目标 Line 更新之前被观测为 其 head 的那个确切的存活 Snapshot，而 `landed_snapshot_index_plus1` 引用被装入 为该 Line 新 head 的那个确切的存活 Snapshot。被引用 Snapshot 的 `line_index_plus1` 仍然是它不可变的创作 Line 身份，可能与 `target_line_index_plus1` 不同；Land 绝不给它重新贴标签，也不把这两个 Line 索引拿来做相等比较。已 land 的 Change 的公开 `target_line` 和 `landed_at` 值， 由它成功的那次 Land 的目标 Line 和 `updated_at_s` 推导；排队中、运行中、被 阻塞、失败、已取消和更新中的 Land 尝试，即使两个 Snapshot 都还不存在，也保留 它们的目标 Line。

被接纳的遗留服务端格式可能为一个 Change 保留不止一次成功的 Land，也可能在一次成功之后追加被阻塞、失败、已取消或更新中的尝试。转换保留每一行 Land 及其确切 的被接受 Patchset。只要有任何一次存活的 Land 成功过，无论重复的源 Change 状态 如何，Change 的生命周期都归一化为 landed。`land_ordinal` 最大的那次成功 Land 提供公开的目标 Line 和 land 时间；之后未成功的尝试仍然是不可变的历史，不会 撤销已完成的 land。在那次成功之后创建的 selected/current Patchset，仍然是独立 的可变 Change 权威，不必等于被接受的 Patchset。声称已 land 却没有可证明的成功 Land 的源 Change，只能遵循下文明确的重建规则或点名的省略规则；它绝不会被隐式 地补上一个伪造的 Land。

当存在确切的 Land 权威时，遗留 Change 的 `landed_at` 是一份非权威的重复投影。转换会把提供的值作为时间戳来校验，但不要求它等于任何 Land 时间，也不存储它； 权威的公开时间是最近一次成功 Land 的 `updated_at_s`。同样地，提交之后，Change 的可变 selected-Patchset 指针与某次历史 Land 确切的被接受 Patchset 是各自独立 的权威。转换保留两者，并不要求它们保持相等。

提交 Land 时通过精确的 payload 比较来解析归一化的 Line 名称，并追加和 fsync 一条完整的 Land 提交记录，其中包含必需的目标 Line 引用。对于已提交的 Land， 恢复流程会拒绝：目标 Line 为零或缺失、被引用的 Snapshot 缺失或已打墓碑，或者 状态与 Snapshot 存在与否不一致。恢复流程不会把被引用 Snapshot 的创作 Line 与 Land 的目标 Line 做比较。

紧邻的上一版 layout-1 表示，即 32 字节的 Local 或 36 字节的 Server Land 记录， 加上按 ordinal 对齐的 4 字节 `land_target_line.bin`，只能作为离线转换输入。 转换要求头部匹配且记录数完全一致，把每个确切的目标 Line 值追加到未改动的 Land 前缀之后，校验每一条加宽后的记录，省略退役的旁路文件，并在输入为零、缺失、多余、错位或有歧义时失败拒绝。源文件不被修改。

更早的、旁路文件出现之前、没有 `land_target_line.bin` 的 layout-1 generation，同样只能作为离线转换输入。转换把确切的源 `target_line` 解析为权威本地的 `line_index`，保留每一处确切存储的存活 Snapshot 引用，不比较也不重写它的创作 Line；如果在物化完整的加宽 Land 记录之前，目标或某个被引用的 Snapshot 不可得 或有歧义，则失败拒绝。

BINARY DB V0 结构

Binary DB v0：Snapshot 与内容记录

展开 Snapshot 父级沿革、Snapshot 链接、Tree、Blob、对象包、manifest 以及可选的文件缓存布局。

适用人群 存储实现者与恢复工具作者

https://ait-native.dev/technical/v1.1.1/binary-db/content/

1.1.1 文档路由 · Binary DB v0 · `layout_id = 1`

本章定义不可变的内容身份及其物理存储关系。有序的 parent edge 和归一化的 Tree
区间是权威，而下文明确点名的那几类缓存则如所述保持可重建或可选。

内容 Snapshot 记录

```
CONTENT_SNAPSHOT_RECORD_SIZE = 88

ContentSnapshotRecord - snapshot.bin:
u8  snapshot_meta
u8  history_flags
u16 payload_len
u64 payload_offset
u64 snapshot_hash48
u32 parent_snapshot_index_plus1
u32 root_tree_pack_index_plus1
u32 root_entry_ordinal
u32 line_index_plus1
u8  manifest_hash[32]
u32 file_count
u64 total_bytes
u64 created_at_s
```

规范的公开 Snapshot ID 是 `SNP-` 后跟十二个大写十六进制字符。 `snapshot_hash48` 把该后缀存放在它的低 48 位；高 16 位必须为零。非标准的 Snapshot ID 在这个 layout 中无法表示，必须拒绝。

`history_flags` 属于固定的 Snapshot schema，不是 payload。位 0 标记一个导入的 remote-head 历史边界：源 Snapshot 至少有一个父节点，但本地导入有意没有物化该 head 之前的祖先。这样的 Snapshot 不是真正的根。位 1 到位 7 保留，必须为零。

有序 Snapshot Parent 记录

```
SNAPSHOT_PARENT_EDGE_RECORD_SIZE = 12

SnapshotParentEdgeRecord - snapshot_parent_edge.bin:
u32 child_snapshot_index
u32 parent_snapshot_index
u16 parent_ordinal
u16 flags
```

当 `ContentSnapshotRecord.snapshot_meta` 位 4 置位时， `snapshot_parent_edge.bin` 中的有序行是该 Snapshot 唯一的 parent 权威。 此时固定字段 `parent_snapshot_index_plus1` 只是 ordinal 零的一份经过校验的缓存：对真正的根或导入的 remote-head 历史边界，它为零，否则必须等于 ordinal 为零那条 edge 的 `parent_snapshot_index + 1`。Snapshot 的 payload 字节绝不携带 parent 索引。

被接纳的 parent 集合最多有 1,024 行。非空集合从 ordinal 零开始，使用连续且不 重复的 ordinal。ordinal 零是显式 first-parent 遍历所用的主 parent。子节点不得 引用自身、缺失的 Snapshot、已打墓碑的 Snapshot，也不得构成环。v0 中每个 `flags` 值都是零。parent 顺序不可变，并参与 Snapshot 的 manifest 身份。

`history_flags` 位 0 置位的 Snapshot 还必须置位 `parent_edges_authority`、 满足 `parent_snapshot_index_plus1 = 0`，且没有 parent edge 行。parent 遍历和 first-parent 遍历在该 Snapshot 处停止。导出、可达性和祖先证明必须保留这个边界
区分，不得把本地被截断的图当作完整历史来报告。在源根上、在带有任何本地 parent 的 Snapshot 上、或者在没有证据表明源确有 parent 的情况下设置该标志，属于损坏， 失败拒绝。

位 4 未置位的记录以原有的固定指针作为它完整的"零或一个 parent"权威，并且没有 parent-edge 行。这个状态只是为了接纳前代 layout-1 generation 作为离线转换输入 而存在。新的写入和新激活的 generation 对每个存活的 Snapshot（包括根）都使用 edge 权威。同一个 Snapshot 上出现混合权威、payload 中的 parent 扩展、不连续的 ordinal、指针与 edge 不一致，或为位 4 未置位的子节点存在 edge，都属于损坏， 失败拒绝。

Snapshot 权威是一个 DAG 森林，不是单根树。存活 Snapshot 数为零的权威是合法的。 每个存活的非边界 Snapshot 都必须能到达一个被证明的真正的根，但不同的连通分量 可以到达不同的真正的根。前代位 4 未置位、且其完整固定指针为零的记录证明了一个 真正的根；转换写入不带 parent 行的 edge 权威，并保持 `history_flags` 为零。 记录位置、Line 归属、分量大小和当前 head 状态，都不足以允许合并分量，或把一个真正的根转成 remote-head 历史边界。

Task Snapshot Link 记录

```
TASK_SNAPSHOT_INDEX_RECORD_SIZE = 8

TaskSnapshotIndexRecord - task_snapshot_index.bin:
u32 latest_snapshot_link_index_plus1
u16 snapshot_count
u8  next_snapshot_ordinal
u8  reserved0

CHANGE_SNAPSHOT_INDEX_RECORD_SIZE = 8

ChangeSnapshotIndexRecord - change_snapshot_index.bin:
u32 latest_snapshot_link_index_plus1
u16 snapshot_count
u16 reserved0
```

```
AUTHORITATIVE_SNAPSHOT_LINK_RECORD_SIZE = 40

LocalTaskSnapshotLinkRecord / ServerTaskSnapshotLinkRecord - snapshot_link.bin:
u8  link_meta
u8  snapshot_ordinal
u16 payload_len
u64 payload_offset
u32 task_index
u32 change_index_plus1
u32 content_snapshot_index
u32 previous_task_snapshot_link_index_plus1
u32 previous_change_snapshot_link_index_plus1
u64 created_at_s
```

```
REMOTE_MIRROR_SNAPSHOT_LINK_RECORD_SIZE = 52

RemoteMirrorTaskSnapshotLinkRecord - snapshot_link.bin:
u8  link_meta
u8  remote_meta
u16 reserved1
u8  snapshot_ordinal
u8  reserved0
u16 payload_len
u64 payload_offset
u32 task_index
u32 change_index_plus1
u32 content_snapshot_index
u32 previous_task_snapshot_link_index_plus1
u32 previous_change_snapshot_link_index_plus1
u64 created_at_s
u64 fetched_at_s
```

`snapshot_ordinal` 使用完整的 `u8` 范围。值 `0..255` 渲染为 `S-01..S-256`。 写入方必须拒绝 ordinal 255 之后的下一个 link，不得回绕、截断、复用，也不得 发明扩展的 Snapshot 句柄。

Tree 记录

```
TREE_PACK_RECORD_SIZE = 32

TreePackRecord - tree_pack.bin:
u8  pack_meta
u8  pack_format_kind
u16 pack_hash_hi16
u32 pack_hash_lo32
u32 first_tree_index
u32 tree_count
u64 total_bytes
u64 created_at_s
```

```
TREE_RECORD_SIZE = 20

TreeRecord - tree.bin:
u8  tree_meta
u8  reserved0
u32 pack_entry_ordinal
u32 entry_count
u8  tree_hash80[10]
```

```
TREE_ENTRY_RANGE_RECORD_SIZE = 4

TreeEntryRangeRecord - tree_entry_range.bin:
u32 first_entry_index
```

```
TREE_ENTRY_RECORD_SIZE = 16

TreeEntryRecord - tree_entry.bin:
u8  entry_meta
u8  name_len
u16 mode_bits
u64 name_offset
u32 target_index
```

Tree entry 的名字是单个归一化的 UTF-8 路径片段，不是完整路径，且限制为 255 字节。Blob entry 指向 `blob.bin`；Tree entry 指向 `tree.bin`。

`TreeRecord.pack_entry_ordinal` 是提供该 Tree 原始 payload 的那个 Tree pack 归档内部确切的物理 entry ordinal。对于 `sparse_physical_ordinals` 未置位的 Tree Pack，其逻辑区间中的每个 Tree 都满足 `pack_entry_ordinal = tree_index - first_tree_index`，且该 ordinal 小于 `tree_count`。当该位置位时，这个 ordinal 改为确切的稀疏物理归档定位符，不必 等于逻辑偏移，也不必小于逻辑上的 `tree_count`；归档边界、解码出的 Tree ID 和 解码出的 entry 数仍然要校验。

`TreeEntryRangeRecord` 属于固定 schema，不是 payload。它的记录 ordinal 就是 `tree_index`，并且在每个稳定的激活边界上，`tree_entry_range.bin` 的记录数与 `tree.bin` 完全相同。结合所属的 `TreeRecord.entry_count`，`first_entry_index` 指明该 Tree 在 `tree_entry.bin` 中权威的、连续的归一化行。对于空 Tree，规范的 `first_entry_index` 是当前 `tree_entry.bin` 的记录数。该区间必须落在 `tree_entry.bin` 之内；新的写入按 Tree 顺序追加归一化的 entry 行，绝不与已提交 Tree 的区间重叠。

归一化区间对元数据遍历具有权威性。物理 pack 定位符对定位和校验已归档的原始 Tree payload 具有权威性。名字、mode、目标种类、目标、Tree ID 和 `entry_count` 在归一化区间与解码出的 pack entry 之间必须一致；不一致属于损坏，失败拒绝。

创建 Tree 时先追加并 fsync 它的归一化 entry 行和 range 依赖，然后才追加 Tree 的提交记录。恢复流程忽略或截断孤立的 entry/range 依赖，拒绝已提交 Tree 缺失 range 的情况，并校验每一个已提交的 range 和物理 pack 定位符。

没有 `tree_entry_range.bin` 的前代 layout-1 generation 只能作为离线转换输入。 转换保留已有的 `pack_entry_ordinal`，从 pack payload 重建并校验每一个归一化 区间，并在激活之前物化出完整的旁路文件。

Blob 与 object-pack 记录

```
OBJECT_PACK_RECORD_SIZE = 32

ObjectPackRecord - object_pack.bin:
u8  pack_meta
u8  pack_format_kind
u16 pack_hash_hi16
u32 pack_hash_lo32
u32 first_member_index
u32 member_count
u64 total_bytes
u64 created_at_s
```

```
OBJECT_PACK_MEMBER_RECORD_SIZE = 16

ObjectPackMemberRecord - object_pack_member.bin:
u8  member_meta
u8  delta_chain_depth
u16 reserved0
u32 pack_index
u32 blob_index
u32 base_blob_index_plus1
```

```
BLOB_RECORD_SIZE = 64

BlobRecord - blob.bin:
u8  blob_meta
u8  hash_kind
u16 reserved0
u64 size_bytes
u32 pack_member_index_plus1
u64 created_at_s
u64 pruned_at_s
u8  sha256[32]
```

Blob 的 SHA-256 以 32 个原始字节存储。公开的 `BLB-*` 身份由前 80 位渲染而来。 object pack 和 Tree pack 记录以 `(pack_hash_hi16, pack_hash_lo32)` 存放它们 48 位的公开 ID 后缀。

在被接纳的遗留转换中，完整 32 字节 SHA-256 相同的多条源 Blob 行，只有在 `blob_meta`、归一化后的 `hash_kind`、`reserved0`、`size_bytes` 和 `pruned_at_s` 都一致时，才会合并成一条目标 Blob。不同的 pack-member 指针是同一内容的物理 副本，每一处 member、base-Blob、Tree-entry 和重建索引的引用都会被稠密重映射。 目标的 `created_at_s` 取源创建时间中最小的合法非零值。时间为零、元数据不一致、 大小不一致、prune 不一致、哈希前缀相同但完整 SHA-256 字节不同的碰撞，或者 字节与哈希不匹配，都失败拒绝。这项归一化不会新增任何 Blob 别名行或映射文件。

可选的 manifest 与 file 缓存

这些记录是可丢弃的本地加速手段，从不具有权威性：

```
SNAPSHOT_FILE_INDEX_RECORD_SIZE = 4

SnapshotFileIndexRecord - snapshot_file_index.bin:
u32 first_file_index_plus1
```

```
SNAPSHOT_FILE_RECORD_SIZE = 24

SnapshotFileRecord - snapshot_file.bin:
u64 path_hash64
u64 path_offset
u32 tree_entry_index
u16 path_len
u16 reserved0
```

权威路径仍然是 `snapshot.bin -> tree_pack.bin/tree.bin/ tree_entry.bin -> blob.bin`。当所有可选的 manifest/file 缓存都不存在时，读取方 必须仍然正确。

BINARY DB V0 结构

Binary DB v0：Git 互操作记录

展开可选的 Git 仓库、generation、身份、commit、文件、ref、Tag 与 checkpoint 映射布局。

适用人群 Git 互操作实现者

https://ait-native.dev/technical/v1.1.1/binary-db/git/

1.1.1 文档路由 · Binary DB v0 · `layout_id = 1`

Git 记录在 AIT 内容之外构成一个可选的导入/导出映射权威；它们不取代主工作流权威或内容权威。这些 layout 在指定之处保留确切的源字节，遇到不支持的对象或身份表示时失败拒绝。

本地 Git 导入/导出记录

Git 互操作是一个可选的本地逃生舱权威。它的定长记录保留已 land 的 `git-identity-map/v1` 与 `git-interop-checkpoint/v1` 语义，同时不让 Git 成为 AIT 的主权威。v0 中每个 Git Object ID 字段都恰好是 20 个原始 SHA-1 字节。 其他 Git 对象格式会在任何 AIT 或 Git ref 变更之前失败；在 `layout_id = 1` 下， 写入方不得加宽这些字段。

```
GIT_REPOSITORY_RECORD_SIZE = 28

GitRepositoryRecord - git_repository.bin:
u8  repository_meta
u8  object_format_kind
u16 reserved0
u32 identity_len
u64 identity_offset
u8  fingerprint96[12]
```

`fingerprint96` 以 12 个原始字节存放当前 `GSR-*`、`GTR-*` 或 `ASR-*` 仓库指纹 的 24 位十六进制数字。前缀由 `repository_meta` 重建。带类型的 identity payload 存放用于计算该指纹的同一个规范源身份、目标路径或 AIT 仓库身份。

```
GIT_GENERATION_RECORD_SIZE = 20

GitGenerationRecord - git_generation.bin:
u8  generation_meta
u8  reserved0
u16 reserved1
u32 source_repository_index
u32 target_repository_index_plus1
u8  generation_hash64[8]
```

`generation_hash64` 以八个原始字节存放 `GIT-IMP-*` 或 `GIT-EXP-*` 的 16 位 十六进制数字。导入有一个 Git 源仓库，且不存储目标仓库。导出有一个 AIT 源仓库 和一个 Git 目标仓库。generation 不可变；它把由同一份导入源状态、或同一份 确定性导出计划所产生的映射归为一组。这个内容 generation 不是 Binary DB 的 事务 generation 或激活 generation。

```
GIT_IDENTITY_RECORD_SIZE = 24

GitIdentityRecord - git_identity.bin:
u8  identity_meta
u8  reserved0
i16 timezone_offset_minutes
u32 payload_len
u64 payload_offset
i64 timestamp_s
```

identity 记录保留当前的 `raw`、`name`、`email`、`timestamp` 和 `timezone` 字段。`timestamp_s` 是有符号的 Git identity 时间戳；`timezone_offset_minutes` 是解析出的数值偏移。原始 identity 字节仍然可用，作为无损的导入对象证据。

```
GIT_COMMIT_MAPPING_RECORD_SIZE = 96

GitCommitMappingRecord - git_commit_mapping.bin:
u8  mapping_meta
u8  reserved0
u16 parent_count
u32 generation_index
u32 snapshot_index
u32 author_identity_index
u32 committer_identity_index
u32 first_parent_mapping_index_plus1
u32 first_file_mapping_index_plus1
u32 file_count
u32 payload_len
u64 payload_offset
u8  git_object_id[20]
u8  git_tree_object_id[20]
u32 lfs_pointer_count
u64 created_at_s
```

被引用的 Snapshot 提供 AIT 根 Tree 定位符和文本形式的 Snapshot 身份；两者都不 在此处重复存放。`parent_count` 受 Snapshot 的 1,024 上限约束。parent 与 file 的映射区间是连续的；当且仅当对应的计数为零时，它们的 `first_*_index_plus1` 字段才为零。commit 映射记录不可变，且只追加。

```
GIT_COMMIT_PARENT_RECORD_SIZE = 28

GitCommitParentRecord - git_commit_parent.bin:
u32 commit_mapping_index
u32 snapshot_parent_edge_index
u8  parent_git_object_id[20]
```

被引用的 Snapshot parent edge 拥有 `parent_ordinal` 和 AIT 的父 Snapshot 索引。 它指向的子节点必须与所属 commit 映射的 `snapshot_index` 相同。parent 映射区间 的顺序必须等于 edge 的 ordinal 顺序，因此 Git commit 的父顺序与 Snapshot DAG 的顺序不会分歧。

```
GIT_FILE_MAPPING_RECORD_SIZE = 52

GitFileMappingRecord - git_file_mapping.bin:
u8  git_object_type_kind
u8  reserved0
u16 reserved1
u32 commit_mapping_index
u32 entry_ordinal
u32 tree_entry_index
u32 git_mode
u32 path_len
u64 path_offset
u8  git_object_id[20]
```

Tree entry 拥有 AIT mode、目标 Tree/Blob 索引和内容身份；这些值不重复存放。 `git_mode`、Git 对象类型与 Object ID，以及确切的 Git 路径字节，保留了 Git 一侧 的 tree 映射。entry ordinal 在同一个 commit 映射区间内是连续的。

```
GIT_REF_MAPPING_RECORD_SIZE = 80

GitRefMappingRecord - git_ref_mapping.bin:
u8  ref_meta
u8  git_object_type_kind
u16 reserved0
u32 generation_index
u32 snapshot_index_plus1
u32 target_index_plus1
u32 symbolic_target_ref_mapping_index_plus1
u32 ref_name_len
u64 ref_name_offset
u8  git_object_id[20]
u8  expected_previous_git_object_id[20]
u64 created_at_s
```

branch 映射把 `target_index_plus1` 用作数值 `line_index + 1`；Tag 映射把它用作 `tag_index + 1`。它们绝不存放 AIT Line ID、Line 名称、Tag 名称、`ait_kind`、 `ait_name` 或 `ait_identity`。符号式 `HEAD` 记录以数值方式引用被映射的目标 ref 记录，而不是重复存一个 Line 名称。只有当

`expected_previous_git_object_id` 的 meta 位置位时它才有意义，此时它记录导出的 compare-and-swap 证据；否则全部 20 字节为零。

```
GIT_TAG_MAPPING_RECORD_SIZE = 48

GitTagMappingRecord - git_tag_mapping.bin:
u8  tag_mapping_meta
u8  reserved0
u16 reserved1
u32 git_ref_mapping_index
u32 payload_len
u64 payload_offset
u8  peeled_commit_git_object_id[20]
u64 created_at_s
```

被引用的 Git ref 映射提供 Git ref 名称、Git 对象 ID/类型、Snapshot 索引和数值 形式的 AIT Tag 索引。轻量 Tag 的 Git 对象类型是 `commit`，原始 Tag 字节为空。 附注 Tag 的对象类型是 `tag`，并在带类型的 payload 中保留它的消息和完整的原始 Tag 对象。

```
GIT_OPERATION_CHECKPOINT_RECORD_SIZE = 44

GitOperationCheckpointRecord - git_operation_checkpoint.bin:
u8  checkpoint_meta
u8  reserved0
u16 reserved1
u32 generation_index
u8  operation_hash64[8]
u8  plan_hash96[12]
u32 next_commit_index
u32 next_ref_index
u64 updated_at_s
```

`operation_hash64` 存放 `GIO-IMPORT-*` 或 `GIO-EXPORT-*` 的 16 位十六进制 数字；`plan_hash96` 存放 `GIP-*` 或 `GEP-*` 的 24 位十六进制数字。两个 next 索引是可续跑的计划游标。checkpoint 就地更新，`checkpoint_meta` 最后写入，作为 状态提交标记。已完成的操作结果由它的 generation 和不可变映射行确定性地重建；没有任何不透明的结果 JSON 具有权威性。

映射的 `contract`、文本 `record_id`、文本形式的 AIT ID、Base64 包装，以及重复 的 JSON 数组 `parent_*_ids` 和 `file_modes` 都不存储。公开的 `GIM-*` 记录身份 仍由已 land 的 `git-identity-map/v1` 逻辑规范化推导，排除 `created_at` 和记录 ID 本身。commit、ref 和 Tag 映射的物理 ordinal 就是它们确切的追加顺序；不持久 化任何由 AIT 生成的亚秒级排序时间戳。有符号的 `GitIdentityRecord.timestamp_s` 和时区单独保留，因为它们是确切的 Git identity 数据。`identity_source`、 `published_remote_name` 以及新增的 planning-state 字段都不属于 Git 互操作。Git 导入不会伪造任何 Task 或策略沿革；现有的 `TaskRecord.task_meta` planned 位 仍然是完整的二进制 planning 状态。

BINARY DB V0 结构

Binary DB v0：服务端与策略记录

展开 Patchset、attestation、Actor、评审、策略、豁免、Plan、Repository 注册表与 Worker Job 权威。

适用人群 服务端、runner、策略与恢复实现者

https://ait-native.dev/technical/v1.1.1/binary-db/server/

1.1.1 文档路由 · Binary DB v0 · `layout_id = 1`

本章讲的是以仓库为作用域的远端工作流权威，以及独立的、以安装为作用域的 Repository 注册表。数值索引只在各自声明的根里才有意义；Worker Job 的身份和引用都留在路由到的那个 Repository 权威内部。

服务端 Patchset 记录

```
TASK_PATCHSET_INDEX_RECORD_SIZE = 8

TaskPatchsetIndexRecord - task_patchset_index.bin:
u32 latest_patchset_index_plus1
u16 patchset_count
u16 reserved0

CHANGE_PATCHSET_INDEX_RECORD_SIZE = 8

ChangePatchsetIndexRecord - change_patchset_index.bin:
u32 latest_patchset_index_plus1
u16 patchset_count
u8  next_patch_ordinal
u8  reserved0
```

```
SERVER_PATCHSET_RECORD_SIZE = 65

ServerPatchsetRecord - patchset.bin:
u8  patchset_meta
u8  patch_ordinal
u8  change_ordinal
u8  reserved0
u32 change_index
u32 previous_task_patchset_index_plus1
u32 previous_change_patchset_index_plus1
u32 base_snapshot_index
u32 revision_snapshot_index
u64 created_at_s
u64 ci_completed_at_s
u32 ci_run_seq
u16 ci_selected_suite_count
u16 ci_suite_result_count
u16 ci_blocking_failure_count
u8  ci_status_bits
u64 summary_offset
u16 summary_len
u32 ci_worker_job_index_plus1
```

前 51 字节是完整的定长 Patchset/紧凑 CI 区域：其中前 32 字节是加宽后的 Patchset 身份前缀，修正后的紧凑 CI 尾部恰好是 19 字节。这两个区域分别是 `u32-time-v0` 前身那 28 字节身份前缀和 15 字节紧凑 CI 尾部的零扩展形式。追加的 summary 定位符恰好是 10 字节。最后的 Worker Job 定位符恰好是 4 字节，合起来构成完整的 65 字节记录。这些区域 都没有隐式填充。`summary_len` 的取值范围是 `1..65535`， `summary_offset` 指向 `patchset_summary_payload.bin` 中恰好这么多个 非空 UTF-8 字节。Patchset summary 是这个可评审 Patchset 由人撰写的描述；它不是 Task intent、不是 Snapshot 消息、不是 Tree 差异统计、不是 Review 文本，也不是 Policy 输出。

`patch_ordinal` 在 `(change_index, patch_ordinal)` 上唯一。取值 `0..63` 在完整的所属 Change 身份之下渲染为 `P-01..P-64`。 `ChangePatchsetIndexRecord.next_patch_ordinal` 是已分配序号的最大值加一， 超过 64 之后就拒绝分配；删除或省略绝不会复用一个序号。
`ChangePatchsetIndexRecord.latest_patchset_index_plus1` 和 `previous_change_patchset_index_plus1` 遵循以 Change 为作用域的 Patchset 序号。

`TaskPatchsetIndexRecord` 只表示物理清单：它的 latest 指针和 每一个 `previous_task_patchset_index_plus1` 都遵循已提交的物理 Patchset 记录顺序，既不分配、也不暗示以 Task 为作用域的 Patchset 序号。
恢复过程从已提交的记录顺序重建 Task 清单，并从以 Change 为作用域的身份 重建每条 Change 链、计数和 next 序号。

Patchset 创建时先追加 summary 字节并 fsync，之后才追加 定长的 Patchset 提交记录。Patchset 的身份、属主、序号、Snapshot 引用、创建时间以及 summary 定位符/字节都是不可变的。紧凑 CI 字段和 `ci_worker_job_index_plus1` 只能在声明的 Worker Job 变更边界之下，
通过整条记录替换来改变。被打断的追加可能只留下未被引用的尾部 summary 字节，恢复时会忽略它们。已提交但缺失、为空、重叠或包含非法 UTF-8 的 summary 区间属于损坏。

`ci_completed_at_s` 是非负的 `u64` Unix 秒时间戳。 `ci_run_seq = 0` 表示还没有 CI 运行启动过。`ci_run_seq` 非零而 `ci_completed_at_s = 0` 表示一次进行中的运行，此时要求所有状态和计数字段都为零。完整的证据要求完成时间非零、 运行序列非零，且整体状态不是 `none`。

`ci_worker_job_index_plus1 = 0` 表示该 Patchset 没有选中任何由 Worker Job 支撑的 CI 运行；这包括遗留的紧凑 CI 证据和 内联 CI 运行。非零值在与该 Patchset 同属一个服务端 Repository 权威的 `worker_job.bin` 中解析为 `index + 1`。它必须指向一个 未被墓碑标记、类型恰好是 `patchset.ci` 的 Job，且该 Job 定长的 `patchset_index_plus1` 要能解析回这个确切的 Patchset。这条同根关系里不涉及任何 Repository ID 或 Repository 索引。

该定位符选中的是这个 Patchset 已提交的最大 `worker_job_index`。之后 重跑会追加一个新 Job，并用那个新的本地索引替换整条 Patchset 记录；更早的 Job 仍然是 Job 历史，但在被取代之后就不能再覆写紧凑 CI 证据。 在终态完成时，只有在重新确认那个完成的 Job 仍是定位符选中的目标之后， 才可以用 Job 结果替换紧凑 CI 字段。

```
ServerPatchsetRecord.ci_status_bits:
bits 0..1 overall_status
bits 2..3 tests_status
bits 4..5 lint_status
bits 6..7 reserved = 0

Each two-bit status:
00 none
01 pass
10 fail
11 error
```

`ci_suite_result_count` 不得超过 `ci_selected_suite_count`， `ci_blocking_failure_count` 也不得超过 `ci_suite_result_count`。各分项 状态要求整体证据已完成。这些字段只存放紧凑的 Patchset CI 证据。队列状态、尝试次数、固定的结果和重试错误 类别都留在同一个 Repository 的定长 Worker Job 家族里。lease 凭据、请求物化、详细结果、日志、制品和调度器配置都在 Binary DB v0 之外。

源端的 `author_mode`、`publish_state` 中 published/superseded 的那部分，以及 pending/非 pending 的 Policy 缓存状态，都按下文定义编码进 `patchset_meta`。 源端的 selected-for-landing 投影会按 Remote Change 指针规则归一化， 它不是 Patchset 状态。源端的 `diff_stats` 不做存储：它是通过精确比较 base 和 revision Snapshot 的 Tree 重新算出来的。除了"bin 到 bin 的转换接纳条件"一节里那个 确切具名的遗留全零闸门之外，转换器必须拒绝与该比较结果不一致的 给定值。非 pending 的 `evaluation_state` 就是最新存活的 Policy Decision 类别；Patchset 上的 pending 位用来区分新建或已失效的缓存 和一个更早的非 pending 决定。任何 Patchset JSON payload 都不具备权威性。

Patchset 存储由服务端权威负责。本地工作流权威不定义任何 Patchset 文件。

服务端 Attestation 记录

```
TASK_ATTEST_INDEX_RECORD_SIZE = 8

TaskAttestIndexRecord - task_attest_index.bin:
u32 latest_attest_index_plus1
u16 attest_count
u8  next_attest_ordinal
u8  reserved0

PATCHSET_ATTEST_INDEX_RECORD_SIZE = 8

PatchsetAttestIndexRecord - patchset_attest_index.bin:
u32 latest_attest_index_plus1
u16 attest_count
u16 reserved0
```

```
SERVER_ATTEST_RECORD_SIZE = 24

ServerAttestationRecord - attest.bin:
u8  attest_meta
u8  attest_ordinal
u8  patch_ordinal
u8  change_ordinal
u32 patchset_index
u32 previous_task_attest_index_plus1
u32 previous_patchset_attest_index_plus1
u64 created_at_s
```

Attestation 那四个有限的要求值存放在 `attest_meta` 的第 3 到第 6 位，绝不放进 payload。如果源端某条紧凑 Attestation 的时间和要求位全为零，就视为不存在，不写出 v0 记录。 存在的源端紧凑 Attestation 是以 Change 为作用域的可变源状态； 遗留格式并不会把选中 Patchset 的历史持久化到 `attested_at_s`。在持有一份一致的锁定源快照期间，转换 会通过所属 Change 权威的当前 `selected_patchset_number` 解析该行， 并写出一条 v0 Attestation，其 `patchset_index` 指向那个确切的 Patchset。序号到目标索引的查找属于 转换器本地状态，不会产生任何历史选择字段、记录、bin 或 payload。这个查找不会让当前 Change 的选择变成临时的：同一个源指针 会独立地填充上文定义的既有 `RemoteChangeRecord.selected_patchset_index_plus1` 权威。当前 selected 指针缺失、无效、不属于该属主或不唯一，都会 fail closed。源端 Attestation 的任何 author mode 都必须等于所绑定 Patchset 的 author mode，因为 v0 不会把它复制一份。

Actor 与 Review 记录

```
ACTOR_RECORD_SIZE = 36

ActorRecord - actor.bin:
u8  actor_meta
u8  reserved0
u16 payload_len
u64 payload_offset
u64 actor_key_hash
u64 created_at_s
u64 last_seen_at_s
```

对 bin 到 bin 的转换而言，这条既有的定长记录加上既有的 `ActorPayload` 就是完整的 Actor schema；不存在仅用于转换的 Actor bin 或 通用 payload。源端每一个互不相同的非空评审者身份，都会逐字节 存为 `ActorPayload.user_name_bytes`，同时 `user_id_len = 0`、 `email_len = 0`、memo 为空。转换器不会去裁剪、折叠大小写，也不会把 `Name <email>` 这类展示字符串解析成臆造的组成部分。除非有结构化的源字段能 证明是别的类别，否则 Actor 类别就是 `unknown`。 通过 `requested_groups` 提供的非空身份属于结构化的团队 证据，使用 `team` 类别。类别相同且身份字节相同的会复用同一个 Actor；字节不同则仍是不同的 Actor。

这份必填 payload 是身份权威和冲突证据，不是 可选的 Review 注解。它的长度必须放得进既有的 `u8 user_name_len` 和 `u16 payload_len`；溢出或非法 UTF-8 会 fail closed。 `actor_key_hash` 是对确切的 `user_name_bytes` 计算的 FNV-1a-64：

```
actor_key_hash = fnv1a64(user_name_bytes)
offset_basis   = 0xcbf29ce484222325
prime          = 0x00000100000001b3
```

`ActorLookupIndexRecord` 可能返回哈希冲突的候选。查找 只有在 `actor_kind` 和 `ActorPayload` 字节都精确比对通过之后才接受某个候选； 仅凭哈希绝不会合并身份。`created_at_s` 是引用该 Actor 的源端 `ReviewRecord.created_at_s` 的最小值， `last_seen_at_s` 是最大值。当被接纳的遗留源格式没有 Review 时间时，这两个字段都为零，表示历史时间未知。 提供的有效时间会在推导中保留，提供的无效时间会 fail closed；绝不会拿转换时间来顶替。

```
TASK_REVIEW_INDEX_RECORD_SIZE = 8

TaskReviewIndexRecord - task_review_index.bin:
u32 latest_review_index_plus1
u16 review_count
u16 reserved0

PATCHSET_REVIEW_INDEX_RECORD_SIZE = 8

PatchsetReviewIndexRecord - patchset_review_index.bin:
u32 latest_review_index_plus1
u16 review_count
u8  next_review_ordinal
u8  reserved0
```

```
SERVER_REVIEW_RECORD_SIZE = 40

ServerReviewRecord - review.bin:
u8  review_meta
u8  review_ordinal
u8  patch_ordinal
u8  change_ordinal
u32 actor_index_plus1
u32 patchset_index
u32 previous_task_review_index_plus1
u32 previous_patchset_review_index_plus1
u64 payload_offset
u16 payload_len
u16 reserved0
u64 created_at_s
```

Review 的各种动作变体，都由一个基础动作加上 `review_meta` 里的固定修饰位 组成。确切的源端映射如下：

```
request              -> request
comment              -> comment
task_comment         -> comment + task_lane
code_review_summary  -> comment + code_review_summary
approve              -> approve
task_approve         -> approve + task_lane
request_changes      -> request_changes
task_request_changes -> request_changes + task_lane
defer                -> comment + defer
task_defer           -> comment + task_lane + defer
dismiss              -> dismiss
```

其他任何写法，或者非法的基础动作/修饰位组合，都会 fail closed。 独立的 `blocking` 位在每一种映射中都会被保留。普通 Review 的 `actor_index_plus1` 指向它的评审者 Actor。恰好带一个 `requested_groups` 值的评审请求使用 `request` 动作，并指向对应的团队 Actor； 活跃的 v0 schema 无法在一条 Review 里编码多个组。它源端的 `reviewer` 可以缺失，也可以包含与那唯一一个非空组完全相同的 UTF-8 字节。相同的情况下它只是一份冗余的遗留投影：转换会创建或复用那一个团队 Actor 并存下一个 `actor_index_plus1`；它不会创建第二个 Actor 或 Review。如果转换器遇到不同的评审者、空身份、零个组，或一个请求里超过一个 requested group，它会 fail closed， 而不是凭空造出 Review 行或 payload。Review 的评论或请求备注 文本仍留在 `ReviewPayload` 里。

`review_ordinal` 在 `(patchset_index, review_ordinal)` 上唯一。取值 `0..63` 在完整的所属 Patchset 身份之下渲染为 `R-01..R-64`。被 接纳的、以 Change 为作用域的遗留 Review ID，在挪到该 Review 确切的 Patchset 之下时会保留其确切的数字后缀；同一个 Patchset 内允许存在空档，但不允许序号重复。`PatchsetReviewIndexRecord` 负责 next 序号分配，其 latest/previous 链接遵循以 Patchset 为作用域的 Review 序号。`TaskReviewIndexRecord` 和 `previous_task_review_index_plus1` 只保留 Task 的物理清单。 恢复过程重建每条 Patchset 链、计数和 next 序号，同时不会 给源端的 Review 身份重新编号。

Policy 与 Waiver 记录

```
TASK_POLICY_INDEX_RECORD_SIZE = 8

TaskPolicyIndexRecord - task_policy_index.bin:
u32 latest_policy_index_plus1
u16 policy_count
u16 reserved0

PATCHSET_POLICY_INDEX_RECORD_SIZE = 8

PatchsetPolicyIndexRecord - patchset_policy_index.bin:
u32 latest_policy_index_plus1
u16 policy_count
u8  next_policy_ordinal
u8  reserved0
```

```
POLICY_DECISION_RECORD_SIZE = 32

PolicyDecisionRecord - policy.bin:
u8  policy_meta
u8  policy_ordinal
u8  patch_ordinal
u8  change_ordinal
u32 patchset_index
u32 previous_task_policy_index_plus1
u32 previous_patchset_policy_index_plus1
u32 first_check_index_plus1
u16 check_count
u16 reserved0
u64 created_at_s
```

`policy_ordinal` 在 `(patchset_index, policy_ordinal)` 上唯一，而不是在 所属 Task 内唯一。取值 `0..63` 在完整的所属 Patchset 身份之下渲染为 `K-01..K-64`。`PatchsetPolicyIndexRecord.next_policy_ordinal` 是已分配序号的最大值加一，超过 64 之后就拒绝分配；删除 或墓碑标记绝不会复用一个序号。因此一个 Task 跨多个 Patchset 可以拥有超过 64 条 Policy Decision，只要不超过它 `u16 policy_count` 的 容量即可，而每个 Patchset 仍然以 64 为上限。

`PatchsetPolicyIndexRecord.latest_policy_index_plus1` 和 `previous_patchset_policy_index_plus1` 遵循以 Patchset 为作用域的 Policy 序号。 `TaskPolicyIndexRecord` 只保留完整的 Task 清单：它的 latest 指针和每一个 `previous_task_policy_index_plus1` 都遵循已提交的物理 Policy 记录顺序，既不分配、也不暗示以 Task 为作用域的 Policy 序号。 恢复过程从已提交的记录顺序重建 Task 清单，并从以 Patchset 为作用域的身份 重建每条 Patchset 链、计数和 next 序号。

```
POLICY_CHECK_RECORD_SIZE = 8

PolicyCheckRecord - policy_check.bin:
u8  check_kind
u8  check_status
u16 subject_ordinal
u32 detail_flags
```

从 `first_check_index_plus1` 开始的那些行保留源端的检查顺序， 并且是完整的定长 Policy 检查权威。`subject_ordinal` 和 `detail_flags` 的确切含义如下：

```
check_kind 0..7:
  subject_ordinal = 0
  detail_flags = 0

check_kind 8 ci_rollout_phase:
  subject_ordinal = exact rollout phase in 0..65535
  detail_flags = 0

check_kind 9 ci_patchset_suite:
  subject_ordinal = one-based position of the exact suite ID in the
                    normalized UTF-8-byte-sorted exact Policy tuple catalog
  detail_flags bit 0 = blocking suite
  detail_flags bit 1 = informational suite
  detail_flags bits 2..31 = 0
```

`ci_rollout_phase` 是 CI 强制策略的数值 rollout 阶段。它 不是 Task、Change、Patchset 或 Land 的生命周期状态，它本身也不 指代某个 CI 套件。对于每一个被接纳的遗留 phase-0 投影，源端的检查 名称恰好是 `ci_rollout_phase`；转换会写入 `check_kind = 8`，保留源端的检查状态，写入 `subject_ordinal = 0`，并写入 `detail_flags = 0`。

遗留转换只接受下列确切的 `(catalog, blocking set, informational set, phase message)` 元组。目录比较使用归一化后的 UTF-8 字节序。位于 blocking 集合中的套件获得 detail 位 0；位于 informational 集合中的套件获得 detail 位 1。

```
        catalog = [rust_core]
        blocking = [rust_core]
        informational = []
        message = CI rollout phase 0 blocks `rust_core` and keeps none visible as non-blocking surfaces.

        catalog = [full_repo_contract]
        blocking = [full_repo_contract]
        informational = []
        message = CI rollout phase 0 blocks `full_repo_contract` and keeps none visible as non-blocking
          surfaces.

        catalog = [full_repo_contract]
        blocking = [full_repo_contract]
        informational = []
        message = CI rollout phase 0 blocks `full_repo_contract` and keeps none visible as non-blocking
          surfaces. Future promotions are modeled as phase1: `full_repo`.

        catalog = [package_smoke, preflight, recent_regression, stable_smoke]
        blocking = [package_smoke, preflight, stable_smoke]
        informational = [recent_regression]
        message = CI rollout phase 0 blocks `package_smoke`, `preflight`, `stable_smoke` and keeps
          `recent_regression` visible as non-blocking surfaces.

        catalog = [package_smoke, preflight, recent_regression, stable_smoke, task_batch]
        blocking = [package_smoke, preflight, stable_smoke]
        informational = [recent_regression, task_batch]
        message = CI rollout phase 0 blocks `preflight`, `stable_smoke`, `package_smoke` and keeps
          `recent_regression`, `task_batch` visible as non-blocking surfaces. Future promotions are
          modeled as phase1: `recent_regression`, `task_batch`, phase2: `full_repo`.

        catalog = [package_smoke, preflight, recent_regression, stable_smoke, task_batch, tg1_required]
        blocking = [package_smoke, preflight, stable_smoke, tg1_required]
        informational = [recent_regression, task_batch]
        message = CI rollout phase 0 blocks `preflight`, `stable_smoke`, `package_smoke`, `tg1_required`
          and keeps `recent_regression`, `task_batch` visible as non-blocking surfaces. Future
          promotions are modeled as phase1: `recent_regression`, `task_batch`, phase2: `full_repo`.

        catalog = [package_smoke, preflight, recent_regression, stable_smoke, task_batch, tg1_required]
        blocking = [package_smoke, preflight, stable_smoke]
        informational = [recent_regression, task_batch, tg1_required]
        message = CI rollout phase 0 blocks `preflight`, `stable_smoke`, `package_smoke` and keeps
          `recent_regression`, `task_batch`, `tg1_required` visible as non-blocking surfaces. Future
          promotions are modeled as phase1: `recent_regression`, `task_batch`, phase2: `full_repo`.

        catalog = [package_smoke, preflight, recent_regression, stable_smoke, task_batch, tg1_required]
        blocking = [package_smoke, preflight, stable_smoke]
        informational = [recent_regression, task_batch, tg1_required]
        message = CI rollout phase 0 blocks `preflight`, `stable_smoke`, `package_smoke` and keeps
          `tg1_required`, `recent_regression`, `task_batch` visible as non-blocking surfaces. Future
          promotions are modeled as phase1: `recent_regression`, `task_batch`, phase2: `full_repo`.

        catalog = [package_smoke, preflight, recent_regression, stable_smoke, tg1_required]
        blocking = [package_smoke, preflight, stable_smoke]
        informational = [recent_regression, tg1_required]
        message = CI rollout phase 0 blocks `package_smoke`, `preflight`, `stable_smoke` and keeps
          `recent_regression`, `tg1_required` visible as non-blocking surfaces. Future promotions are
          modeled as phase1: `recent_regression`, `task_batch`, phase2: `full_repo`.

        catalog = [package_smoke, preflight, recent_regression, stable_smoke, tg1_required]
        blocking = [package_smoke, preflight, stable_smoke, tg1_required]
        informational = [recent_regression]
        message = CI rollout phase 0 blocks `package_smoke`, `preflight`, `stable_smoke`, `tg1_required`
          and keeps `recent_regression` visible as non-blocking surfaces. Future promotions are modeled
          as phase1: `recent_regression`, `task_batch`, phase2: `full_repo`.
```

阶段检查、确切的套件名检查、目录、blocking/informational
划分以及消息，必须与其中某一个元组一致。标签属于推导出来的 展示内容，不做持久化。不同的或有歧义的表示会 fail
closed；转换器绝不会从自由文本里解析出阶段编号或套件分配。 原生 v0 写入方只有在显式的 Policy
配置直接提供了这些定长字段时，才可以使用 另一个数值阶段。

对 `check_kind = 9`，两个套件 detail 位中恰好置位一个。在 同一条 Policy 内，套件序号非零、唯一、完整，且不超过 该
Policy 确切元组目录的长度。它们并不受所属 Patchset 的 `ci_selected_suite_count` 约束：Patchset
那个字段是某一次 CI 运行的紧凑 证据，而不可变的 Policy 行保留的是历史评估结果，这些结果可能早于
那份证据，也可能使用了另一种被接纳的 blocking/informational 投影。转换会把源端的
`ci_patchset_suite_<suite-id>` 名称对照确切的 Policy 元组目录做解析； 套件身份缺失、重复、不完整或有歧义都会
fail closed。 Policy 检查的 `label` 和 `message` 是展示投影，不做 存储。未知的检查名称/状态会 fail
closed，而不是变成文本 payload。源端的 `input_fingerprint` 是缓存元数据，不是 Policy 权威； v0 会从定长的
Patchset、Attestation、Review、Waiver、CI 和 Policy 输入重新计算它，并不持久化源端的缓存键。

```
TASK_WAIVER_INDEX_RECORD_SIZE = 8

TaskWaiverIndexRecord - task_waiver_index.bin:
u32 latest_waiver_index_plus1
u16 waiver_count
u8  next_waiver_ordinal
u8  reserved0

PATCHSET_WAIVER_INDEX_RECORD_SIZE = 8

PatchsetWaiverIndexRecord - patchset_waiver_index.bin:
u32 latest_waiver_index_plus1
u16 waiver_count
u16 reserved0
```

```
WAIVER_RECORD_SIZE = 44

WaiverRecord - waiver.bin:
u8  waiver_meta
u8  waiver_ordinal
u8  patch_ordinal
u8  change_ordinal
u32 patchset_index
u32 previous_task_waiver_index_plus1
u32 previous_patchset_waiver_index_plus1
u64 payload_offset
u16 payload_len
u16 rule_code
u64 created_at_s
u64 expires_at_s
```

Plan 记录

```
PLAN_RECORD_SIZE = 48

PlanRecord - plan.bin:
u8  plan_meta
u8  reserved0
u16 payload_len
u64 payload_offset
u32 latest_revision_index_plus1
u32 published_plan_index_plus1
u32 published_latest_revision_index_plus1
u64 created_at_s
u64 updated_at_s
u64 published_at_s
```

```
PLAN_REVISION_RECORD_SIZE = 56

PlanRevisionRecord - plan_revision.bin:
u8  revision_meta
u8  reserved0
u16 payload_len
u16 revision_number
u16 item_count
u64 payload_offset
u32 plan_index
u32 previous_revision_index_plus1
u32 item_start_index
u32 published_revision_index_plus1
u32 root_tree_pack_index_plus1
u32 root_entry_ordinal
u64 created_at_s
u64 published_at_s
```

```
PLAN_ITEM_RECORD_SIZE = 16

PlanItemRecord - plan_item.bin:
u8  item_meta
u8  reserved0
u16 payload_len
u64 payload_offset
u32 line_number
```

Plan 的 revision 历史是一条线性链：

```
PlanRecord.latest_revision_index_plus1
  -> PlanRevisionRecord.previous_revision_index_plus1
  -> ...
  -> 0
```

Task 到 Plan 的绑定使用 Task 记录里数值形式的 revision 和 item 索引。 要确定是否真的可开成 Task，仍然需要与 `PlanItemPayload.plan_item_ref_bytes` 做比较。

服务端全局 Repository 注册表权威

服务端全局的 Repository 注册表是一个以安装为作用域的 Binary DB 根。它绝不会嵌套在某个仓库权威内部，也绝不会被复制 进本地仓库。明确划归这个根的每一个文件，都使用 全局的四字节 `layout_id = 1` 头部。必需但为空的家族就是一个 只有头部的文件。这个根里的物理索引和另一个权威 根里数值相同的索引之间没有任何关系。

下文每一个 `repository_index` 都是直接指向这个根的 `repository.bin` 的稠密索引，也是该 Repository 唯一的主键。索引零是 合法的。Repository 记录只追加：已分配的索引绝不会被重新编号、移除、 复用或转移给另一个 Repository。`*_index_plus1` 使用全局的"缺省"编码。这个根绝不持久化任何指向仓库工作流、Worker Job 或共享内容权威的数值索引。

前四个 Repository 索引是固定的：

```
0 ait-core
1 ait-server
2 ait-python
3 ait-node
```

之后每一个 Repository 都从 4 开始，按下一个记录序号追加。 所有服务端路由、配置和运行期关系，都通过 `repository_index` 标识一个 Repository，绝不通过 Repository 名称。

活跃的服务端全局定长家族清单恰好只有 `repository.bin`。 它活跃的带类型 payload 清单恰好只有 `repository_payload.bin`， 它唯一活跃的索引是 `repository_namespace.idx`。 `operational_public_sequence.bin`、任何 Worker Job 定长文件，以及任何 Worker Job 索引，在这个全局根里都是禁止的。Worker Job 文件必须按下文声明的方式， 存在于每一个数值形式的服务端 Repository 权威中。任何 其他的运行期 `.bin`、payload 或 `.idx` 文件都会导致激活失败。

运行期时间戳是非负的 `u64` Unix 秒。必填的 时间戳非零。可选的时间戳在缺省时为零，否则 非零。被接纳的 RFC 3339 时间戳会 归一化为 UTC；对非负值而言，小数秒会在整秒值做范围检查并 存储之前被有意丢弃。早于 Unix 纪元的值、大于 `u64::MAX` 的 整秒值，或无法解析的文本，都会 fail closed。Repository 和 Job 的 `updated_at_s` 不早于各自的

`created_at_s`。当某个 Job 被加锁时，`created_at_s <= locked_at_s <= updated_at_s`。

Repository 注册表记录

注册表是以安装为作用域的权威，负责 Repository 身份和 路由元数据。它独立于每一个以仓库为作用域的工作流和内容根。

```
        OPERATIONAL_REPOSITORY_RECORD_SIZE = 33

        OperationalRepositoryRecord - repository.bin:
        u8  repository_meta
        u8  lifecycle_kind
        u8  namespace_ascii[2]
        u8  policy_flags
        u32 payload_len
        u64 payload_offset
        u64 created_at_s
        u64 updated_at_s
```

Repository 的主键是该记录的物理 `repository_index`；它 不会在记录或 payload 内部再复制一份。Repository 名称是非空、 不可变的 UTF-8 展示元数据，可以无限制地重复。名称绝不会被 当作身份或路由权威。非空的 `namespace_ascii` 值 在 `active` 和 `retiring` 记录之间唯一；为空是合法的，且没有 命名空间索引行。历史上的 `purged` 身份可能与后来某个存活的 Repository 共用一个命名空间，因此命名空间查找可能返回多个候选，必须校验生命周期加上那确切的两个定长字节。`repo_name` 和 `created_at_s` 在创建之后不可变。策略、命名空间、生命周期和 `updated_at_s` 只能在注册表锁之下通过整条记录替换来变更。

已激活的根至少有四条 Repository 记录。第零到第 三条记录的 payload 名称分别恰好是 `ait-core`、`ait-server`、`ait-python` 和 `ait-node`，且绝不会被墓碑标记；退役改用保留的 `lifecycle_kind = purged` 记录来表示。后来的记录可以重复其中任何一个 名称，但不会因此获得那条保留记录的身份。

逻辑上的 Repository 默认 Line 始终是确切的 UTF-8 `main`。它是 schema 常量，没有对应的定长或 payload 字段，会在暴露默认 Line 值的 API 边界上合成出来。原生创建流程无法选择 别的默认 Line。

Repository 权威目录路由

配置好的服务端 Repository 权威父目录，用规范的无符号 十进制 `repository_index` 作为每个 Repository 的目录名：

```
        <server-repository-authority-parent>/
          0/  # ait-core
          1/  # ait-server
          2/  # ait-python
          3/  # ait-node
          4/
          ...
```

零恰好写作 `0`；其他每一个目录名都不带前导零、 符号、空白、后缀、前缀或其他数字写法。Repository 名称绝不出现在权威目录名中。该目录是那个 Repository 的 工作流、内容、Plan 和 Worker Job 权威家族的路由容器。目录 本身不增加任何 `.bin` 记录；Worker Job 子布局在下文声明。

活跃的父目录里，为每一条未被墓碑标记的 Repository 记录恰好包含一个真实的、非符号链接的目录，且没有任何文本别名目录。解析一个目录时，会把它的目录名 按 `u32` 解析，核实对应的 `repository.bin` 记录存在且未被墓碑标记，绝不按 Repository 名称扫描。压实过程保留每一个 Repository 目录名，因为它保留了每一个 `repository_index`。

以 Repository 为作用域的 Worker Job 记录

每一个规范的数值形式服务端 Repository 权威，都恰好包含一个 `worker_job.bin` 运行期队列权威。必需但为空的文件就是只有头部的文件。这个文件是那一个 Repository 的 Remote Binary 输入：它不会被复制进本地 Repository，不是被排除在外的通用 `job.bin`，不是 Patchset 的紧凑 CI 证据，也不是队列投影。活跃的 v0 不定义任何 Worker Job payload 文件。

```
SERVER_WORKER_JOB_RECORD_SIZE = 52

ServerWorkerJobRecord - worker_job.bin:
u8  job_meta
u8  job_kind
u8  state_kind
u8  outcome_kind
u16 attempt_count
u16 max_attempts
u16 error_kind
u16 reserved0
u32 patchset_index_plus1
u32 snapshot_index_plus1
u64 available_at_s
u64 locked_at_s
u64 created_at_s
u64 updated_at_s
```

所属的 Repository 隐含在规范的数值权威 目录里。Repository ID 和 `repository_index` 都不会在定长记录里复制一份。物理上直接的 `worker_job_index` 记录序号，就是那个 Repository 内部 Worker Job 唯一的主键。记录只追加： 已分配的索引绝不会被重新编号、移除、复用或转移。 墓碑标记会保留它确切的槽位。

完整的安装级身份是 `(repository_index, worker_job_index)`。单独一个 `worker_job_index` 在它所路由到的 Repository 权威之外没有任何 含义，且 v0 不存储任何独立的公开或全局 Job ID。`attempt_count <= max_attempts`；两者都必须放得进 `u16`， 且 `max_attempts` 非零。`job_kind` 是定长权威；API 的 `job_type` 字符串是由它合成的，绝不会作为 Job 字节持久化。 `available_at_s`、`created_at_s` 和 `updated_at_s` 是必填的。只有 `running` 才有非零的 `locked_at_s`；其他每种状态都是 `locked_at_s = 0`。

那两个定长的领域引用字段有下面这套确切的、按 Job kind 区分的解释。 `+1` 保留零表示缺省。每一个非零引用都在 与该 Job 相同的数值 Repository 权威内部解析：

job_kind	API job_type	patchset_index_plus1	snapshot_index_plus1
2	content.gc	零	零
3	content.optimize	零	零
4	content.pack	零	零
5	land.process	所属 Patchset	零
6	main-seed.refresh	所属 Patchset	先前 Snapshot 或零
7	patchset.ci	所属 Patchset	零

job_kind	API job_type	patchset_index_plus1	snapshot_index_plus1
8	patchset.ci.aggregate	所属 Patchset	零
9	policy.evaluate	所属 Patchset	零
10	reconcile.repo	零	零
11	repo.ci	零	选中的 Snapshot

`job_kind = 1` 在活跃的 v0 中未分配。`agent.turn.submit` 需要一份 不透明的 turn 请求，它无法由既有的 Repository 领域权威表示， 因此在有带类型的 agent-turn schema 之前，它不是持久化的 Worker Job。其他任何 `job_kind`、本表未分配却非零的字段、家族错误的目标、跨 Repository 的目标，或 被墓碑标记的目标，都会 fail closed。 `land.process` 从解析出的 Patchset 推导它的 Change 和 revision Snapshot。Job 在任何 Land 存在之前就已提交；执行时会针对逻辑上的 `main`、以固定的服务端默认 `direct` 模式创建 Land。 Patchset 相关的 Change 和 Snapshot 数据在其余情况下都从解析出的 Patchset 闭包推导。`main-seed.refresh` 使用逻辑上的 `main`，且只冻结一个独立要求的先前 Snapshot。原生 `repo.ci` 会在提交 Job 之前冻结它选中的 Snapshot，同样在逻辑上的 `main` 上操作；这两种 kind 都不会持久化 Line 索引，也不会把 Snapshot 身份推迟给一个可变的 Repository head。

不存在变长的 Worker Job 请求或输入字节。`job_kind` 和那两个 定长引用就是完整的持久化执行选择器。只有当每一个影响执行的选择都已经提交到那些字段、或提交到被引用的既有领域权威之后， 写入方才可以提交一个 Job。触发标签、传输标签、调度器资源键、优先级、重试延迟、 runner 上下文和物化出来的运行期 payload，都是重建出来的运行期数据，不能改变那份持久化的选择。内容维护和 Repository 对账这两种 kind 会调用它们那一个由服务端拥有的 kind 级操作，不带任何按 Job 的覆盖参数。Patchset CI 和聚合从 选中的 Patchset 及其经过校验的 Policy/CI 权威推导出套件

闭包。Repository CI 使用服务端拥有的 Repository CI 契约，在选中的 Snapshot 和逻辑上的 `main` 上操作。如果某个被请求的操作 无法在这些规则之下被精确重建，入队会在 分配 `worker_job_index` 之前失败。

运行期 Repository payload 文件

服务端全局根和每一个服务端 Repository 权威都没有共享的 字符串池，也没有 `operational_payload.bin`。Repository 注册表的定位符 指向全局的 `repository_payload.bin`。Worker Job 没有 payload 定位符，也没有 payload 文件。

```
        OperationalRepositoryPayload - repository_payload.bin:
        u16 repo_name_len
        u8  repo_name_bytes[repo_name_len]
```

Repository 的 `payload_len` 非零，`payload_offset` 位于 `BIN_HEADER_SIZE` 或其之后；它恰好是 `2 + repo_name_len`，没有填充也没有 尾部字段。Repository 名称是精确合法的 UTF-8。完整的 Repository payload 最多 65,537 字节。

可重建的注册表与 Worker Job 索引

本小节里的所有索引都是可选的可重建加速器。一个文件 使用的权威根和目标家族，恰好由它的文件名隐含决定。`repository_namespace.idx` 属于服务端全局注册表。那两个 Worker Job 索引分别属于每一个数值形式的服务端 Repository 权威。持久化的行按声明顺序对所有字段排序。

```
        OPERATIONAL_NAMESPACE_INDEX_RECORD_SIZE = 8

        OperationalNamespaceIndexRecord - repository_namespace.idx:
        u8  namespace_ascii[2]
        u16 reserved0
        u32 repository_index_plus1
```

空命名空间字节 `[0x00, 0x00]` 没有索引行。非空且字节精确匹配的 命名空间候选，可能包含多个历史 `purged` 身份，但最多只有一个 `active` 或 `retiring` 身份。

```
        SERVER_WORKER_READY_INDEX_RECORD_SIZE = 12

        ServerWorkerReadyIndexRecord - worker_ready.idx:
        u64 available_at_s
        u32 worker_job_index_plus1

        SERVER_WORKER_STATE_INDEX_RECORD_SIZE = 8

        ServerWorkerStateIndexRecord - worker_state.idx:
        u8  state_kind
        u8  reserved0
        u16 reserved1
        u32 worker_job_index_plus1
```

`worker_ready.idx` 只包含存活的 `queued` 行。队列认领会在查找之后校验 本地 Job 索引、权威状态、确切的 `available_at_s`、尝试 预算，以及不存在存活的运行期 lease。`worker_state.idx` 包含每一个存活的 Job。在这两个文件里，`worker_job_index_plus1` 只在 持有该索引的同一个 Repository 权威内部解析。

不会为被墓碑标记的记录写 `.idx` 行。缺失、陈旧、重复 或损坏的索引会被丢弃，并从该权威的 `worker_job.bin` 重建。索引绝不会修复或覆盖定长记录。

安装范围的调度器会从注册表枚举 `active` 和 `retiring` 的 Repository 索引，从各个本地 `worker_ready.idx` 读取经过精确校验的候选，并在内存里按 `(available_at_s, repository_index, worker_job_index)` 归并它们。那次内存归并 是一次性的，绝不会作为全局 Job 索引、序列或 身份权威持久化。

运行期元数据与状态编码

除非下文另有覆盖，可变的运行期 `*_meta` 字节使用：

```
        bits 0..6 reserved = 0
        bit 7 tombstoned
```

被墓碑标记的记录属于保留下来的历史，不会出现在索引中，也不能 成为存活关系的目标。源转换不会写入任何墓碑标记。

Repository 元数据是：

```
repository_meta:
bits 0..6 reserved = 0
bit 7 tombstoned

lifecycle_kind:
1 active
2 retiring
3 purged

namespace_ascii[2]:
[0x00, 0x00] empty
[x,    0x00] one-byte namespace
[x,    y   ] two-byte namespace

policy_flags:
bit 0 require_attestation
bit 1 require_tests
bit 2 require_lint
bit 3 require_security_scan
bit 4 require_license_scan
bit 5 require_ai_provenance
bit 6 require_code_review_summary
bit 7 docs_only_relaxed_checks
```

每个 x 或 y 都是一个非零的 ASCII 字母数字、_ 或 - 字节。 `[0x00, y]`、内嵌的零字节、控制/非 ASCII 字节，以及长度超过两字节的输入，都是非法的。大小写和字节都是精确的；读取方不做任何 裁剪、折叠或 Unicode 归一化。逻辑上的命名空间长度 由尾部的零推导得出，绝不存储。

第 0 到第 6 位是该 Repository 的默认要求。当第 7 位置位、 且有效的内容类别是 `docs_only` 时，测试、lint、安全扫描和许可证扫描全都不做要求；attestation、AI 溯源和代码评审 摘要保持它们的默认位。逻辑上的策略 ID 和版本固定 为 `prototype` 和 `1`，不做持久化。暴露 Repository 策略 JSON 的 API 会从 `policy_flags` 合成出规范的逻辑对象。

每一个未被墓碑标记的 Repository 都要求有非空的名称、一个合法的 `namespace_ascii[2]` 值，以及一个 `policy_flags` 字节。名称重复是 合法的。 `active -> retiring -> purged` 和 `retiring -> active` 是仅有的生命周期 转换。`purged` 的 Repository 没有存活的 Worker Job，也不能成为 新 Job 的属主。已 purge 的身份、Repository payload 字节，以及保留下来的 Repository 本地 Job 历史，在离线压实之前都保持精确不变；它们绝不会从另一个领域重建出来。

Worker Job 状态是：

```
state_kind:
1 queued
2 running
3 succeeded
4 failed

outcome_kind:
0 none
1 completed
2 skipped
3 attached
4 superseded
5 failed

error_kind:
0 none
1 retryable_execution
2 terminal_execution
3 lease_expired
```

每一个存活的 Job 都存放在某个 `active` 或 `retiring` Repository 的数值权威 目录之下。`queued` 和 `running` 要求 `outcome_kind = none`；`succeeded` 要求 `completed`、`skipped`、`attached` 或 `superseded`；`failed` 要求 `outcome_kind = failed`。新入队的 Job 是 `error_kind = none`。排队重试或运行中重试可以保留 `retryable_execution` 或 `lease_expired`。`succeeded` 要求 `error_kind = none`；`failed` 要求 `terminal_execution` 或 `lease_expired`。

`attached` 和 `superseded` 只保留"这个 Job 为什么没有独立完成 成功工作"这一信息。它们不标识另一个 Job。去重时用到的任何 等价活跃 Job 或后来成功 Job 的身份，都属于运行期/投影 数据，不是 Worker Job 权威。

lease 凭据不是 Binary DB 权威。认领时服务端会生成一个 密码学随机的 16 字节 `lease_token`，把存活条目保存在内存中，并把它镜像到一个用于崩溃恢复的临时 Binary，该 Binary 位于所有 已激活的全局或 Repository

权威根之外。运行期条目以 `(repository_index, worker_job_index, attempt_count)` 为键，携带令牌、最近一次心跳时间、过期时间和撕裂写检测。只有当定长的 Job 处于 `running`、其 `attempt_count` 精确匹配、出示的令牌精确匹配，且运行期过期时间尚未到达时，它才有效。

那个临时 Binary 是一次性的运行期副本，不是 layout-1 的文件 家族。它被排除在权威清单、转换输出、Remote Binary 传输、离线压实和领域恢复之外。它不能把 排队中的 Job 变成运行中、不能修复 `worker_job.bin`，也不能提供 Job 身份。条目缺失、 损坏、陈旧、不匹配或过期，都会让 lease 失效；在 Repository 队列锁之下，服务端会按其尝试预算把该 Job 重新入队或判定为 终态失败，清除 `locked_at_s`，并记录 `lease_expired`。只有在运行中的 Job 记录和运行期镜像都已持久化之后， 认领才会返回令牌。心跳、完成和失败在持有队列锁期间都要求 同一个令牌。Worker 身份属于诊断用的运行期 数据，既不存进 Worker Job 权威，也不存进 lease 令牌。

这里未分配的每一个数值状态、结果或错误值都是保留的， 会 fail closed。完整的成功结果对象归属于它们所变更的领域记录。 完整的失败消息归属于被排除在外的追踪/日志或诊断投影边界。两者都不是 Worker Job 权威。

运行期变更与恢复边界

这些全局和以 Repository 为作用域的运行期家族，不增加任何权威的 WAL、journal、事务 ID 或通用 generation 记录。锁文件、 临时替换文件和 fsync 记账都属于运行期文件系统 元数据，不是 `.bin` 权威。如果一次变更需要多个领域锁，写入方先获取服务端全局注册表锁，然后按 `repository_index` 升序获取各个 Repository 本地队列锁，并按相反顺序释放：

```
<server-global-registry-root>/01-registry.lock
<server-repository-authority-parent>/<repository_index>/worker-queue.lock
```

获取锁时会拒绝符号链接，也会拒绝归属于另一个已激活根的锁。 需要检查或墓碑标记 Job 的 Repository 生命周期转换，会按声明的顺序获取 注册表锁和它自己的 Repository 队列锁。Job 变更只获取其所属 Repository 的队列锁。那把本地锁负责给 Worker Job 定长家族、非权威的运行期 lease 副本、两个本地 Job 索引，以及每一次 Patchset `ci_worker_job_index_plus1` 或紧凑 CI 替换排序。读取方在解析定长记录期间持有对应的共享锁；写入方则一直独占持有到领域提交点。

对于追加式创建，会先追加完整的定长明细记录并 fsync。 定长记录就是提交点。可重建索引最后写。已提交记录的物理序号成为它永久的 `worker_job_index`；不存在独立的分配器、序列头、预留记录、 payload 追加或可复用的未提交身份。

覆写沿用既有的 v0 事务层契约：把 完整的前像保留在非权威的恢复存储中，写入一条完整的定长记录（保留字节置零），fsync，然后丢弃前像。
除非那次替换抵达了它的领域提交点，否则恢复会还原完整的前像。同一条定长记录内部的字段绝不会被独立 观测到。

其他的领域提交规则是：

- Repository 创建会在下一个从未使用过的 `repository_index` 上暂存并 fsync 真实的数值权威目录，然后追加 Repository 记录 并在 `repository.bin` 上提交。被打断的、提交前的目录属于不可达的暂存内容，绝不会按名称被激活。策略标志位、命名空间字节、生命周期和更新时间的变更，都通过替换完整的 Repository 记录来提交。
- Job 创建从已提交记录的完整计数推导出下一个 `worker_job_index`， 并在所属数值 Repository 权威内部的 `worker_job.bin` 上提交。 非 `patchset.ci` 的 Job 到此就完成了。对于 `patchset.ci`，同一次持有队列锁的变更接着会替换完整的 所属 Patchset 记录，以选中那个新的 `worker_job_index`；那次 Patchset 替换就是关系的提交点。在此之前被打断，可能 留下一个合法但未被选中的 Job，它无法提供紧凑 CI 证据。
- 认领会递增 `attempt_count`，创建新的运行期 lease 条目， 写入 `state_kind = running` 和 `locked_at_s`，并通过替换完整的 Job 记录来提交权威。提交前的运行期条目会被忽略，除非定长的状态和尝试计数与它精确匹配。已提交的 运行中 Job 若缺少其有效的运行期条目，即为 lease 丢失，会按尝试规则重新入队或 判定失败。只有在两次写入都完成 fsync 之后才会返回令牌。心跳会精确比对令牌和尝试计数，持久化 更新后的运行期心跳/过期时间，并用同一个更新后的 `locked_at_s` 替换完整的定长 Job 记录。失败会让运行期条目失效、 清除 `locked_at_s`、记录定长的错误类别，并选择 重新入队或写入终态失败结果。
- 一次成功的执行会先把结果提交到既有的所属领域 权威，然后再把 Job 替换为 `state_kind = succeeded`、写入其定长结果和 `locked_at_s = 0`，最后让运行期 lease 条目失效。因此被选中的 `patchset.ci` 紧凑证据，是在那个 Job 仍处于选中状态、 且 Job 成功标记尚未写入之前，通过整条 Patchset 替换提交的。领域优先的完成过程若被打断，会以幂等方式重试。`attached` 和 `superseded` 只提交那个终态结果；

不会持久化第二个 Job 身份。陈旧的运行期条目无法让 终态 Job 复活。队列索引和调度视图都是可修复的。

- Repository 转为 `purged` 时，会先在保留紧凑 CI 证据的前提下，通过整条记录替换清除每一个 Patchset 的 `ci_worker_job_index_plus1`，然后墓碑标记它拥有的每一个存活 Job，最后 提交完整的 Repository 生命周期替换。在最后那次替换之前被打断， 会同时还原 Patchset 和 Job 的前像。

恢复过程首先校验头部、记录的精确整除关系、保留位为零、 那四个固定的 Repository 槽位、Repository 的 UTF-8、命名空间字节、策略 标志位、时间戳、结果、错误，以及所有定长引用不变式。它会 在打开每一个以仓库为作用域的 Worker Job 权威之前，精确校验数值 Repository 目录闭包。随后它会构建完整的源索引图，并拒绝任何指向不存在、已被墓碑标记或属主错误 记录的存活引用；拒绝互相冲突的存活非空命名空间；也拒绝任何非零的 Patchset Job 定位符--如果它无法精确校验出一个同根、`job_kind = 7`、其 `patchset_index_plus1` 选中该 Patchset 的 Job。重复的 Repository 名称， 以及不同 Repository 根中数值相同的本地 Worker Job 索引，绝不构成 恢复冲突。

只有在权威校验通过之后，恢复才可以重建注册表命名空间 索引和各个 Repository 的本地 Job 索引。可变的 Repository 元数据、 Patchset 选择和 Job 状态，绝不会从后面的行或某个 索引推断出来。随后恢复会加载运行期 lease 副本；副本缺失就当作 空副本。每个条目都必须与某一个运行中的 Job 及其尝试 计数精确匹配，且未过期；否则该条目会被丢弃，且该 Job 按 lease 丢失规则处理。 不完整的尾部定长记录属于损坏，除非事务层 能证明它是唯一一次被打断的追加。中间位置的畸形字节绝不会 被截断修复。

运行期容量与压实

每一个定长家族和可重建索引，其存活目标索引都以 `u32::MAX - 1` 为上限，因为非零的 `*_index_plus1` 必须保持可表示。 这个定长家族上限对每个 Repository 里的 `worker_job.bin` 独立生效； 不存在任何持久化的全局 Job 家族，会额外施加一个 安装范围的 Job 数量上限。由于 Worker Job 的墓碑标记会保留其 主键槽位，它的容量检查用的是已提交记录总数，而不是存活 Job 数。直接的 `u32` 源值和计数在写入前都会预检。Repository 的 payload 偏移和文件大小必须放得进 `u64`；每一处 `offset + length` 计算都使用带检查的算术。

离线压实会写出一个新的、非激活的注册表根和 Repository generation，保留每一个确切的 `repository_index` 和 `worker_job_index`、Repository 名称、命名空间字节、策略标志位、状态、 kind、定长引用、结果、错误和时间。它绝不会重排、 移除或重新编号 `repository.bin` 或 `worker_job.bin` 的记录； Repository 和 Worker Job 的墓碑标记都保留其槽位，之后的任何身份 都不得复用它们。其他物理家族只有在改写完它们完整的引用和索引闭包之后，才可以被稠密重编号。压实会重建 那两个本地 Job 索引，同时不改变 Job 记录顺序和 Patchset 定位符。 它不会读取、复制或创建那个临时的运行期 lease 副本， 并且会在原子激活之前校验完整的目标。

改变定长宽度、状态分配、payload 语法、文件家族、 主键规则、转换源边界或容量，都需要一次新的 布局转换，以及兼容的读写方。`layout_id` 没有变化 并不构成做出这些改变的许可。

BINARY DB V0 结构

Binary DB v0：payload、索引与编码

展开每一类带类型的 payload、可重建索引记录、元数据位分配、enum 以及保留值规则。

适用人群 编解码、校验与迁移实现者

https://ait-native.dev/technical/v1.1.1/binary-db/physical/

1.1.1 文档路由 · Binary DB v0 · `layout_id = 1`

定长记录经常指向带类型的 payload 文件，并使用紧凑的元数据字段。本章给出完整的字节布局、查找记录、位分配、枚举和校验规则，让你不必自行发明通用字符串或 payload 格式就能解码这些值。

带类型的 Payload 文件

本格式没有定义共享的 `strings.bin`、`strings.idx` 或 `strings.hash`。 变长字节由某个带类型的领域 payload 文件持有。

```
TaskPayload - task_payload.bin:
u16 title_len
u8  title_bytes[title_len]
u8  intent_bytes[payload_len - 2 - title_len]

ChangePayload - change_payload.bin:
u8  title_bytes[payload_len]

PatchsetSummaryPayload - patchset_summary_payload.bin:
u8  summary_bytes[summary_len]

SnapshotPayload - snapshot_payload.bin:
u16 message_len
u8  message_bytes[message_len]
u8  line_name_bytes[payload_len - 2 - message_len]

TagPayload - tag_payload.bin:
u16 tag_name_len
u8  tag_name_bytes[tag_name_len]
u8  message_bytes[payload_len - 2 - tag_name_len]
```

```
SnapshotLinkPayload - snapshot_link_payload.bin:
u16 worktree_name_len
u16 line_name_len
u16 task_id_len
u16 change_id_len
u16 author_mode_len
u8  worktree_name_bytes[worktree_name_len]
u8  line_name_bytes[line_name_len]
u8  task_id_bytes[task_id_len]
u8  change_id_bytes[change_id_len]
u8  author_mode_bytes[author_mode_len]
u8  model_name_bytes[
      payload_len - 10 - worktree_name_len - line_name_len - task_id_len
      - change_id_len - author_mode_len
    ]
```

```
ActorPayload - actor_payload.bin:
u8  user_name_len
u8  user_id_len
u8  email_len
u8  user_name_bytes[user_name_len]
u8  user_id_bytes[user_id_len]
u8  email_bytes[email_len]
u8  memo_bytes[payload_len - 3 - user_name_len - user_id_len - email_len]

ReviewPayload - review_payload.bin:
u8  message_bytes[payload_len]

WaiverPayload - waiver_payload.bin:
u8  reason_bytes[payload_len]

PlanPayload - plan_payload.bin:
u8  title_bytes[payload_len]
```

```
PlanRevisionPayload - plan_revision_payload.bin:
u16 title_snapshot_len
u16 summary_len
u16 artifact_path_len
u16 artifact_selector_len
u16 artifact_heading_len
u8  title_snapshot_bytes[title_snapshot_len]
u8  summary_bytes[summary_len]
u8  artifact_path_bytes[artifact_path_len]
u8  artifact_selector_bytes[artifact_selector_len]
u8  artifact_heading_bytes[artifact_heading_len]
u8  artifact_blob_id_bytes[
      payload_len - 10 - title_snapshot_len - summary_len - artifact_path_len
      - artifact_selector_len - artifact_heading_len
    ]
```

```
PlanItemPayload - plan_item_payload.bin:
u16 plan_item_ref_len
u16 text_len
u8  plan_item_ref_bytes[plan_item_ref_len]
u8  text_bytes[text_len]
u8  heading_path_bytes[payload_len - 4 - plan_item_ref_len - text_len]
```

```
GitRepositoryPayload - git_repository_payload.bin:
u8  identity_bytes[identity_len]

GitIdentityPayload - git_identity_payload.bin:
u32 raw_len
u32 name_len
u8  raw_identity_bytes[raw_len]
u8  name_bytes[name_len]
u8  email_bytes[payload_len - 8 - raw_len - name_len]

GitCommitMappingPayload - git_commit_mapping_payload.bin:
u32 message_len
u8  message_bytes[message_len]
u8  raw_commit_bytes[payload_len - 4 - message_len]

GitFileMappingPayload - git_file_mapping_payload.bin:
u8  path_bytes[path_len]

GitRefMappingPayload - git_ref_mapping_payload.bin:
u8  ref_name_bytes[ref_name_len]

GitTagMappingPayload - git_tag_mapping_payload.bin:
u32 message_len
u8  message_bytes[message_len]
u8  raw_tag_bytes[payload_len - 4 - message_len]
```

`heading_path_bytes` 需要一种由 schema 定义的稳定编码。布局 v0 没有
选定新的规范表示。因此写入方必须保留已被接纳的 layout-1 编码， 否则就要 fail closed。

单字段 payload 不会重复内层长度。它们的长度来自所属记录的 `payload_len`，只有 Patchset summary 例外，用的是 `summary_len`。多字段 payload 为除最后一个之外的每个变长字段存储长度； 最后一个字段的长度由所属记录的 `payload_len` 推导得出。下溢、必填文本字段 为空，以及声明为 UTF-8 的字段中出现非法

UTF-8，都会被拒绝。明确声明为 不透明 Git 字节的字段不做 UTF-8 解码。

Actor、Task、Change、Snapshot、Snapshot Link、Tag、Review、Waiver 以及 Plan 家族的记录使用 `u16 payload_len`，因此单个 payload 上限为 65535 字节。 Patchset summary 使用 `u16 summary_len`，要求 `1..65535` 字节，上界相同， 但并不因此变成通用的 Patchset payload。 Actor 各组成部分的长度和 Tree 路径段长度使用 `u8`，因此各自上限为 255 字节。 Git identity、commit 和 Tag 映射 payload 使用 `u32 payload_len`；Git repository identity、文件路径和 ref 名称长度同样是 `u32`。超出 `u32` 的值在写入之前就会被拒绝。`line_name_payload.bin`、`tree_name_payload.bin` 以及可选的 `snapshot_path_payload.bin` 存放所属记录 归一化后的 UTF-8 字节。

在当前命令契约下，AIT Tag 名称/消息、Git repository identity、Git identity 各组成部分、Git 文件路径和 Git ref 名称必须是合法 UTF-8。Git commit 消息、 原始 commit、Git Tag 消息以及带注解 Tag 的原始字节都是有长度上界的不透明 字节，在 Binary DB 中绝不做 Base64 编码。由 AIT 生成的 identity 可以用 `raw_len = 0`；从 Git 保留下来的 identity 必须保留其原始字节。轻量 Git Tag 的原始 Tag 字节为空。

Attestation、Policy Decision 和 Land 记录不持有 payload 文件。Patchset 只持有 `patchset_summary_payload.bin`；`patchset_payload.bin` 不是 v0 的文件。

可重建索引

`.idx` 文件是查找加速器，绝不是身份或权威。它们可以缺失，也可以重建。 命中只有在比对过所引用的权威记录或 payload 字节之后才被接受。持久化的行 先按查找键字节排序，再按目标索引排序。

```
SNAPSHOT_ID_INDEX_RECORD_SIZE = 12
SnapshotIdIndexRecord - snapshot_id.idx:
u64 snapshot_hash48
u32 content_snapshot_index_plus1

SNAPSHOT_PARENT_CHILD_INDEX_RECORD_SIZE = 8
SnapshotParentChildIndexRecord - snapshot_parent_child.idx:
u32 child_snapshot_index
u32 parent_edge_index_plus1

TREE_ID_INDEX_RECORD_SIZE = 14
TreeIdIndexRecord - tree_id.idx:
u8  tree_hash80[10]
u32 tree_index_plus1

TREE_PACK_ID_INDEX_RECORD_SIZE = 12
TreePackIdIndexRecord - tree_pack_id.idx:
u64 pack_hash48
u32 tree_pack_index_plus1
```

```
BLOB_ID_INDEX_RECORD_SIZE = 14
BlobIdIndexRecord - blob_id.idx:
u8  blob_hash80[10]
u32 blob_index_plus1

OBJECT_PACK_ID_INDEX_RECORD_SIZE = 12
ObjectPackIdIndexRecord - object_pack_id.idx:
u64 pack_hash48
u32 object_pack_index_plus1

MANIFEST_HASH_INDEX_RECORD_SIZE = 36
ManifestHashIndexRecord - manifest_hash.idx:
u8  manifest_hash[32]
u32 content_snapshot_index_plus1
```

```
ACTOR_LOOKUP_INDEX_RECORD_SIZE = 12
ActorLookupIndexRecord - actor_lookup.idx:
u64 actor_key_hash
u32 actor_index_plus1

LINE_NAME_INDEX_RECORD_SIZE = 12
LineNameIndexRecord - line_name.idx:
u64 line_name_hash64
u32 line_index

TAG_NAME_INDEX_RECORD_SIZE = 12
TagNameIndexRecord - tag_name.idx:
u64 tag_name_hash64
u32 tag_index
```

```
GIT_REPOSITORY_FINGERPRINT_INDEX_RECORD_SIZE = 16
GitRepositoryFingerprintIndexRecord - git_repository_fingerprint.idx:
u8  fingerprint96[12]
u32 repository_index_plus1

GIT_GENERATION_ID_INDEX_RECORD_SIZE = 12
GitGenerationIdIndexRecord - git_generation_id.idx:
u8  generation_hash64[8]
u32 generation_index_plus1

GIT_COMMIT_OBJECT_INDEX_RECORD_SIZE = 28
GitCommitObjectIndexRecord - git_commit_object.idx:
u32 generation_index
u8  git_object_id[20]
u32 commit_mapping_index_plus1

GIT_COMMIT_SNAPSHOT_INDEX_RECORD_SIZE = 8
GitCommitSnapshotIndexRecord - git_commit_snapshot.idx:
u32 snapshot_index
u32 commit_mapping_index_plus1
```

```
GIT_REF_LOOKUP_INDEX_RECORD_SIZE = 16
GitRefLookupIndexRecord - git_ref_lookup.idx:
u32 repository_index
u64 ref_name_hash64
u32 ref_mapping_index_plus1

GIT_TAG_REF_INDEX_RECORD_SIZE = 8
GitTagRefIndexRecord - git_tag_ref.idx:
u32 git_ref_mapping_index
u32 git_tag_mapping_index_plus1

GIT_OPERATION_ID_INDEX_RECORD_SIZE = 12
GitOperationIdIndexRecord - git_operation_id.idx:
u8  operation_hash64[8]
u32 checkpoint_index_plus1
```

Snapshot 父子键允许出现重复候选，并按升序返回边索引；权威的边序号在查找 之后再做校验。Git ref 查找使用导入源 repository 索引或导出目标 repository 索引。Tag 名称和 Git ref 名称的哈希命中，只有在做过精确的带类型 payload 比对之后才被接受。按 Snapshot 查 commit 可能同时返回导入和导出候选；由 generation 方向、映射标志位、对象格式和记录顺序共同决定 landed 优先级-- 未改动的导入优先于确定性导出。

Plan Item 的 ref 通过某个 revision 的条目区间加精确 payload 比对来解析。 布局 v0 没有定义持久化的 Plan Item ref 哈希索引。

元数据编码

下面未分配的所有位和值都是保留的，必须写为零。

```
TaskRecord.task_meta:
bit 0 planned
bit 1 snapshotted
bit 2 review_pending
bit 3 validation_pending
bit 4 blocked
bit 5 ready_to_land
bit 6 completed
bit 7 canceled

LocalTaskRecord.local_meta:
bit 0 published
bit 1 abandoned
bit 2 later_promotion_excluded

Remote task/change/mirror remote_meta:
bit 0 tombstone
bit 1 partial
bit 2 conflict
bit 3 stale
```

```
ChangeRecord.change_meta:
bits 0..1 lifecycle: 00 draft, 01 active, 10 landed, 11 archived
bit 2 has_patchsets
bit 3 review_pending
bit 4 validation_pending
bit 5 ready_to_land
bit 6 blocked
bit 7 superseded; valid only with lifecycle 11

ChangeRecord.change_state:
bit 0 canceled; valid only with lifecycle 11
bits 1..7 reserved = 0

LocalChangeRecord.local_meta:
bit 0 published

WorktreeCursorRecord.cursor_meta:
bit 0 has_selected_change
bit 1 has_pending_pre_land_target_snapshot
bit 2 has_pending_landed_snapshot
```

对于 Local Change，`local_meta` 的 bit 0 和 `published_remote_change_ordinal_plus1` 遵守 Change 记录一节里的存在性与 所属 Task 不变式。这个四字节的发布槽位绝不存放全局 Remote Change 索引。

```
LineRecord.line_meta:
bit 0 archived
bit 1 tombstone

TagRecord.tag_meta:
bit 7 tombstone

StashRecord.stash_meta:
bit 0 workspace_cleared
bit 7 tombstone

LandRecord.land_meta:
bits 0..2 status_kind:
  000 queued
  001 running
  010 succeeded
  011 blocked
  100 failed
  101 canceled
  110 updating
  111 reserved
bit 3 has_pre_land_target_snapshot
bit 4 has_landed_snapshot
bits 5..6 mode_kind:
  00 direct
  01 merge
  10 ff_only
  11 reserved
bit 7 tombstone

LandRecord.failure_kind:
0 none
1 base_stale
2 policy_blocked
3 review_blocked
4 ci_blocked
5 conflict
6 target_update_failed
7 internal_error
```

```
ContentSnapshotRecord.snapshot_meta:
bits 0..1 snapshot_kind: 00 line, 01 stash
bit 2 has_message
bit 3 has_line_name_payload
bit 4 parent_edges_authority
bit 5 has_root_locator
bit 7 tombstone

ContentSnapshotRecord.history_flags:
bit 0 remote_head_history_boundary
bits 1..7 reserved = 0

SnapshotLinkRecord.link_meta:
bit 0 has_change
bit 1 has_session
bit 2 has_checkpoint
bit 3 has_worktree_name
bit 4 has_line_name
bit 5 has_author_or_model
bit 7 tombstone
```

```
GitRepositoryRecord.repository_meta bits 0..1:
00 Git import source; public prefix GSR
01 Git export target; public prefix GTR
10 AIT export source; public prefix ASR
11 reserved

GitRepositoryRecord.object_format_kind:
0 not_applicable for an AIT repository
1 sha1

GitGenerationRecord.generation_meta bits 0..1:
00 import; public prefix GIT-IMP
01 export; public prefix GIT-EXP
10..11 reserved

GitIdentityRecord.identity_meta:
bit 0 has_raw_identity

GitCommitMappingRecord.mapping_meta:
bit 0 signed
bit 1 imported_unchanged
bit 2 deterministic
```

```
GitRefMappingRecord.ref_meta:
bits 0..1 ref_kind: 00 branch, 01 tag, 10 symbolic_ref, 11 reserved
bit 2 has_snapshot
bit 3 has_target
bit 4 has_expected_previous_git_object_id

GitTagMappingRecord.tag_mapping_meta:
bit 0 signed
bit 1 deterministic
bit 2 has_raw_tag

GitOperationCheckpointRecord.checkpoint_meta bits 0..1:
00 running
01 completed
10..11 reserved

Git git_object_type_kind:
0 commit
1 tree
2 blob
3 tag
```

```
PlanRecord.plan_meta:
bits 0..1 plan_state: 00 draft, 01 archived, 10 superseded
bit 2 published
bit 3 stale
bit 4 conflict
bit 5 tombstone

PlanRevisionRecord.revision_meta:
bit 0 published
bit 1 stale
bit 2 conflict
bit 3 has_items
bit 4 tombstone

PlanItemRecord.item_meta:
bits 0..1 checkbox_state: 00 none, 01 open, 10 done
bit 2 has_item_ref
bit 3 taskable
```

```
TreePackRecord.pack_meta:
bit 0 ready
bit 1 corrupt
bit 2 sparse_physical_ordinals
bit 7 tombstone

TreePackRecord.pack_format_kind:
0 zip_deflate_tree_v1
1 zstd_chunked_tree_v1

TreeRecord.tree_meta:
bit 7 tombstone

TreeEntryRecord.entry_meta bits 0..1:
00 blob
01 tree
```

```
ObjectPackRecord.pack_meta:
bit 0 ready
bit 1 corrupt
bit 2 pruned
bit 7 tombstone

ObjectPackRecord.pack_format_kind:
0 zip_deflate_v2
1 zstd_chunked_v1

ObjectPackMemberRecord.member_meta:
bits 0..1 entry_kind: 00 full, 01 delta
bits 2..3 compression_kind: 00 none, 01 deflate, 10 zstd
bit 7 tombstone

BlobRecord.blob_meta:
bit 0 has_pack_member
bit 1 pruned
bit 7 tombstone

BlobRecord.hash_kind:
0 sha256
```

```
PatchsetRecord.patchset_meta:
bit 0 withdrawn
bit 1 invalidated
bits 2..4 author_mode_kind:
  000 human_only
  001 human_with_ai_assist
  010 ai_with_human_review
  011 ai_only_experimental
  100 agent (legacy conversion only)
  101 codex (legacy conversion only)
  110 xhigh (legacy conversion only)
  111 reserved
bits 5..6 publish_state_kind:
  00 published
  01 reserved
  10 superseded
  11 reserved
bit 7 evaluation_pending

AttestationRecord.attest_meta:
bits 0..1 verification_state: 00 unknown, 01 pending, 10 pass, 11 fail
bit 2 revoked
bit 3 require_tests_pass
bit 4 require_human_review
bit 5 require_lint_pass
bit 6 ci_backed
bit 7 tombstone

ActorRecord.actor_meta:
bits 0..2 actor_kind:
  000 user
  001 team
  010 bot
  011 service_account
  100 automation
  101 unknown
bit 3 has_user_id
bit 4 has_email
bit 5 has_memo
bit 7 tombstone
```

原生 v0 的 Patchset 写入方只会写出四种非遗留的 author mode。那三个遗留
编码存在的唯一目的，是在转换过程中原样保留已被接纳的源行；无法识别的 author 文本会 fail closed。原生 v0 的 Patchset 写入方只会写出 `published` 或 `superseded` 发布状态。遗留源里的 Patchset 状态 `selected_for_landing` 是非权威投影，会归一化为 `published`；它绝不会被写成 `01`，不会被用来推断选择结果，也不允许覆盖 `RemoteChangeRecord.selected_patchset_index_plus1`。如果一个有效的源 Change 选中指针与陈旧的 Patchset 投影不一致，以 Change 指针为准，转换不会仅仅因为
这种投影不一致而失败。但只要需要源端的选择结果，缺失、无效或有歧义的源 Change 指针依然会导致失败。只有当源 Change 指针明确这么指、并且所有权和 存活规则也全部通过时，选中指针才可以引用 `published` 或 `superseded` 的遗留行。

当 `evaluation_pending` 置位时，即便存在更早的 Policy Decision，对外暴露的 Patchset `evaluation_state` 也是 `pending`。当它清零时，必须有一条最新的、 存活的非 pending Policy Decision，由它提供确切的 `pass`、`soft_fail`、`hard_fail` 或 `waived` 值。新建 Patchset，以及任何会让缓存策略失效的 review、attestation、waiver
或策略输入变更，都会把这一位置位，作为状态最后 提交的标记；一次完成的非 pending Policy
评估会先追加其决定，再清除该位。

```
ReviewRecord.review_meta:
bits 0..2 action_kind:
  000 request
  001 comment
  010 approve
  011 request_changes
  100 dismiss
bit 3 blocking
bit 4 task_lane
bit 5 code_review_summary
bit 6 defer
bit 7 tombstone

PolicyDecisionRecord.policy_meta:
bits 0..2 decision_kind:
  000 pending
  001 pass
  010 soft_fail
  011 hard_fail
  100 waived
bit 7 tombstone

WaiverRecord.waiver_meta:
bit 0 revoked
bit 7 tombstone
```

```
PolicyCheckRecord.check_kind:
0 require_attestation
1 ai_provenance
2 code_review_summary
3 tests
4 lint
5 security_scan
6 license_scan
7 required_human_review
8 ci_rollout_phase
9 ci_patchset_suite

PolicyCheckRecord.check_status:
0 absent
1 not_required
2 pending
3 pass
4 hard_fail
5 soft_fail
6 waived
7 optional_fail
```

BINARY DB V0 结构

Binary DB v0：身份、恢复与转换

展开公开句柄的作用域、原子写入与恢复边界、转换的接纳条件、容量与排除项。

适用人群 迁移、恢复与兼容性实现者

https://ait-native.dev/technical/v1.1.1/binary-db/recovery-conversion/

1.1.1 文档路由 · Binary DB v0 · `layout_id = 1`

schema 的最后一章把物理记录同对外身份、崩溃恢复以及被接纳的转换连接
起来。它同时给出硬性容量上限、明确排除在外的领域和文件，以及布局 v0 的兼容性边界。

对外身份与紧凑句柄

- Task 身份以权威为作用域，由稠密的记录序号推导得出。
- Change 用 `C-01..C-64` 表示序号 `0..63`。
- 已发布 Local Change 的 Remote 身份，是所属 Local Task 已发布的 Remote Task 索引，与所存的 Remote Change 序号配对而成； 它不是全局 Remote Change 索引。
- Land 在所属 Change 内用 `L-01..L-64` 表示序号 `0..63`；它完整的 对外身份包含 Task 和 Change 身份。
- Patchset 在所属 Change 内用 `P-01..P-64` 表示序号 `0..63`；它完整的 对外身份包含 Task 和 Change 身份。
- Attestation 用 `A-01..A-64` 表示序号 `0..63`。
- Review 在所属 Patchset 内用 `R-01..R-64` 表示序号 `0..63`；它完整的 对外身份包含 Task、Change 和 Patchset 身份。
- Policy Decision 在所属 Patchset 内用 `K-01..K-64` 表示序号 `0..63`； 它完整的对外身份包含 Task、Change 和 Patchset 身份。
- Waiver 用 `W-01..W-64` 表示序号 `0..63`。
- Task Snapshot Link 用 `S-01..S-256` 表示序号 `0..255`。
- Stash 身份由 `stash_index` 推导得出，例如 `STH-000001`。
- AIT Tag 的对外身份就是它的名字；稳定的内部引用使用稠密的 `tag_index`，从不需要文本形式的 Tag ID。
- 规范的 Snapshot 身份是 `SNP-` 加一段 48 位十六进制后缀。
- Tree 和 Blob 的对外后缀是 80 位。
- Tree Pack 和 Object Pack 的对外后缀是 48 位。
- Git repository 指纹是 96 位后缀，导入/导出 generation 与 operation 身份是 64 位后缀，Plan 哈希是 96 位后缀， Git 映射身份是推导出的 80 位 `GIM-*` 后缀。

每个八位工作流句柄，在其声明的所属作用域内都使用下面这套位模型：

```
bits 0..5 ordinal 0..63 within (owning identity, kind)
bit 6 tombstone/deleted
bit 7 indirect/extended
```

Change、Attestation 和 Waiver 保持各自声明的、以 Task 为作用域的分配方式。 Patchset 和 Land 在所属 Change 内分配。Review 和 Policy Decision 在所属 Patchset 内分配。因此同一个 Task 里不同 Change 之间， Patchset 或 Land 序号可以重复；不同 Patchset 之间，Review 或 Policy
序号也可以重复。完整的所属身份能把每一个重复的数字后缀区分开。

布局 v0 没有指定扩展句柄的 payload 格式。写入方一旦会超出声明的序号 范围，就必须在写入之前失败；它不得把 indirect 位当成自行发明一种格式的 许可。

多文件写入与恢复边界

布局 v0 没有定义工作流层面的 `wal.bin`、`journal.bin`、事务 ID 字段、 按记录的 WAL 序列，也没有 Binary DB 激活 generation 令牌。 `GitGenerationRecord` 标识的是导入的 Git 源状态或一份确定性的 导出计划；它不是数据库事务令牌。写入方的事务层负责单写入方/按 Task 加锁、依赖优先写入、有序的 fsync、只追加的提交点，以及可修复的 head/索引投影。

追加进去的明细记录，就是 Patchset、Review、 Policy Decision、Waiver、Attestation 和 Land 创建时的领域提交点。相关的 task、 change 或 patchset 头部记录都是可修复的投影。Land 状态变更采用 status-last 协议：先写入终态/重置数据并 fsync，再把 `status_kind` 作为 提交标记写入并 fsync。

`tree_entry_range.bin` 是唯一按序号对齐的定长依赖文件，它既不是 payload，也不是可重建索引。创建时会先写入每一条必需的区间行并 fsync， 之后才写所属的 Tree 提交记录。在稳定的激活边界上，它的条数等于 Tree 属主文件的条数；恢复时只能丢弃未提交的尾部依赖，遇到已提交依赖缺失或中间序号不匹配则必须 fail closed。Task 关闭的变更完全发生在 `task.bin` 内部；Change 归档/重开的变更完全发生在 `change.bin` 内部。Land 创建会追加并持久提交一条完整的 `land.bin` 记录。每一次变更都替换或追加它那一条完整的定长 记录。若一次覆写没有抵达提交点，事务层会恢复其前像；单条记录内部 字段之间的顺序对外不可观测。

恢复过程会校验每一个头部和记录对齐，遍历已提交的明细 记录，修复 `latest_*_index_plus1`、计数和 next 序号，重算 最新/当前 Patchset 索引，按所属关系和存活性校验每个 Remote Change 选中的 Patchset 指针（而不是从 Patchset 元数据重建它），忽略孤立的依赖/payload 行， 并把每个声明身份作用域内的重复序号判定为损坏而拒绝。Task 层面用于 Patchset、Review、Policy Decision 和 Land 的索引只重建物理 清单顺序。它们的属主索引则重建序号链和 next 序号分配：Patchset 和 Land 的属主是 Change，Review 和 Policy Decision 的属主是 Patchset。不同属主作用域下出现相同的数字序号是合法的。

Snapshot 创建时会先追加每一条经过校验的父边并对边文件 fsync， 之后才追加带 bit 4 置位的子 Snapshot 记录。Snapshot 记录 就是提交点；写入被打断可能只留下孤立的边，恢复时会忽略它们。 恢复过程还会校验：远端 head 的历史边界没有本地父级， 且零父级的非边界 Snapshot 确实是真正的根。Tag payload 必须在 Tag 记录追加/更新之前落盘持久化。

Tree 创建时先追加归一化的 Tree entry 行和与之匹配的定长区间， 再写 Tree 提交记录。恢复过程会对照所属 Tree 校验归一化区间，并按 Tree Pack 的稀疏序号位校验每一个物理 pack 序号，然后才接纳该 pack 投影。

Git repository、generation、identity、parent、file 以及带类型的 payload 行都是 依赖。不可变的 Git commit/ref/Tag 映射记录是其映射的 提交点，会先于可重建索引候选完成 fsync。恢复过程会 忽略从已提交映射不可达的依赖行，校验全部映射区间和数值形式的 AIT 引用，然后重建 Git 索引。Git operation checkpoint 会先写入自己的游标和时间，再写入并 fsync `checkpoint_meta`；重放一个已完成的 checkpoint 是空操作，可从不可变映射 重建得出。

bin 到 bin 的转换接纳条件

转换器必须把每一个源字段归入以下几类之一：精确的定长字段/位 映射、唯一被接纳的 Patchset summary payload、schema 定义的 推导，或明确的非权威投影。落在这几类之外的字段会 fail closed； 不得把它塞进通用 payload。

离线转换器写入的是一个独立的目标 generation，可以分配新的 稠密物理目标记录索引。它保留源端的 Task 顺序，并且对其他 每一个权威文件家族，在已写出的行之间保留经过校验的源记录顺序。 在写记录之前，它会为每个被接纳的家族建立完整的源索引到目标索引的 映射。随后通过这些映射改写每一个属主索引、`*_index`、`*_index_plus1`、前一条记录的链接、selected/latest 指针、定长区间起点、按序号对齐的旁路行，以及重算后的 payload 偏移。可重建的 `.idx` 文件是从完成后的目标权威生成的，而不是复制过来的。

物理重编号绝不改变 `change_ordinal`、`patch_ordinal`、 `attest_ordinal`、`review_ordinal`、`policy_ordinal`、`waiver_ordinal`、 `land_ordinal` 或 `snapshot_ordinal`；它们的对外句柄在各自声明的作用域内 保留与源端完全一致的序号。Patchset 序号在 Change 内保留， Review 序号在 Patchset 内保留，Policy Decision 序号在 Patchset 内保留，Land 序号在 Change 内保留。跨不同所属 Change 或 Patchset 的重复是合法的，不会被重新编号。对 Policy Decision 而言， 被接纳的遗留 `.../P-##/POL-##` 身份会把确切的 `POL-##` 后缀映射到 以 Patchset 为作用域的 `K-##`。

同一个声明属主作用域内出现任何重复、同一种类在该属主作用域内超过 64 行、所属关系有歧义，或源 ID 与定长序列不一致，都会 fail closed。Task 层面的 Patchset、Review、Policy 和 Land 清单 链接按写出的物理记录顺序重建。Change 的 Patchset 和 Land 链接，以及 Patchset 的 Review 和 Policy 链接，按确切序号重建。任何 在目标映射中没有恰好一条对应项的源引用都会 fail closed。源

文件保持不可变，完全通过校验的目标 generation 会被原子激活。 不会额外增加任何新旧映射的 bin 或 payload。

唯一被接纳的 `diff_stats` 不一致情形，是显式选中的 离线遗留 stale-zero 闸门。该闸门使用一份精确配置的、由完整 Patchset 身份组成的允许列表；禁止部分匹配、只按序号匹配或通配匹配。

对于列表中的每个 Patchset，提供的值必须是完整对象，其中 `files_added = 0`、`files_changed = 0`、`files_deleted = 0`、 `files_modified = 0`，并且 `paths.added`、`paths.deleted` 和 `paths.modified` 数组恰好为空。它规范的 base 和 revision Snapshot 引用 必须都能解析到存在且存活的 v0 Snapshot 权威，且对二者 完整 Tree 的精确比较必须成功并产生非空差异。转换器随后 只用这份重算出来的差异来校验是否可接纳； 目标端不存储任何 diff-stat 值。

没有显式开启该闸门时，任何不一致都会 fail closed。即便开启了 闸门，未列入或仅部分匹配的身份、缺失/多余/改名的 `diff_stats` 成员、提供的非零计数、提供的非空路径 数组、重算出的零差异、缺失/被墓碑标记/格式错误的 Snapshot、不可读或格式错误的 Tree，以及任何其他不一致，都会 fail closed。不接纳任何 通配、只按序号匹配、兜底 Snapshot、重建出来的源值、 新字段、新位、新 bin、新索引、新 payload、记录宽度变更或 `layout_id` 变更。

唯一被接纳的另一种源 Task 状态写法，是显式 选中的、绑定到确切 repository 键和完整锁定源清单摘要的离线闸门。服务端编排器会在转发该闸门之前校验这两个值。仅在该闸门下，源端的 `status = "canceled"` 映射到既有的 `TaskRecord.task_meta` bit 7， 与源端 `status = "abandoned"` 完全一致。遗留 payload 里没有 Task 关闭时间字段，因此转换后的终态 Task 会依据既有的"关闭时间未知"规则 保留 `closed_at_s = 0`。

没有显式开启该闸门时，源端 `status = "canceled"` 会 fail closed。即便 开启了它，另一份清单、另一个 repository 键、部分或大小写折叠的 写法，以及任何其他未知的 Task 状态，都会 fail closed。该闸门不会改变转换报告，也不会新增任何字段、位、bin、索引、payload、 记录宽度变更、重建标记、推断出的时间戳，或 `layout_id` 变更。

第一种引用缺失的例外，是针对某个 Patchset 的显式选中离线 遗留抢救模式--该 Patchset 规范的 `revision_snapshot_id` 在同一份锁定源 generation 的内容 Snapshot 权威中不存在。没有开启那个显式模式时，引用缺失会 fail closed。开启之后，转换器 不为该 Patchset 写出任何行，也不写出任何依赖被省略 Patchset 的 Review、Policy Decision 或 Check、Attestation、Land。如果 被省略的 Patchset 是所属 Change 权威的 current/latest 或 selected Patchset，转换器就不为该 Change 及其任何 Patchset 和依赖项写出行；它绝不会挑一个兜底的 current 或 selected Patchset。所属的 Task 仍然保留。

这一例外不接纳缺失的或哨兵值的 Patchset Snapshot 索引， 不会凭空造出或墓碑标记一个 Snapshot，也不会豁免缺失的 base Snapshot、格式错误的 Snapshot ID、已被墓碑标记或损坏的现存 Snapshot， 以及任何无关的权威缺失。目标端的 Change、Patchset、Review、Policy、 Attestation、Land 和 Actor 行在写出的行之间保留源顺序，每一个存留下来的定长引用、链头、计数、selected/latest 指针、 按序号对齐的旁路行和可重建索引都会被稠密地重映射。转换报告必须逐一列出被省略的 Change 和 Patchset 身份， 以及写出/省略的依赖行计数；省略绝不静默发生，也绝不改动源字节。

第二种省略例外，是针对某个确切配置的 Change 身份、需要单独选中的 具名遗留抢救。那个源 Change 声称已 landed，但没有存留下来的成功 Land， 也没有完整的、可据以重建的已接受 Patchset/Snapshot 闭包。没有那个确切具名选项时，转换会 fail closed。开启之后， 转换器会省略该 Change、它拥有的每一个 Patchset 和依赖行，以及每一条引用它的 Land 行； 它不会凭空造出 Land、不会挑兜底 Patchset、不会推断时间戳，也不会影响任何其他 Change。报告中必须写明所选中的 Change 及全部依赖行计数。不允许任何通配、基于状态或隐式的省略。

被接纳的遗留 landed Change 归一化，比上面几种省略例外的范围更窄。 对于 `current_patchset_number = 0` 的 Change，只有当它显式的 selected 指针指向同一个 Patchset 时，才从它存留下来的最大 `patch_ordinal` 重建 latest/current。即便重复出现的 Change `status` 或 `landed_at` 投影有出入，确切的 Land 行依然权威。每一次尝试都会被保留， 只要有一次尝试成功，Change 的生命周期就是 landed，且由最大的成功 `land_ordinal` 提供 landed 状态--哪怕之后还有一次未成功的尝试。当前 可变的 selected 指针不必等于任何历史 Land 所接受的 Patchset。这些规则只丢弃重复的投影，绝不丢弃 权威行，也不新增任何字段、位、bin、payload、兜底 Patchset 或 推断出的时间戳。

在转换之前，服务端编排器可以在不获取也不创建源锁的前提下，把源权威 复制到一个隔离的冻结 generation 中。冻结操作会在复制前 捕获当前每一个权威数据文件，复制这些捕获到的文件而不是信任 活跃 generation 那份可能已经过期的文件列表，然后再次捕获完整的 源清单，并要求按 `relative_path`、`byte_size` 和完整 SHA-256 做到复制前/复制后/副本三者精确相等。它还要求源注册表清单的 确切字节在整个操作过程中保持不变。任何漂移都会 fail closed，且不会留下 已发布的目标。新生成的冻结 generation 沿用既有的清单 schema，其文件列表和指纹恰好对应那些被复制的字节。Repository 清单副本和运行期锁文件不计入数据文件列表。只有当每一个声明的文件都匹配、且不存在未声明的权威数据文件时， 转换才会接受那个冻结 generation。

无论在选中的源里还是在冻结副本里，都不要求也不创建源锁文件。
只有当选中的源清单明确证明某个确切文件或它整个声明家族的记录数为零，
或者该权威类别根本不包含那整个可选家族、并且没有任何存留的源记录 会引用它时，编排器才可以把缺失的遗留 `.bin` 或 `.idx` 创建为只有头部的文件。家族只存在一部分、清单计数非零、 存在未解析的引用，或缺少证明，都会 fail closed。
绝不在选中的源根目录里做 头部合成，且转换报告会逐一列出每一个暂存的只有头部的文件。

新接纳的时间缺失置零情形仅限于 `RemoteTaskRecord.updated_at_s`、`RemoteTaskRecord.fetched_at_s`、`RemoteChangeRecord.fetched_at_s`、已归档的 `RemoteChangeRecord.archived_at_s`，以及 `ActorRecord.created_at_s`/`last_seen_at_s`。它们只在所选 遗留源格式确实没有对应时间字段时才适用。
先前列举的遗留 Task `closed_at_s` 和首次成功 Land 的 `submitted_at_s`
例外保持不变。除此之外，其他缺失或非法的时间都不会 变成零，且零值不授权任何合成时间或重建位。

源端的 `identity_source` 是注解，不是被存储的身份。只有当它缺失，
或与从选中权威根和数值形式的记录关系推导出的规范身份一致时，才可以 省略它。非默认、冲突或无法解析的值会 fail closed。源端的 `published_remote_name` 同样不被存储。只有当它缺失/为空，
或与本次转换显式选定的那一个远端权威完全相等时，才可以省略它； 不同的或有歧义的远端名称会 fail closed。

只有当源权威能证明某个源 Snapshot 确实没有父级时，它才会成为真正的 v0
根。因此被证明为空的权威依然为空，两个或更多被独立证明的根，在同一片 DAG
森林中依然是各自独立的连通分量。每一个存留的非边界 Snapshot 都必须
能到达这样一个根。如果源端证据反而表明它有祖先、只是那些父级没有被 导入，转换器就会设置
`remote_head_history_boundary`；若证据不足、 无法区分这两种情况，则 fail
closed。转换器绝不会仅凭位置、记录顺序、 连通分量大小、Line 归属或当前 head 状态就推断出一个根或历史边界。

文本形式的对外 ID 只按其声明的紧凑序号或 哈希规则解析，随后再对照所属的定长记录做校验。源端的缓存
键、推导出的标签/消息、重复的名称、diff 统计和结果
投影，只有在本权威文档所列的显式推导规则之下才可以被丢弃。未知的
枚举文本、溢出、不一致、权威缺失或解析有歧义，都会在激活之前中止转换。

容量与扩展

在做任何部分追加之前，写入方都会预检可表示的计数和窄字段：

```
current_count  = (file_size - BIN_HEADER_SIZE) / record_size
projected_count = current_count + records_to_append
```

超长的 u8/u16 文本属于输入校验失败，不是加宽布局 v0 的 许可。除了明确列举的 Task `closed_at_s` 尾部、Change
生命周期尾部、Land 目标 Line 尾部，以及 Patchset summary/Worker Job 定位符/时间宽度这几项修正，加上列举出的
Git 映射 时间字段移除，以及单独划定作用域的运行期家族之外，真正的 记录/索引容量扩展都需要显式迁移到新的
`layout_id`，转换完整的依赖引用闭包，写入 独立的新文件，校验它们，并执行原子切换。有界的
运行期转换会迁移并收窄 Worker Job 记录，废止它的 全局身份家族，只新增两个声明的本地索引，并只追加 声明的
Patchset 定位符；更早的定长时间修正只收窄了具名的、 带时间戳的那些记录以及 ready 索引。随后的 u64 秒转换只加宽
完整列举出的普通时间字段集合。这两次转换都不允许改动其他字段或文件。 即便持久化的布局编号仍然是
1，同一个文件里混用不同记录宽度依然是禁止的。

尽管 `parent_count` 和 `parent_ordinal` 是 `u16`，Snapshot 父级数量的上限仍是 1,024。AIT Tag payload 的上限由 `u16` `payload_len` 决定。 Git payload 和单字段长度值的上限由 `u32` 决定；Git SHA-1、 指纹、generation、operation 和 Plan
哈希的字节数组，都采用上面声明的确切 定长宽度。除了明确移除的 AIT
生成的映射时间戳之外，禁止收窄、截断、用哈希前缀 替代，也禁止在这些 20 字节字段里接受 SHA-256 的 Git Object ID。

剩下的那一个按序号对齐的旁路文件家族，会在写入前预检其属主计数和 所有 `u32` 索引。它是 layout-1 的 schema
扩展，不是 加宽既有记录或把其字段搬进 payload 的许可。

Task 尾部是一次性的 layout-1 转换，不构成通用的 加宽先例。转换必须显式选中旧的本地 40 字节或 远端 36
字节源布局，把整个文件改写成修正后的本地 44 字节或远端 40 字节布局，并且只有在完整校验之后才激活。

Change 尾部同样是一次性的 layout-1 转换。 对紧邻的前一种拆分表示做转换时，必须显式 选中 44 字节的 Change
源布局及其对齐的 8 字节生命周期 源，改写并校验完整的 52 字节 `change.bin`，在新 generation 中省略
`change_lifecycle.bin`，并且只有在每一处 base Line、归档时间、属主和 Snapshot 关系都校验通过之后才激活。

Land 尾部同样是一次性的 layout-1 转换。 对紧邻的前一种拆分表示做转换时，必须显式 选中 32 字节的 Local 或 36 字节的
Server Land 源布局及其对齐的 4 字节目标 Line 源，改写并校验完整的 36 字节 Local 或 40 字节 Server `land.bin`，在新
generation 中省略 `land_target_line.bin`， 并且只有在每一处目标 Line、属主、存在时的 Patchset、

Snapshot、序号链、状态和时间戳关系都校验通过之后才激活。

Patchset summary 定位符同样是一次性的 layout-1 转换。 转换 47 字节的源时，必须先构造并校验声明的 57 字节前缀加 `patchset_summary_payload.bin`。之后的 Worker Job 定位符修正再显式选中那个确切的 57 字节 前缀，追加 `ci_worker_job_index_plus1`，在一个独立 generation 中写入完整的 61 字节 `patchset.bin`，并且只有在每一处 summary 区间、同 Repository 的 Worker Job 目标以及 Patchset 关系都校验通过之后才原子激活。

Repository 内本地化的 Worker Job 身份变更也是一次性的 layout-1 转换。转换时必须在声明的 100 字节源记录存在时显式选中它，移除其中的 `u64 job_id`，改写并 校验完整的 92 字节 `worker_job.bin`，用两个声明的 Repository 本地索引 替换被取代的全局 Job 序列/索引文件，并且 只有在每一个永久本地索引、带类型 payload、Patchset 定位符、状态和时间戳都校验通过之后才激活。

之后的 Worker Job 定长记录变更是又一次一次性的 layout-1 转换。转换时必须显式选中那个确切的 92 字节源记录及其四个带类型 payload 家族，解析并校验 它们的完整闭包，在一个独立的非激活 generation 中写入声明的 88 字节 `worker_job.bin` 加 `worker_job_input_payload.bin`，并在目标中省略 `worker_job_request_payload.bin`、`worker_job_result_payload.bin`、 `worker_job_error_payload.bin` 和 `worker_job_lease_owner_payload.bin`。 只有在每一个 Job kind、定长引用、归一化输入、结果、相关 Job、错误、lease 哈希、Patchset 定位符、状态和时间戳都校验通过之后，才会激活。

取消 payload 并改用运行期 lease 的变更是又一次一次性的 layout-1 转换。转换时必须显式选中那个确切的 88 字节源记录和 `worker_job_input_payload.bin`，解析并校验 它们的完整闭包，在一个独立的非激活 generation 中写入声明的 60 字节 `worker_job.bin`，并在目标中省略 `worker_job_input_payload.bin`。 只有在定长引用、源 payload 契约以及"已静默、无运行中 Job"的 闸门都校验通过之后，它才丢弃源输入区间和 `lease_owner_hash`。 转换不会写入任何运行期 lease 副本。只有在 每一个 Job kind、定长引用、结果、相关 Job、错误、 Patchset 定位符、状态和时间戳都校验通过之后，才会激活。

之后的 Worker Job 直接引用变更是又一次一次性的 layout-1 转换。转换会显式选中那个确切的 60 字节源记录，并写入声明的 52 字节 `worker_job.bin`。 对 `land.process` 而言，旧的主体 Land 必须解析为确切的 `main`、 `direct`、同 Repository 的 Server Land，其已接受的 Patchset 成为 `patchset_index_plus1`。对 `main-seed.refresh` 而言，旧的主体 Patchset 和 辅助的先前 Snapshot 成为那两个具名字段；它旧的上下文 Line 必须恰好是 `main`，并被丢弃。Patchset 的 CI、聚合和 Policy 主体成为 `patchset_index_plus1`；`repo.ci` 主体在其旧的上下文 Line 精确校验为 `main` 之后成为 `snapshot_index_plus1`。 内容维护和 Repository 对账要求那三个旧的领域引用全部为零。 旧的相关 Job 引用在存在时会对源端的 `attached` 或 `superseded` 结果做精确校验，随后被丢弃。 只有在每一个新的具名引用、Job kind、结果、 错误、Patchset 定位符、状态和时间戳都校验通过之后，才会激活。这次历史 改写不会让原生 v0 的入队依赖于一个已存在的 Land。

定长时间变更是又一次一次性的 layout-1 转换。它选中 确切的 33 字节 Repository、52 字节 Worker Job、 12 字节 ready 索引和 61 字节 Patchset 前身，并改写出 声明的 25 字节、36 字节、8 字节和 57 字节目标。被接纳的 RFC 3339 运行期时间戳会按上面声明的方式归一化，并有意缩减到整秒。既有的 Patchset `ci_completed_at_s` 值必须放得进 `u32`；溢出会在写入任何目标之前 失败，而不是被截断。Patchset 的改写保留其前 28 字节以及逻辑上的 CI、summary 和 Job 定位符值，同时把尾部移到 修正后的偏移上。

同一次修正会选中确切的 100 字节 commit 映射、84 字节 ref 映射 和 52 字节 Tag 映射前身，并写出它们声明的 92 字节、76 字节和 44 字节形式，其中不含 `recorded_at_unix_nanos`。在丢弃 这个由 AIT 生成的值之前，转换会核实物理记录顺序 能复现源端每一次"最新记录"的选择结果；不一致则 fail closed。 带签名的 Git identity 时间戳和时区仍然是逐字节保留语义的 源数据。只有在非激活 generation 中，记录整除关系、每一条 时间戳区间与排序规则、所有索引以及完整的依赖 引用闭包都校验通过之后，才会激活。

u64 秒变更是又一次一次性的 layout-1 转换。 它唯一被接纳的前身选择符是确切的 `u32-time-v0`；只选 `layout_id = 1`，或者从文件整除关系去猜，都是不够的。该选择符指的是 紧邻的前一个完整活跃 v0 表示，其普通 `*_at_s` 字段和宽度已在上面的修正表中 列举。缺少选择符、换成别的选择符、出现未声明的文件、某个源文件 未精确对齐到其具名前身宽度，或者出现混合宽度的 文件，都会在发布任何目标之前失败。

转换会冻结或读锁定完整的源权威，并记录其 完整文件清单和指纹。对每一条定长记录，它会按确切的声明顺序 复制声明时间字段之前、之间和之后的全部字节，并把每个小端 `u32` 秒值零扩展成对应的小端 `u64`。它不会解析、取整、重排、推断或 替换任何时间戳。它会逐字节精确复制每一个不受影响的 payload 和对象。 `GitIdentityRecord.timestamp_s` 不做转换。Worker 的 `worker_ready.idx` 会从转换后的权威 Job 记录重建； 其余所有可重建索引，要么从转换后的权威重建，要么对照未改动的身份和序号 键做精确校验。

转换器写出一个完整的非激活 generation，用活跃的 v0 编解码器校验 每一条修正后的记录，校验完整的跨文件引用和内容闭包， 并把仍处于锁定状态的源指纹与它捕获的前置条件做比较。激活要求 读写方都兼容，会重新检查那个源前置条件，并原子地只选中完整的 generation，同时保留一个可恢复的先前 generation

或直接 权威。转换失败或冻结后源发生变化，都不会动到活跃 权威。活跃文件绝不会混用前身和修正后的记录宽度，且在转换或激活持有权威锁期间，任何运行时都不得写入其中任一表示。

排除在外的领域与文件

布局 v0 不定义：

- `wal.bin`、`journal.bin` 或按领域划分的事务字段；声明的 Git 导入/导出内容 generation 是 v0 中唯一的 generation ID 例外；
- Local Patchset 文件、Local Change 记录中的 selected-Patchset 游标，或 不可变 Patchset 元数据中的 selected-for-landing 权威；
- 通用的 `ci.bin` 或 `job.bin`；运行期作业只使用带类型的、 以 Repository 为作用域的 `worker_job.bin` 家族，而紧凑的 Patchset CI 证据 及其选中的本地 Job 定位符仍留在 Patchset 记录里；
- 通用的 `patchset_payload.bin`、`attest_payload.bin`、`policy_payload.bin` 或 `land_payload.bin`；类型收得很窄的 `patchset_summary_payload.bin` 是 Patchset 唯一的例外；
- 任何 payload 或旁路文件中的 Task 关闭时间；所属 Task 记录的 `closed_at_s` 尾部字段是其唯一权威；
- 任何 payload 或旁路文件中的 Change base Line/归档状态；所属 Change 记录的定长尾部字段是它们的唯一权威；
- 任何 payload 文件中的 Land 目标 Line 身份或归一化的 Tree entry 区间； 上面声明的所属 Land 记录尾部和定长 Tree 区间旁路记录 是它们的唯一权威；
- 服务端的 Stash 文件；
- 仅用于转换的 Actor 文件或通用 Actor payload；转换使用 既有的 `actor.bin` 和 `actor_payload.bin` 权威；
- 共享字符串池；
- `object_pack_data.bin`；对象字节仍留在它们的 pack 容器中；
- `line.bin` 里全仓库范围的 current-Line 字段；
- `snapshot_payload.bin` 里的 Snapshot 父索引扩展；
- `LocalChangeRecord` 里的全局已发布 Remote Change 索引；
- Git mirror 映射、最后一次 mirror 的 head 状态、mirror 方向/阶段、 运行期结果 JSON，或通用的 Git 映射 payload；
- Worker Job 的结果 JSON、完整错误文本、文本形式的 lease 属主身份，或 `worker_job_request_payload.bin`、`worker_job_result_payload.bin`、`worker_job_error_payload.bin` 和 `worker_job_lease_owner_payload.bin` 的活跃定义；`worker_job_input_payload.bin`、`input_len`、`input_offset` 和 `lease_owner_hash` 同样从活跃 v0 中 废止；
- Worker Job 的 Land 索引、目标 Line 索引、相关 Job 索引，或第二个 Snapshot 引用字段；活跃 v0 只存储 `worker_job.bin` 中声明的那些具名 Patchset 和单个 Snapshot 引用；
- `identity_source`、`published_remote_name`、Git 映射中重复的 AIT Line/Tag 名称，或文本形式存储的 Line ID；
- 作为 Binary DB 权威的通用运行期 payload、服务端调度策略/配置 payload、指标/时序存储、追踪/日志存储、通知/outbox 存储、软件包制品字节、外部提供方令牌保险库，或临时的运行期 lease 副本；
- 其他服务端全局运行期领域；活跃的运行期 v0 只包含 全局 Repository 注册表/名称 payload/命名空间索引，以及上面声明的 以 Repository 为作用域的 Worker Job 家族；
- 服务端全局的 Worker Job 定长记录、payload、公开序列，或 调度/身份索引；定长的 Job 权威和两个可重建的 Job 索引都留在各个数值 Repository 权威内部；
- 作为 Binary 权威的已废止运行期/规划会话，或 Test 清单/覆盖率数据； 以及
- 从服务端全局指向仓库工作流/内容/Plan 权威的数值引用。那些跨根引用仍然是上面声明的、带类型的确切对外身份字节。

遗留的 SQLite 可以由显式的一次性导入/导出工具读取，但它不是 权威的运行期兜底。已落地的 `.ait/git-interop/v1` JSON 映射/checkpoint 存储同样只能被一次显式的一次性转换读取，转换成声明的 Git 定长记录和带类型 payload；它不是并行的运行期权威。可选缓存和可重建索引，绝不能成为某个领域关系的 唯一来源。

参考

分发与发布

了解七个组件的兼容家族、各安装渠道的角色，以及确切的 1.1.1 权威。

适用人群 开发者、运维与打包者

https://ait-native.dev/technical/v1.1.1/distribution/

同一个兼容家族

ait-native 1.1.1 一次接纳七个组件：

- `ait` -- 原生仓库与工作流 CLI；
- `ait-agent` -- 可选的传输 worker 配置与监管 CLI；
- `ait-agent-worker` -- 原生 Telegram、LINE、Discord 和 Slack worker 运行时；
- `ait-server` -- 远程协议与持久权威；
- `ait-runner` -- 远程原生执行面；
- `ait-python` -- 直接的进程内 Python 绑定；以及
- `ait-node` -- 直接的 Node-API 绑定和 npm 打包组件。

这七个组件来自五个独立的源码权威。每个组件都保留自己的源 Snapshot、 产物 digest 和许可证身份。光是版本号对得上，并不能替代 release family 的接纳。

公开安装渠道

- GitHub Release 提供这七个组件家族，形式是规范的原生产物、源码、 校验和、验证与回执资产。
- Homebrew 为受支持的 macOS 和 Linux 目标安装 stable 的 `ait-native` formula。
- PyPI 发布 `ait-native==1.1.1` 平台 wheel，内含 `ait`、`ait-server` 和 直接 Python 绑定。
- npm 在 `latest` dist-tag 下发布 `@wa120/ait-native@1.1.1`，外加六个 锁定确切版本的 Node-API 平台包。
- 签名的 apt 仓库在 `stable` suite 中，为 Debian 和 Ubuntu 的 `amd64` 与 `arm64` 发布 `ait-native` bundle 和独立的 `ait-runner` 包。
- GHCR 为 `ait-server` 和 `ait-runner` 发布 Linux `amd64` 与 `arm64` 索引。

包的构成

路线	已接纳的组件内容
GitHub Release	家族清单接纳的六个原生目标三元组下的全部七个组件。
Homebrew `ait-native`	`ait` 和 `ait-server`；安装 server 不会启动服务。
PyPI `ait-native==1.1.1`	`ait`、`ait-server` 和 `ait-python`。
npm `@wa120/ait-native@1.1.1`	`ait-node` 外观层加上匹配的平台插件；不含 server 可执行文件。
apt `ait-native`	`ait` 和 `ait-server`；安装 server 后服务仍处于未启动状态。
apt `ait-runner`	独立的 `ait-runner`。

路线	已接纳的组件内容
GHCR	独立的 `ait-server` 和 `ait-runner` 镜像。

可选的 `ait-agent` 和 `ait-agent-worker` 可执行文件可以从 GitHub Release 获取。它们与主要的本地编码客户端体验是分开的，正常的 `ait` 工作流并不 需要它们。

发布状态

GitHub Release、PyPI、全部七个 npm 包、Homebrew formula、签名的 apt `stable` suite，以及两个带 attestation 的 GHCR 镜像，都已针对 stable `1.1.1` 发布并独立回读验证过。每条渠道的默认路线（GitHub latest release、PyPI 最新版本、npm `latest` dist-tag、stable formula 与 suite） 都解析到 `1.1.1`，所以不锁版本的安装拿到的就是 stable 版本。

许可证边界

`ait`、`ait-agent`、`ait-agent-worker`、`ait-runner`、`ait-python` 和 `ait-node` 使用 Apache-2.0。`ait-server` 使用 AGPL-3.0-only。打成一个 bundle 既不会合并这些许可证，也不会抹掉各组件的声明。

确切的发布证据

页面的权威面板给出了不可变的 `v1.1.1` 源码 tag、GitHub Release、中心分发约定和确切的解析器源码链接。版本清单还锁定了五个源 Snapshot，以及 两份源文档的 SHA-256。复现构建或验证已安装的可执行文件时，用这些权威 和带校验和绑定的发布资产。

参考

故障排查

先用只读证据，保住可恢复的状态，并照着 AIT 给出的确切恢复提示走。

适用人群 开发者与运维

https://ait-native.dev/technical/v1.1.1/troubleshooting/

先看只读证据

这些恢复步骤针对的是锁定版本的 1.1.1 命令面。

别去猜工作区、Task 或远程是不是健康的。先跑一条和症状对得上的窄范围 只读命令。

```
ait status --json
ait diff --stat
ait queue summary
ait worktree status --json
```

把确切的报错和下一步提示原样留着。AIT 的决策面是有意把那条能安全推进 或安全查看当前状态的命令报出来的。

仓库没有初始化

在打算作为仓库根的目录下执行 `ait init`。如果 `.ait` 目录已经存在但不
完整，别盲目替换它；先看报错，只有在确实对得上所报状况时，才使用那个 有界的修复选项。

某个回归需要定位归属

选定修复方案之前，先对最小的有用源码范围跑 `ait blame`。

```
ait blame <path> --line <number>
```

把修复记进新的 sprint 条目和 Task。已完成的沿革仍然是历史证据。

Task 工作区丢失或过期

重新物化之前，先查 Task 和 worktree。

```
ait task audit <task-id>
ait worktree doctor --refresh --json
```

照着那个 Task 报出来的确切恢复命令走。保留无关的工作区改动，别删任何 没被精确认定为 AIT 所有的路径。

远程就绪一直卡着

再跑一次纯文本的就绪查看面，照它写明的关卡走。

```
ait workflow ready <change-id> --apply
```

本地测试通过，替代不了必需的远程 CI。评审、策略、attestation 或 CI 关
卡挂了，就得一直显示出来，直到对应的证据被修好为止。

server 或 runner 不可用

检查配置的远程、server 健康状况、仓库索引、runner 兼容性，以及仓库自己 持有的 CI
入口。别为了绕开故障，就把一个有远程支撑的 Task 切到另一个 权威上去。

上报时别毁掉证据

当恢复需要用户做新决定时，就地停下，保持当前 Task、Snapshot 和 worktree
原样。报出确切的标识符、失败的关卡、命令输出，以及上一次成功的验证。

附录

附录：配置文件

把每一个公开的 1.1.1 AIT 配置文件对应到它的用途、归属、写入边界和详细 schema 参考。

适用人群 开发者、运维、集成方与发布工程师

https://ait-native.dev/technical/v1.1.1/appendix/configuration-files/

配置文件一览

1.1.1 用一小组仓库文件来承载持久的用户配置。这些文件各自是独立的约定：一个文件里的值，不会悄悄取代另一个文件所拥有的权威。

文件	用途	常规归属
`.ait/config.json`	仓库身份、Line 与远程路由、工作流默认值、Task worktree 放置位置，以及有界的运行时提示。	`ait init`、`ait config`、`ait remote`，以及拥有它的那条 AIT 生命周期。
`.ait/policy.yaml`	由 Repository 策略档位选定的接纳要求。	`ait init`；改动要和 `.ait/config.json.policy_profile` 一起评审。
`.aitignore`	面向工作区、status、Snapshot 和 Plan 文件系统发现的仓库相对可见性规则。	仓库作者。
`ci/patch_ci.json`	远程就绪所用的 Patchset CI 套件目录。	仓库作者；`ait remote add` 可以创建一个起步文件，但不会覆盖已有文件。
`ci/config.contract.json`	可选的本地定位器，指向非默认的 CI 套件目录。	仓库作者。它不能替代远程注册时要求的那份规范目录。
`ait-external.toml`	直接的外部 Repository 声明和语言绑定。	仓库作者。

文件	用途	常规归属
`ait-external.lock`	解析出的确切外部依赖图和 Snapshot 锁定。	`ait external update`；生成结果要提交。
`ait-external.links.toml`	外部物化的本地开发覆盖项。	`ait external link` 和 `ait external unlink`；不要用于可发布的物化。
`.ait/agent-workers.json`	具名的 Telegram、LINE、Discord 和 Slack agent worker 实例。	`ait-agent <transport>` 命令，或者按 schema 操作的运维人员。
`ait-release.json`	通用命令适配器的发布包、组件、命令和产物。	发布作者。
`ait-release-family.json`	协调一致的 AIT 原生家族发布矩阵。	AIT release family 维护者；这是一份专用的 AIT 家族约定。
`.ait-worktree.json`	一个受管 worktree 的叠加层和绑定元数据。	仅由 AIT 维护。

1.1.1 里，server 没有单独的公开配置文件 `ait-server.json`、`ait-server.toml` 或 `.ait/server.*`。它支持的运维输入是命令行选项和 写明的环境变量约定。

我该改哪个文件？

- 优先用拥有该字段的那条命令。它能校验取值，并保住生命周期不变量。

- 只手改作者持有的清单文件，比如 `.aitignore`、`ci/patch_ci.json`、 `ait-external.toml` 和各类发布清单。
- 把 `ait-external.lock`、`ait-external.links.toml` 和 `.ait-worktree.json` 当成生成的或由命令托管的产物。
- 只要有写明的流程或 worker 环境文件入口可用，就别把凭据放进纳入版本 管理的文件里。
- 启动 Task 或做远程注册之前，先校验 JSON、YAML 或 TOML 语法。未知字段的处理方式各约定不同，各自的参考页里都有说明。

哪些东西是有意排除在外的?

运行时状态、缓存、Binary DB 权威、激活回执、生成的产物、包管理器清单、 构建工具配置和 agent 指令文件，都不属于 AIT 自身的用户配置。它们各有 归属，不能从这份附录里推断出来。

相关参考

- 仓库与 worktree 配置
- 策略与忽略规则
- Patchset CI 配置
- 外部 Repository 配置
- Agent worker 配置
- 发布清单
- 环境变量

附录

附录：.ait/config.json

当前 1.1.1 持久化 Repository 配置的完整约定，包含归属、默认值、生命周期写入方、worktree 覆盖项与恢复边界。

> 适用人群 开发者、运维与 Repository 所有者

https://ait-native.dev/technical/v1.1.1/appendix/configuration-file/

约定与权威

`.ait/config.json` 是 1.1.1 里权威的、持久化的 Repository 配置。它保存仓库身份与路由、工作流默认值、远程注册信息、Task worktree 放置位置，以及 一小组由 AIT 托管的运行时提示。这一页把当前支持的每个字段和嵌套字段都 列全。

有三个 JSON 面是相关的，但彼此不能互换：

面	权威与行为
`.ait/config.json`	持久化的仓库级权威。AIT 从规范的 Repository 根目录读这个文件。
`.ait-worktree.json`	某个已物化 worktree 里由 AIT 托管的叠加层。非 null 的叠加值只在那个 worktree 的生效运行时里替换对应的根值。
`ait config show --json`	经过默认值、推导、actor 检测、远程解析和 worktree 叠加之后的生效诊断投影。它不是这两个文件中任何一个的逐字节呈现。

JSON 读取器在改写某个设置时，可能会把不认识的 key 原样保留下来。保留并 不等于支持。别把任意 key 当成扩展点。

安全的修改边界

只要 1.1.1 提供了拥有某个值的那条命令，就用那条命令：

```
ait config show --json
ait config set --workflow-mode solo_remote
ait config set --id-namespace-prefix W
ait config unset task-worktree-main-seed-ram-max-bytes
ait remote add origin https://server.example.invalid --default
ait doctor memory-root --json
```

`ait config set` 接受十项用户持有的设置，`ait config unset` 接受八项可选 覆盖项。仓库索引、Line 选择器、远程记录、推导出来的作用域、Plan 绑定的 内部细节、worktree 叠加层和等待提示，都不能通过 `ait config set` 写。让 拥有它们的那条 AIT 生命周期去写。

如果确实要维护某个只能改文件的高级值，就先停掉并发的 AIT 变更，留一份 可信副本，原子地写入合法 JSON，验证结果之后再继续干活。绝对不要照抄别的 Repository 的 `repository_index`、远程映射或 worktree 叠加层。

有代表性的根文件

这份中性样例只是展示常见形状；它不是恢复用的模板。server 分配的索引、远程元数据、各种路径和身份，都必须来自这个 Repository 自己的生命周期。

```
{
  "repo_name": "sample-project",
  "default_line": "main",
  "current_line": "main",
  "default_remote": "origin",
  "repository_index": 42,
  "remotes": {
    "origin": {
      "remote_id": 1,
      "url": "https://server.example.invalid",
      "repo_name": "sample-project",
      "created_at": "2026-08-17T08:00:00Z"
    }
  },
  "id_namespace_prefix": "W",
  "policy_profile": "prototype",
  "default_author_mode": "ai_with_human_review",
  "workflow_mode": "solo_remote",
  "workflow_default_scope": "remote",
  "task_default_scope": "remote",
  "change_default_scope": "remote",
  "sprint": "on",
  "plan_task_binding": { "mode": "required" },
  "task_worktree": {
    "alias_root": ".ait-worktree-links",
    "main_seed_ram_max_bytes": 8589934592
  }
}
```

全新初始化

`ait init` 会创建 `repo_name`、`default_line`、`current_line`、一个为 null 的 `default_remote`、一个空的 `id_namespace_prefix`、`policy_profile`、`default_author_mode`、`sprint` 和 `plan_task_binding`。在受支持的宿主上， 它还能记录检测到的 `task_worktree.memory_root`。可选的默认模型之后用 `ait config set --default-model <model>` 设置。

默认 Line 是 `main`。没有明确给出 Repository 名时，用规范根目录的目录名。 全新的策略档位是 `prototype`、`team` 或 `release` 之一；它必须和 `.ait/policy.yaml` 一致。

仓库身份与 Line 选择

字段	JSON 约定	归属、缺省与修改方式
`repo_name`	非空字符串。	由 `ait init` 写入。老仓库里如果缺这个字段，运行时回落到规范根目录的目录名。它不是远程身份。
`default_line`	非空的 Line 名字符串。	由 `ait init` 写入；缺省时回落到 `main`。改动由仓库初始化流程负责。
`current_line`	非空的 Line 名字符串。	由 AIT 的 Line 操作维护，表示规范根上选中的 Line。缺省时回落到 `default_line`。
`default_remote`	远程名字符串或 `null`。	由 `ait remote add ... --default` 写入。非 null 的值必须对应 `remotes` 里的某一项；缺省表示没有默认远程。
`repository_index`	无符号 32 位整数。	由 server 分配，`ait remote add` 负责持久化。一旦存在，1.1.1 就拒绝用新返回的索引替换它。没有公开的 setter。
`id_namespace_prefix`	只含 ASCII 字母或数字的字符串；按大写存储。空值也合法。	`ait init`、`ait config set --id-namespace-prefix` 或 `ait config unset id-namespace-prefix`。当前 server 命名空间最多两个字符；用空值，或者用客户端和 server 都接受的值。

策略、溯源与本地评审人默认值

字段	JSON 约定	归属、缺省与修改方式
`policy_profile`	`prototype`、`team` 或 `release`。	由 `ait init` 选定；它必须等于 `.ait/policy.yaml` 里的策略 ID。别只改一边。
`default_author_mode`	`human_only`、`human_with_ai_assist`、`ai_with_human_review` 或 `ai_only_experimental`。	`ait init`、`ait config set` 或 `ait config unset`。缺省时解析为 `ai_with_human_review`。
`default_model`	存在时为非空字符串。	`ait config set --default-model`；`ait config unset default-model` 会删掉它。缺省时报告为 null。
`user_name`	存在时为非空字符串。	`ait config set` 或 `unset`。这是必需评审或自动功能性 Task 评审所用的唯一配置身份。
`user_email`	存在时为非空字符串。	`ait config set` 或 `unset`。用于非 Task 评审的 actor 身份展示面。
`task_review`	布尔值：`true` 表示必需；`false` 表示自动。	`ait config set --task-review required` 或 `ait config set --task-review automatic` 会把这个词转成上面的布尔值。缺省时解析为自动。

生效的 actor 也可能来自 `AIT_NATIVE_ACTOR`。所以 `ait config show --json` 显示出来的生效值，可能和存下来的身份字段不一样。

工作流预设的联动

设整套预设，别单独去改它下面那些从属字段：

`workflow_mode`	存储的作用域值
`solo_local`	`workflow_default_scope`、`task_default_scope` 和 `change_default_scope` 都是 `local`。
`solo_remote`	`workflow_default_scope`、`task_default_scope` 和 `change_default_scope` 都是 `remote`。

`ait config set --workflow-mode <value>` 会写 `workflow_mode` 和那三个作用域 字段。除非同一条命令里带了 `--sprint off`，否则它还会写 `sprint: "on"` 和 `plan_task_binding.mode: "required"`。

字段	JSON 约定	归属、缺省与修改方式
`workflow_mode`	`solo_local`、`solo_remote` 或 `team_remote`。	`ait config set --workflow-mode`。缺省或不自洽时，生效投影由各作用域和 Plan 绑定推导得出，可能报成 `custom`。
`workflow_default_scope`	`local` 或 `remote`。	由工作流预设写入。缺省时解析为 `local`。
`task_default_scope`	`local` 或 `remote`。	由工作流预设写入。缺省时继承 `workflow_default_scope`。
`change_default_scope`	`local` 或 `remote`。	由工作流预设写入。缺省时继承 `workflow_default_scope`。
`sprint`	`on` 或 `off`。	`ait init` 或 `ait config set --sprint`。只要它存在，它就是 Plan 绑定的生效权威。
`plan_task_binding.mode`	`off`、`advisory`、`strict` 或 `required`。	当前的公开写入方在 sprint 开启时写 `required`，关闭时写 `off`。如果 `sprint` 缺省，就由这个对象提供生效模式；两者都缺省时，1.1.1 会预置 `required`。

改这些默认值，不会搬动或改写已有的 Task、Change、Plan 绑定或 worktree。
已经开始的工作，按它开始时记录下来的那份约定走完。

远程映射

`remotes` 是一个以本地远程名为 key 的对象。它的公开写入方是 `ait remote add`。

字段	JSON 约定	归属与行为
`remotes`	key 为非空远程名的对象。	缺省表示没有注册过远程。每个 value 都必须是对象。
`remotes.<name>.remote_id`	有符号整数。	AIT 分配下一个本地序号。老条目没有这个字段时，按它在映射中确定性的序号读取，但当前写入都会带上它。
`remotes.<name>.url`	非空字符串。	传给 `ait remote add` 的必填 server 基础 URL。
`remotes.<name>.repo_name`	非空字符串，或缺省。	AIT 记录用于 server 注册的规范 Repository 目录名。
`remotes.<name>.created_at`	RFC 3339 时间戳字符串。	AIT 记录注册时间。老记录可能没有。

远程 URL 可能暴露内网拓扑。它是路由数据，不是凭据。认证 token 属于进程的 机密边界，不该放在这个文件里。

`task_worktree` 对象与 `AIT_RAM`

字段	JSON 约定	归属、缺省与修改方式
`task_worktree`	对象或 `null`。	`ait init` 可以补上检测到的 memory root 数据。缺省时仍然允许宿主探测和回落到持久磁盘。
`task_worktree.memory_root.kind`	`macos_ram_volume`、`linux_memory_root` 或 `windows_ramdisk`。	必须和宿主一致，也要和 `ait doctor memory-root --json` 期望的证明一致。
`task_worktree.memory_root.root`	非空路径字符串。	内存支撑的根目录。由运维维护；用绝对路径，诊断和恢复时才不会有歧义。
`task_worktree.memory_root.volume_name`	非空字符串，或缺省。	仅 macOS 用的卷标。其他 kind 会忽略它。macOS 上缺省时，AIT 可以从挂载点的 basename 推导。
`task_worktree.memory_root.sector_count`	正整数，或缺省。	macOS RAM 盘容量，以 512 字节扇区计。缺省时用 16,777,216 个扇区，正好 8 GiB。其他 kind 会忽略它。
`task_worktree.ephemeral_root`	非空路径字符串，或缺省。	只能改文件的 Task 运行时根目录。相对路径从规范的 Repository 根解析。缺省时，只要有可用的有效 memory root，就据此推导运行时位置。

字段	JSON 约定	归属、缺省与修改方式
`task_worktree.alias_root`	非空路径字符串，或缺省。	`ait config set --task-worktree-alias-root`；相对路径从 Repository 根解析。缺省时用 `.ait-worktree-links`。
`task_worktree.main_seed_ram_max_bytes`	非负整数，或缺省。	`ait config set` 或 `ait config unset task-worktree-main-seed-ram-max-bytes`。缺省表示没有配置 Snapshot 大小的截断线；它不代表物理内存无上限。

内存放置是在创建新的 Task worktree 时评估的。严格大于所配置 main seed 上限 的 Snapshot，会走 Repository 内部的持久存储。除此之外，AIT 能用就用经过 校验的受支持 RAM 根，用不了就回落到 Repository 内部的持久存储。完整的 隔离、容量和恢复行为见 并行 Task 隔离。

由 AIT 托管的运行字段

字段	JSON 约定	归属、缺省与行为
`worktree_name`	非空字符串，或缺省。	AIT 会临时把规范根绑定到某个活动的受管 worktree，并在清理时解除绑定。别手改它。
`workflow_ready_poll_seconds`	数字。	AIT 从最近的远程历史里学出一个取整后的就绪等待估计值。为正的生效值会被夹到 5 到 900 秒之间；`0` 表示尝试过引导但没有可用样本。缺省表示没有缓存的估计值。
`workflow_land_poll_seconds`	数字。	同样是 AIT 持有的估计值，用于远程 Land 完成。它有同样的 5 到 900 秒生效边界和 `0` 引导标记。

这些等待值只是给用户反馈用的提示，不是就绪、策略或超时方面的权威。删掉 或改掉它们，并不会让某个操作完成。

已有文件里的兼容字段

这些字段可以留在升级过来的仓库里。这里写出来是为了让备份或审计看得懂； 别把它们加进全新的 1.1.1 文件。

字段	JSON 约定	当前 1.1.1 里的含义
`repo_id`	早期的不透明字符串。	保留的兼容数据。当前 server 的路由身份是 `repository_index`；不得用 `repo_id` 来选定远程权威。
`snapshot_binary_db_storage`	已有的标量值。	当前运行时一律使用 Binary DB layout 1，忽略这个选择器的取值。不写它，存储行为完全一样。
`plan_binary_db_storage`	已有的标量值。	对 Plan 权威同样是固定的 layout 1 行为。
`remote_sync_binary_db_storage`	已有的标量值。	对远程同步权威同样是固定的 layout 1 行为。
`plan_task_binding_mode`	早期的标量模式。	只被一条有界的兼容路径使用。当前配置读写的是 `plan_task_binding.mode`；别把这个标量加进新文件。

完整的 worktree 叠加层

AIT 会在受管 worktree 里创建 `.ait-worktree.json`，并把其中每个非 null 的 值叠加到根配置之上，只对那个 worktree 生效。Task worktree 会用到下面这些 字段：

字段	JSON 约定	含义
`worktree_name`	非空字符串。	稳定的、AIT 持有的物化名称和清理身份。
`current_line`	非空的 Line 名字符串。	在这个 worktree 里选中的 Task 功能 Line。
`repo_root`	绝对路径字符串。	规范的 Repository 根，它的 `.ait` 权威通过受管链接共享出来。
`workspace_root`	绝对路径字符串。	物理 worktree 根。它可以和面向用户的别名路径不同。
`created_at`	RFC 3339 时间戳字符串。	物化时间。
`materialized_snapshot_id`	Snapshot ID 字符串，或缺省。	当前恢复进这个 worktree 的 Snapshot；AIT 会在 restore 和 rebase 操作之后更新它。

这个叠加层是归属元数据，不是可搬运的用户画像。别复制它、别编辑它，也别 把受管的 `.ait` 链接换成另一个权威。

验证、备份与恢复

做完一次受支持的改动之后，把生效投影和相关子系统都验一遍：

```
ait config show --json
ait remote list --json
ait status --json
ait doctor memory-root --json
```

备份整个 `.ait` 权威，不要只备份 `config.json`。这个文件里有 server 分配的仓库索引和远程路由，把本地权威重新接回已有的异地副本要靠它们。分享 诊断信息之前，先把 server URL、本地路径、姓名和邮箱地址脱敏掉。这个文件 里不应该出现密码、bearer token 或集成用的机密。

万一本地权威丢了，别指望对着一个同名的已有 server Repository 跑 `ait remote add` 就能重新挂上。作为已保存本地权威的一部分，先恢复可信的 `repository_index`、`default_remote` 和 `remotes` 路由数据，用 `ait config show --json` 和 `ait repo show --remote origin --json` 验证， 然后先预览 `ait remote recover-head --remote origin`，再考虑加 `--apply`。如果确切的已保存路由恢复不了，就停下来，别去造一个替代权威。

相关

- 配置文件一览
- `ait config`
- 完整的 `ait config` 命令参考
- 并行 Task 隔离
- 远程基础设施
- 附录：环境变量

附录

附录：策略与忽略规则

用确切接受的字段和匹配规则，在 .ait/policy.yaml 里配置 1.1.1 的接纳要求，在 .aitignore 里配置仓库可见性。

适用人群 Repository 所有者、开发者与 coding agent

https://ait-native.dev/technical/v1.1.1/appendix/repository-policy-ignore/

仓库策略与忽略规则

这一页说明位于仓库根目录、控制接纳要求和文件系统可见性的两个文本文件。
它们解决的是不同的问题：`.ait/policy.yaml` 决定需要哪些证据；`.aitignore` 决定 AIT 能看到哪些工作区路径。

`.ait/policy.yaml`

`ait init` 会用仅所有者可读写的权限创建这个文件。它的 `policy_id` 必须 和 `.ait/config.json.policy_profile` 一致。1.1.1 支持的策略 ID 有 `prototype`、`team` 和 `release`。

```
version: 1
policy_id: prototype
defaults:
  require_attestation: true
  require_tests: true
  require_lint: false
  require_security_scan: false
  require_license_scan: false
  require_ai_provenance: false
  require_code_review_summary: false
class_overrides:
  - when:
      content_class: docs_only
    set:
      require_tests: false
      require_lint: false
      require_security_scan: false
      require_license_scan: false
```

根字段与要求字段

字段	类型与含义
`version`	整数 `1`。
`policy_id`	`prototype`、`team` 或 `release` 之一；必须等于 Repository 的策略档位。
`defaults`	必填映射，包含各项基线要求开关。
`class_overrides`	可选的有序列表，存放按条件生效的部分覆盖项。空列表也合法。

每个要求项的值都是确切的 YAML 布尔值：

要求项	所控制的证据
`require_attestation`	需要一份已接纳的 attestation。
`require_tests`	测试证据必须通过。
`require_lint`	Lint 证据必须通过。
`require_security_scan`	安全扫描证据必须通过。
`require_license_scan`	许可证扫描证据必须通过。
`require_ai_provenance`	需要 AI 溯源证据。

要求项	所控制的证据
`require_code_review_summary`	需要一份代码评审摘要。

覆盖项的 `set` 映射里接受的也是这同样的七个名字。`defaults` 必须七个都 写全；`set` 是部分的，只改它写到的那几个字段。

覆盖项选择器

`class_overrides[]` 的每一条都要求有非空的 `when` 映射和非空的 `set` 映射。`when` 接受下面这两个选择器中的一个或两个：

选择器	可取值
`content_class`	`docs_only`、`code_change`
`author_class`	`human_only`、`ai_related`

覆盖项按文件里的先后顺序求值。条件要写明确，别让几条互相重叠、结果取决 于排序的条目混在一起。

生成的档位基线

三个生成出来的档位都要求 attestation 和测试，并且把 AI 溯源和代码评审 摘要这两项要求关掉。它们其余的默认值是：

档位	Lint	安全扫描	许可证扫描
`prototype`	false	false	false
`team`	true	false	false
`release`	true	true	true

生成的「仅文档」覆盖项会把测试、lint、安全扫描和许可证扫描都关掉。改动生成的策略就等于改动接纳策略；要和对应的档位设置一起评审。

YAML 接受规则

- 用空格，别用 tab，缩进按两个空格一级。
- 用确切的小写 `true` 和 `false`。
- 重复字段和未知字段一律校验失败。
- `version`、`policy_id`、`defaults` 以及任何覆盖项映射，都要保持在上面 示例的确切嵌套层级上。
- 1.1.1 里首次远程注册只接纳确切的 `prototype` 基线，以及它完整的 「仅文档」覆盖项。注册新 Repository 之前先选好这个档位。本地配置的 `team` 和 `release` 档位，在受支持的本地用途下仍然是有效的策略文件。

`.aitignore`

`.aitignore` 是一份按行组织的仓库可见性约定。它作用于工作区检查，以及 status、Snapshot 和 Plan 在文件系统发现阶段会考虑的那些文件。它是对 AIT 内置排除规则的补充，不是替代。

```
# Generated outputs
dist/
*.tmp

# Keep one fixture visible
!fixtures/expected.tmp

# Literal leading markers
\#notes.txt
\!important.txt
```

规则语法

形式	行为
空行	忽略。
`# comment`	忽略。
`\#name`	匹配开头是字面量 # 的名字。
`!pattern`	取消前面的匹配，让该路径重新可见。
`\!name`	匹配开头是字面量 ! 的名字。
`./path`	开头的 `./` 会被去掉。

形式	行为
`/path`	锚定到 Repository 根。
`path/`	只匹配目录的规则。
`name`	不含斜杠时，在任意深度上匹配这个 basename。
`*`	匹配一个路径段内的任意字符序列。
`?`	匹配一个路径段内的单个字符。

规则从上往下求值，最后一条命中的规则说了算。其他正则字符按字面量处理； 字符类语法不在 1.1.1 的匹配器范围内。`.aitignore` 自身始终可见，这样它 的效果才能被记录和评审。

忽略规则改变的是可见性，不是归属或删除。它不会抹掉已有的 Snapshot，不 构成存放机密的许可，也不会让某个生成目录变得可以放心提交。

验证

改完这两个文件中的任何一个之后，先做只读检查，再开始正常干活：

```
ait config show --json
ait status
ait diff
```

做远程注册时，要等策略和 Patchset CI 目录都就绪之后，再真正执行 `ait remote add`；它的校验才是权威的。

相关参考

- 配置文件一览
- 仓库与 worktree 配置
- Patchset CI 配置

附录

附录：Patchset CI 配置

定义规范的 1.1.1 Patchset CI 套件目录、支持的 runner、检查项、产物，以及有界的本地定位器。

适用人群 Repository 所有者、CI 维护者与运维

https://ait-native.dev/technical/v1.1.1/appendix/patchset-ci/

Patchset CI 配置

`ci/patch_ci.json` 是 1.1.1 的规范套件目录，在 Repository 向 AIT server 注册时、以及远程就绪要跑 Patchset CI 时使用。目录里至少要有一个阻塞式的 Patchset 关卡。

最小的命令包目录

```
{
  "schema_version": 1,
  "suites": [
    {
      "schema_version": 1,
      "suite_id": "patchset_gate",
      "display_name": "Patchset Gate",
      "plane": "patchset",
      "mode": "gate",
      "default_blocking": true,
      "purpose": "Validate this repository before remote land.",
      "runner": {
        "kind": "command_bundle",
        "commands": ["./ait.sh test"]
      },
      "artifacts": {
        "log_path": ".ait/generated/ci/patchset_gate.log",
        "summary_json": ".ait/generated/ci/patchset_gate.json"
      }
    }
  ]
}
```

文件不存在时，`ait remote add` 会创建一份起步目录，然后停下来，好让你把占位命令替换掉。它绝不会覆盖已有的目录。占位值 `replace-with-your-ci-command` 会被拒绝。

目录与套件字段

字段	约定
`schema_version`	顶层整数 `1`；必填。起步文件也会在每个套件上写一个 `1`。
`suites`	套件对象数组。其中至少要有一个具名、可运行的阻塞式 Patchset 关卡。
`suites[].suite_id`	非空且唯一的操作名。每个被选中的阻塞关卡都必须有。
`suites[].display_name`	可选的、给人看的字符串。
`suites[].plane`	用于 Patchset CI 选择时填 `patchset`。
`suites[].mode`	用于就绪判定时填 `gate`。

字段	约定
`suites[].default_blocking`	布尔值。`true` 让选中的套件成为就绪证据的一部分；`false` 仅供参考。
`suites[].purpose`	可选的说明字符串。
`suites[].runner`	必填对象，其 `kind` 必须非空且受支持。
`suites[].artifacts.log_path`	可选的日志产物相对路径。
`suites[].artifacts.summary_json`	可选的 JSON 摘要相对路径。

只有 `plane` 和 `mode` 解析为 `patchset` 与 `gate` 的套件才会被编排。用于
首次注册的目录里，至少要有一个这样的套件，并且 `default_blocking: true`。 不要把未知的 key 当成扩展点来用。

`command_bundle` runner

这个 runner 按顺序执行 shell 命令字符串，第一次失败就停。预热命令和主命令 共用同一份接纳的环境。

Runner 字段	类型与行为
`kind`	确切字符串 `command_bundle`。
`commands`	非空数组，元素是非空的 shell 命令字符串。
`prewarm_commands`	可选数组，在 `commands` 之前执行。
`env`	可选对象，把字符串环境值加进干净的 CI 进程环境里。
`runner_parallelism`	可选的正整数，用作接纳的命令并行度。
`cpu_tokens`	`runner_parallelism` 的别名。

Runner 字段	类型与行为
`workers`	`runner_parallelism` 的低优先级别名。
`timeout_seconds`	可选的正数有界超时，对每个进程分别生效。

调度器可以给出更严格的接纳并行度。光靠配置本身并不会分到 CPU 容量。完整 日志和有界的 JSON
摘要是两个分开的产物。

`test_discovery_sharded` runner

这个 runner 会发现测试用例、构建选中的测试可执行文件，并跑有界的分片。它 支持 Cargo 和一个通用命令适配器。

通用字段

Runner 字段	类型与行为
`kind`	确切字符串 `test_discovery_sharded`。
`adapter`	默认 `cargo`；填 `command` 选用通用适配器。
`env`	可选的字符串到字符串环境对象。
`runner_parallelism`、`cpu_tokens`、`workers`	正整数并行度的三个名字，优先级依此顺序。
`timeout_seconds`	可选的正数有界进程超时。
`exclude_test_cases`	可选数组，列出要略过的确切已发现测试用例名。

Runner 字段	类型与行为
`checks`	可选的测试前检查数组，见下文。

Cargo 适配器字段

Runner 字段	类型与行为
`cargo_binary`	Cargo 可执行文件名或路径；默认 `cargo`。
`manifest_path`	规范化的相对路径；默认 `Cargo.toml`。
`workspace`	布尔值；默认 `true`。
`build_args`	可选的 argv 字符串，用来替换默认的 Cargo 构建参数。
`doc_tests`	布尔值；默认 `false`。命令适配器下不接受这个字段。
`doc_test_args`	可选的 argv 字符串，用于文档测试。

命令适配器字段

Runner 字段	类型与行为
`discovery_program`	必填，非空的可执行文件名或路径。
`discovery_args`	可选的发现阶段 argv 数组。
`discovery_output_format`	默认 `json_array`，也可以是 `lines`。
`run_program`	必填，非空的可执行文件名或路径。
`run_args`	可选的运行阶段 argv 数组。
`append_test_items`	布尔值；为 true 时，把发现到的用例名追加到运行 argv 后面。默认 `false`。

Runner 字段	类型与行为
`working_directory`	工作区内规范化的相对目录；默认是工作区根。该目录必须存在。

测试前检查

`checks` 的条目接受 `check_id`、`kind`、`file_name_suffix`、`exclude_dirs` 和 `args`。缺少 `check_id` 时会变成 `check-N`。

- `kind: "forbid_files"`：只要有文件以 `file_name_suffix` 结尾、且不在指定的目录段之内，就判失败。该后缀默认是 `.py`。
- `kind: "cargo_fmt"`：跑 Cargo 格式化校验。`args` 可以替换它默认的 argv。

套件运行时，其他种类的 check 会被拒绝。

可选的构建缓存

`runner.build_cache` 接受 `policy` 和 `executable_manifest_path`。唯一能开启 缓存的 policy 是 `reuse_when_rust_inputs_unchanged`；不写或写别的值都表示
复用关闭。变更路径的证据由执行请求提供，不应该当成静态目录数据写进去。

可选的本地定位器：`ci/config.contract.json`

当 `ci/patch_ci.json` 不存在时，本地工作流的命令发现可以读取：

```
{
  "schema_version": 1,
  "ci": {
    "suite_manifest_path": "config/patchset-suites.json"
  }
}
```

1.1.1 只读取非空的 `ci.suite_manifest_path` 值，并且只在被引用的文件确实 存在时才用它。只要 `ci/patch_ci.json` 在，就总是它说了算。远程 Repository 注册依然会校验并发布那份规范的 `ci/patch_ci.json`；这个定位器不会改变注册 时的约定。

运维检查

```
ait remote add origin https://server.example.invalid --default
ait workflow ready <change-id> --apply
ait task audit <task-id>
```

`ait workflow ready` 是当前 Patchset 的决策面。本地跑通一条测试命令，替代 不了有远程支撑的 Task 所要求的 server 证据。

相关参考

- 配置文件一览
- 仓库策略与忽略规则
- 远程基础设施

附录

附录：外部 Repository 配置

通过三份 1.1.1 TOML 约定声明、锁定、物化、绑定并在本地覆盖外部 Repository。

适用人群 开发者、集成方与发布工程师

https://ait-native.dev/technical/v1.1.1/appendix/external-repositories/

外部 Repository 配置

1.1.1 把作者意图、确切解析结果和本地开发覆盖项，拆成三个 TOML 文件：

文件	权威
`ait-external.toml`	由 Repository 编写的直接声明。
`ait-external.lock`	解析完成的完整依赖图，带确切的 Snapshot 锁定；由 AIT 生成。
`ait-external.links.toml`	由 AIT 命令管理的本地路径覆盖项。

`ait-external.toml`

每一条 `[[external]]` 指明一个直接的外部 Repository，以及它的内容物化到 哪里。

```
[[external]]
name = "shared-core"
repo_name = "shared-core"
repository_index = 23
remote = "origin"
line = "main"
snapshot = "SNP-0123456789AB"
materialize_to = ".ait-external/shared-core"
license = "Apache-2.0"
version = "1.2.0"

[external.bindings.rust]
kind = "cargo-path"
path = "rust/crates/shared-core"
package = "shared-core"

[external.bindings.python]
kind = "python-path"
path = "python"
package = "shared-core"
module = "shared_core"
```

声明字段

字段	类型与含义
`name`	必填，非空的本地外部名。它用来标识这条声明和对应的 link 覆盖项。
`repo_name`	必填，非空的源 Repository 名。
`repository_index`	必填，无符号 32 位的 server Repository 索引。
`remote`	必填，非空的已配置远程名，用来解析源。
`line`	必填，非空的源 Line。
`snapshot`	必填，非空且确切的源 Snapshot ID。这是锁定，不只是一个分支提示。

字段	类型与含义
`materialize_to`	必填，规范化的仓库相对目标路径。绝对路径和 `..` 穿越会被拒绝。
`license`	必填，非空的已声明许可证表达式或标签。
`version`	可选，非空的、给人看的版本字符串。内容权威仍然是 Snapshot。
`bindings`	可选映射，每种受支持的语言最多一条绑定。

直接名称必须无歧义。物化路径和绑定路径会按仓库相对路径校验，不能逃出 它们所属的根目录。

绑定字段

表	必填字段	可选元数据
`[external.bindings.rust]`	`kind = "cargo-path"`、`path`	`package`
`[external.bindings.python]`	`kind = "python-path"`、`path`	`package`、`module`
`[external.bindings.node]`	`kind = "file-package"`、`path`	`package`
`[external.bindings.go]`	`kind = "replace-path"`、`path`	`module`

每个 path 都是相对于该外部 Repository 物化位置的。元数据只要写了就必须
非空。绑定校验还会检查相应的语言工具和依赖文件；光是 TOML 形状合法， 并不能证明这条包绑定真的能用。

`ait-external.lock`

不要凭空造这个文件，也别随手手改。`ait external update` 会解析直接和
传递的外部依赖、规范化顺序、校验依赖图，并原子地写入 lock。

```
format = "ait.external.lock"

[[node]]
name = "shared-core"
repo_name = "shared-core"
repository_index = 23
remote = "origin"
line = "main"
snapshot = "SNP-0123456789AB"
parent_path = ""
materialize_to = ".ait-external/shared-core"
license = "Apache-2.0"
version = "1.2.0"

[[node.binding]]
language = "rust"
kind = "cargo-path"
path = "rust/crates/shared-core"
package = "shared-core"
```

Lock 字段

字段	约定
`format`	确切字符串 `ait.external.lock`。
`[[node]]`	零个或多个规范化的直接与传递依赖图节点。
`node.name`	必填非空名称，与 `parent_path` 组合后唯一。
`node.repo_name`	必填，非空的源 Repository 名。
`node.repository_index`	必填，无符号 32 位的源 Repository 索引。
`node.remote`	必填，非空的远程名。

字段	约定
node.line	必填，非空的源 Line。
node.snapshot	必填，非空且确切的 Snapshot ID。
node.parent_path	直接根节点为空；否则是规范化的相对祖先路径。
node.materialize_to	必填，规范化的相对目标路径。
node.license	必填，非空的许可证标签。
node.version	可选的版本字符串。

字段	约定
[[node.binding]]	零个或多个规范化的绑定摘要。
node.binding.language	rust、python、node 或 go。
node.binding.kind	该语言允许的确切 kind。
node.binding.path	规范化的相对路径。
node.binding.package	可选，非空的包名。
node.binding.module	可选，非空的模块名。

lock 会拿 ait-external.toml 来比对，查出缺失的、多余的和字段漂移的 直接根节点。干净的 lock 必须和清单一起提交，这样每个 worktree 和远程 构建看到的才是同一张依赖图。

ait-external.links.toml

本地 link 让开发者可以临时把某个具名外部从一个已有目录物化出来，而不是 用它锁定的 Snapshot。

```
[[link]]
name = "shared-core"
path = "../shared-core"
```

每一条只有 name 和 path 这两个操作值。name 指向一条外部声明；path 必须能解析到一个已存在的目录。用拥有它的那两条命令：

```
ait external link shared-core ../shared-core
ait external unlink shared-core
```

最后一条 link 被移除时，AIT 会把这个文件删掉。本地 link 属于开发者覆盖 项，在锁定的或可发布的物化过程中会被拒绝。它们从不修改已声明的 Snapshot 或生成的 lock。

更新与验证流程

```
ait external update
ait external status
ait external doctor
ait diff ait-external.toml ait-external.lock
```

像评审一次依赖升级那样评审 lock 的改动：在记录最终的 Snapshot 之前，先 核对 Repository 索引、Line、Snapshot、物化目标、许可证和绑定漂移。

相关参考

- 配置文件一览
- 完整 CLI 参考
- 并行 Task 隔离

附录

附录：Agent worker 配置

配置具名的 Telegram、LINE、Discord 和 Slack worker，以及凭据优先级、运行时路径、传输模式和本地应答行为。

适用人群 Agent 运维与集成维护者

https://ait-native.dev/technical/v1.1.1/appendix/agent-workers/

Agent worker 配置

`.ait/agent-workers.json` 为一个 AIT Repository 声明具名的消息 worker。 1.1.1 支持 `telegram`、`line`、`discord` 和 `slack`。默认位置可以用 `AIT_AGENT_CONFIG_PATH` 覆盖。

先读 ait-agent-worker：消息与回复运行时， 了解这个可执行文件的职责、完整命令面、从消息到回复的路径、传输模式、原生 Codex provider、运行时接纳和安全限制。这份附录是逐字段的编写参考。

规范文档形状

```
{
  "version": 1,
  "workers": {
    "telegram/support": {
      "kind": "telegram",
      "name": "support",
      "mode": "poll",
      "username": "support_bot",
      "env_path": ".ait/agent-runtime/telegram.env",
      "request_timeout_seconds": 30,
      "local_reply": {
        "model": "gpt-5.4-mini",
        "reasoning_effort": "low",
        "sandbox": "workspace-write",
        "turn_timeout_seconds": 300
      }
    }
  }
}
```

作者要写的就是上面那种直接的 `version` 加 `workers` 对象。内部诊断投影 可能会把它包在 `config` 里；别把那层包装写进仓库文件。

worker 的 key 用确切的 `<kind>/<name>` 形式。kind 和 name 都必须非空， name 里不能有 `/`，显式写出的 `kind` 必须和 key 一致。当前约定下 `version` 是整数 `1`。规范化过程中出现问题时，worker 会启动失败并直接 关停。

通用 worker 字段

字段	类型与行为
`kind`	`telegram`、`line`、`discord` 或 `slack`；一般从 key 推导。
`name`	非空的实例名；一般从 key 推导。
`runtime_root`	可选，worker 运行时文件的路径；默认 `.ait/agent-runtime`。相对路径从 Repository 根解析；`~` 在可用时按进程的 home 目录解析。
`sync_state_path`	可选的状态文件路径。默认 `<runtime_root>/<kind>-<safe-name>-sync.json`。
`env_path`	可选的凭据环境文件路径。默认 `<runtime_root>/<kind>.env`。
`web_url`	可选，与回复关联的规范化浏览器基础 URL。

字段	类型与行为
`request_timeout_seconds`	可选的正数请求超时。`inf`、`infinite` 或 `none` 关闭受支持的通用超时；Telegram 还接受 `null` 和 `unlimited`。
`local_reply`	可选的本地回复 provider 对象，见下文。
`created_at`、`updated_at`	由命令托管的时间戳。
`pid_file`、`log_file`、`termination_context_path`	存在时是由命令托管的生命周期路径；别拿它们去配置回复行为。

运行目标从 `.ait/config.json` 推导：`solo_local` 在本地跑；有远程的工作流 则需要一个有效的默认远程 URL。当 worker 没有设置 model 时，Repository 的 `default_model` 提供 Telegram 的模型回落值。

凭据来源与优先级

凭据字段必须是非空字符串。解析顺序是：

1. worker 条目本身；
2. 当前进程环境；
3. worker 的 `env_path` 文件。

其他行为类字段只来自 worker 条目或它们编译期的默认值；env 文件覆盖不了 它们。环境文件接受 KEY=VALUE 行、空行、# 注释，以及值两边配对的一对 单引号或双引号。它们不是 shell 脚本。

别把机密留在提交上去的清单里。优先用仅所有者可读的 env 文件或进程环境， 并使用环境变量附录 里那些确切的变量名。

`local_reply`

这个嵌套对象里的未知字段会被拒绝。

字段	约定
`program`	可选，非空的回复 provider 可执行文件。
`args`	可选的字符串数组，元素中不能含 NUL 字符。
`timeout_seconds`	可选的有限正数。
`append_turn_analysis`	可选的布尔值。
`codex_program`	可选，非空的 Codex 可执行文件覆盖值。
`model`	可选，非空的模型名。

字段	约定
`reasoning_effort`	`none`、`minimal`、`low`、`medium`、`high`、`xhigh`、`max` 或 `ultra`。
`sandbox`	`read-only`、`workspace-write` 或 `danger-full-access`。
`turn_timeout_seconds`	正数、数字字符串，或者 `inf`、`infinite`、`none`、`unlimited`。

sandbox 的取舍是一条执行边界，不是消息偏好。只给这个 worker 完成其 Repository 任务所需的访问权限。

Telegram 字段

字段	约定与默认值
`token`	必填的 bot token，或者提供 `AIT_TELEGRAM_BOT_TOKEN`/`BOT_TOKEN`。
`username`	可选；开头的 `@` 会被去掉。
`mode`	`poll`（默认）或 `webhook`。
`bind_host` / `bind_port`	webhook 监听地址；默认 `127.0.0.1` 和 `8090`。
`webhook_path`	HTTP 路径；默认 `/webhook`。
`webhook_secret`	可选的密钥，或用 `AIT_TELEGRAM_WEBHOOK_SECRET`。

字段	约定与默认值
`poll_timeout_seconds`	整数，至少 5；默认 45。
`background_sync_enabled`	布尔值；默认 false。
`background_sync_interval_seconds`	数字，至少 5；默认 30。
`openai_api_key`	可选的 worker 凭据；占位值会被拒绝。
`openai_base_url`	可选的规范化基础 URL；默认 `https://api.openai.com/v1`。
`openai_model`	可选的模型；先回落到 Repository 默认模型，再回落到 `gpt-5.4-mini`。

字段	约定与默认值
`openai_reasoning_effort`	可选字符串；默认 `low`。
`openai_timeout_seconds`	可选的正数超时；不写时继承请求超时。
`openai_max_output_tokens`	整数，至少 64；默认 700。
`turn_merge_window_seconds`	非负数；默认 0.35。
`turn_merge_max_messages`	正整数；默认 4。
`decoupled_reply_enabled`	布尔值；默认 true。

字段	约定与默认值
`reply_markdown_enabled`	布尔值；默认 true。
`owner_bootstrap_enabled`	布尔值；默认 true。
`stt_mode`	`off`（默认）或 `local-stt`。
`stt_model`	本地语音转文字模型名。
`stt_device`	设备选择器；默认 `auto`。
`stt_compute_type`、`stt_language`	可选字符串。

字段	约定与默认值
`stt_include_audio_uploads`	布尔值；默认 false。
`stt_program`	可选的本地语音转文字可执行文件路径。
`stt_timeout_seconds`	正数，取值从 0.1 到 3600；默认 120。
`expected_concurrent_workers`、`workers_per_shard`	可选的正整数。
`event_loop_backend`	可选的后端字符串。

LINE 字段

字段	约定与默认值
`token`	必填的 channel access token，或用 `AIT_LINE_CHANNEL_ACCESS_TOKEN`/`LINE_CHANNEL_ACCESS_TOKEN`。
`secret`	必填的 channel secret，或用 `AIT_LINE_CHANNEL_SECRET`/`LINE_CHANNEL_SECRET`。
`api_base_url`	默认 `https://api.line.me`。
`bind_host` / `bind_port`	默认 `127.0.0.1` 和 `8091`。
`webhook_path`	默认 `/callback`。

Discord 字段

字段	约定与默认值
`application_id`	必填的字符串，或用 `AIT_DISCORD_APPLICATION_ID`/`DISCORD_APPLICATION_ID`。
`public_key`	可选凭据，用于 interaction 校验。
`bot_token`	可选凭据，用于 gateway 或 API 调用。
`turn_timeout_seconds`	可选的正数超时；设置了请求超时时，默认至少 300 秒。
`api_base_url`	默认 `https://discord.com/api/v10`。
`http_user_agent`	默认 `curl/8.7.1`。

字段	约定与默认值
`bind_host` / `bind_port`	默认 `127.0.0.1` 和 `8092`。
`interaction_path`	默认 `/interactions`。

选用哪条 worker 命令，决定了该服务模式下 `public_key`、`bot_token` 是必须 其一还是两者都要。

Slack 字段

字段	约定与默认值
`app_token`	可选凭据，用于 socket 模式。
`signing_secret`	可选凭据，用于 HTTP 命令校验。
`api_base_url`	默认 `https://slack.com/api`。
`http_user_agent`	默认 `ait-agent-worker/0.1`。
`bind_host` / `bind_port`	默认 `127.0.0.1` 和 `8093`。
`command_path`	默认 `/command`。

字段	约定与默认值
`ack_text`	默认 `ait is thinking...`。
`response_type`	默认 `in_channel`。

选用哪条 Slack 命令，决定了必须提供的是 socket 凭据还是 HTTP 签名密钥。

验证

用完整 CLI 参考里的 agent worker 查看和启动命令。诊断输出只报告凭据是否 已设置，不会打印它们的值。webhook 监听器先在环回地址上测试，再通过可信的 反向代理对外暴露。

附录

附录：发布 manifest

展开通用命令适配器与专用 AIT 原生家族发布 manifest 的 schema、边界与归属规则。

适用人群 发布工程师与包维护者

https://ait-native.dev/technical/v1.1.1/appendix/release-manifests/

发布清单

1.1.1 有两份发布清单约定，适用范围不同：

文件	档位与用途
`ait-release.json`	`generic-command`：可复用的发布适配器，面向一个包及其一个或多个组件。
`ait-release-family.json`	`family`：专用协调器，面向官方的 AIT 原生 release family。

这两个文件都从记录下来的发布源 Snapshot 里读取。未知字段会被拒绝，路径
按仓库相对路径规范化，各项声明还会拿那个 Snapshot 里实际存在的文件来 核对。

通用适配器：`ait-release.json`

```
{
  "schema": "ait.release.adapter/v1",
  "package": {
    "name": "sample-tool",
    "version": "1.2.0",
    "description": "Portable sample package.",
    "license_files": [
      {"path": "LICENSE", "role": "license"},
      {"path": "NOTICE", "role": "notice"}
    ]
  },
  "components": [
    {
      "id": "cli",
      "ecosystem": "native",
      "working_directory": ".",
      "dependency_files": ["Cargo.toml", "Cargo.lock"],
      "commands": {
        "prepare": [],
        "test": [["cargo", "test", "--locked"]],
        "build": [["cargo", "build", "--release", "--locked"]],
        "smoke": [["target/release/sample-tool", "--version"]]
      },
      "artifacts": [
        {"path": "target/release/sample-tool", "kind": "native-executable"}
      ]
    }
  ]
}
```

根字段与 package 字段

字段	约定
`schema`	确切字符串 `ait.release.adapter/v1`。
`package`	必填对象。
`package.name`	必填，非空的有界单行字符串。
`package.version`	必填，非空的有界单行字符串。
`package.description`	可选的字符串或 `null`。
`package.license_files`	可选数组，1 到 8 条。

字段	约定
`license_files[].path`	指向发布源中某个文件的规范化相对路径。路径必须唯一。
`license_files[].role`	确切的 `license` 或 `notice`；每种 role 最多出现一次。
`components`	必填数组，1 到 64 个组件对象。

组件字段

字段	约定
`id`	必填标识符，在各组件之间唯一。
`ecosystem`	必填标识符，描述该组件的工具链。
`working_directory`	必填的规范化相对目录，根目录写 `.`。它必须存在于源 Snapshot 中。
`dependency_files`	1 到 64 个相对于 `working_directory` 的唯一规范化路径；每个文件都必须存在。
`commands`	必填对象，只能包含 `prepare`、`test`、`build` 和 `smoke`。
`artifacts`	1 到 128 条产物记录。

命令阶段

每条命令都是 argv 数组，不是 shell 字符串。`test` 和 `build` 是必填的， 各自必须含 1 到 16 条命令。`prepare` 和 `smoke` 可选，可以含 0 到 16 条 命令。每条命令含 1 到 64 个字符串参数；可执行文件那个参数不能为空。

命令直接执行，不隐式套 shell。1.1.1 只替换与下列 token 之一完全相等的 整个 argv 值：

```
$AIT_RELEASE_ID
$AIT_RELEASE_VERSION
$AIT_RELEASE_COMPONENT
$AIT_RELEASE_ECOSYSTEM
$AIT_RELEASE_TARGET
$SOURCE_DATE_EPOCH
```

这些 token 不做子串插值，也不构成任意环境变量展开的许可。argv 值里出现 换行、回车和 NUL 字符会被拒绝。

产物字段

字段	约定
`path`	必填的规范化相对输出路径，在组件内唯一。
`kind`	必填标识符，比如 `native-executable`、`python-wheel`，或适配器自定义的其他 kind。
`target`	可选标识符。不写表示这是可移植产物；写了则把该产物关联到某一个发布目标。

选定的构建必须把声明的每个产物都产出为确切的普通文件。选定某个目标时，
会包含匹配该目标的产物；选择可移植集合时，只包含没有 target 的产物。

通用上限

清单大小上限为 1 MiB。标识符以 ASCII 字母或数字开头，之后可用字母、数字、 `.`、`_` 或 `-`。文本字段是有界的单行字符串。相对路径用 `/`，不能是绝对 路径，也不能包含空段、`.`、`..` 段、盘符前缀、冒号或反斜杠。

AIT 家族协调器：`ait-release-family.json`

这不是一份通用的多项目清单。约定 `ait.release.family/v3` 负责协调官方 AIT 原生组件、目标、公开源码投影和分发身份。

```
{
  "schema": "ait.release.family/v3",
  "family": {
    "name": "ait-native",
    "version": "1.1.1",
    "channel": "stable",
    "tag": "v1.1.1"
  },
  "targets": ["aarch64-apple-darwin", "x86_64-unknown-linux-gnu"],
  "public_source": {},
  "components": [],
  "distributions": [],
  "compatibility": {}
}
```

上面那些空对象和空数组只是展示根层形状，它们不是一份有效的发布。真正的 家族清单必须满足下面所有的关系约束。

家族身份与目标矩阵

字段	约定
`schema`	确切字符串 `ait.release.family/v3`。
`family.name`	必填标识符。
`family.version`	稳定版 `MAJOR.MINOR.PATCH` 或 RC 版 `MAJOR.MINOR.PATCH-rc.N`，要和 `channel` 对得上。
`family.channel`	确切的 `rc` 或 `stable`。
`family.tag`	确切的 `v` 加上 `family.version`。
`targets`	1 到 32 个唯一的目标标识符。

组件

`components` 含 1 到 64 个唯一的组件对象。

字段	约定
`id`	必填的唯一组件标识符。
`source_repository`	必填的源 Repository 标识符。在 v3 的公开源码模式下，它必须是 `ait-core`、`ait-server`、`ait-runner`、`ait-python` 或 `ait-node` 之一。
`source_snapshot`	必填，有效且确切的 Snapshot ID。
`ecosystem`	必填标识符。
`license`	必填的有界 SPDX 风格表达式。
`version_scheme`	`family` 或 `pep440`。

字段	约定
`version`	取 `family` 时必须等于家族版本，取 `pep440` 时必须是它对应的规范 PEP 440 映射值。
`artifacts`	1 到 32 条唯一的产物要求。
`artifacts[].kind`	必填的产物 kind 标识符。
`artifacts[].targets`	取自根目标矩阵的唯一目标标识符。空数组表示该 kind 只有一个可移植产物。

产物 kind 和 target 组成的二元组，在同一个组件内不得重复。

分发

`distributions` 含 1 到 128 条记录。每个组件都必须被至少一条分发覆盖到。

字段	约定
channel	github、pypi、npm、oci、homebrew、apt 或 winget。
role	product、standalone 或 implementation。
identity	必填的有界分发身份；channel 与 identity 这一对必须唯一。
components	1 到 64 个已声明的组件 ID。
targets	取自家族矩阵的 1 到 32 个目标。选中的每个组件都必须提供该目标，或者提供可移植产物。

public_source

v3 的公开源码对象故意定得很死，因为它产出的是一个可评审的发布 monorepo。

字段	约定
model	确切的 release-monorepo。
identity	确切的官方公开源码身份 weita2026/ait-native。
product_document	确切的 docs/distribution.md。
family_manifest	确切的 ait-release-family.json。
mapping_manifest	确切的 ait-monorepo-source.json。
build_entrypoints.unix	确切的 build-release.sh。

字段	约定
build_entrypoints.windows	确切的 build-release.ps1。
build_entrypoints.implementation	确切的 build-release.mjs。
subtrees	各组件用到的每个源 Repository 各占一条。
subtrees[].source_repository	已声明的源 Repository。
subtrees[].path	同名的确切公开目录。
subtrees[].transforms	按顺序作用于该 subtree 的、在允许清单内的 transform ID。

字段	约定
transforms	所选源 Repository 集合所需的确切 transform 定义。
transforms[].id	有界的 ASCII ID，以 /v1 结尾。
transforms[].source_repository	拥有它的源 Repository。
transforms[].path、from、to	该 transform ID 对应的、在允许清单内的确切路径重写三元组。

当前的允许清单里，只有 runner 和 Python 源码 subtree 所需的同级 core 路径投影。任意改写源码不是这份清单的扩展机制。那唯一一条 GitHub product 分发必须使用同一个公开身份，并覆盖家族里的每个组件和每个目标。

兼容性映射

compatibility 是必填对象，最多 64 条。每个 key 是一个标识符，每个 value 是非空的有界单行字符串。它记录的是面向使用者的兼容性事实；它不替代组件 版本、Snapshot 锁定或目标矩阵。

评审边界

- 清单的改动，要和它引用的源码及依赖文件改动记进同一个 Snapshot。
- 更新组件的 source_snapshot 之前，一定要先评审它接纳的产物、许可证、 版本和分发覆盖情况。
- 生成出来的候选、检查、构建、冻结、晋级、校验和、映射与回执文件，别放进这份编写参考里。它们是发布输出，不是额外的用户配置文件。

- 用完整的 `ait release` 命令参考去创建、检查、构建和查看发布证据；不要 手写输出回执。

相关参考

- 配置文件一览
- 完整 CLI 参考
- 分发与发布

附录

附录：环境变量

已安装 1.1.1 环境的规范化 31 条约定，明确写出归属、设置方式、机密性与进程边界行为。

适用人群 开发者、运维、集成方与发布工程师

https://ait-native.dev/technical/v1.1.1/appendix/environment-variables/

这份附录覆盖什么

这是当前 `ait-native 1.1.1` 家族（`ait`、Agent worker、`ait-server` 和 `ait-runner`）已安装进程环境的完整约定。它包含 31 个规范化名称或有界名称族。这些条目来自三份机器可读的运行时注册表，而不是在源码里搜 `AIT_` 字样搜出来的。

整份约定就放在这一篇里。每一行都写明：哪个组件拥有这个值、由谁设置、 是不是机密，以及它会改变什么。

- 用户和运维类的值可以放进被拉起的进程环境里。命令若暴露了同样 的选择，明确的命令行选项仍然优先。
- AIT 注入类的值会跨越受管的子进程边界。用户不应该靠设置它们去把 另一个 Task 或 CI 尝试引到别处。
- 自动化类的值只属于指名的、受支持的构建或发布边界。
- 机密应该放进服务或 CI 的密钥库。不要提交、不要打印，也不要留在会被 保存的命令行历史里。

`ait-server` 和 `ait-runner` 都不会自动加载项目的 `.env` 文件。必须由 shell、服务管理器、容器运行时或 Agent 监管进程把配置好的值放进进程环境。

注册表口径

注册表	公布的口径
`ait.environment-contract/v1`	23 条 `ait-core`、CLI、Agent 与发布边界条目。
`ait.server.environment-contract/v1`	7 条 server 条目。`AIT_NATIVE_SERVER_DATA` 和 `AIT_PERFETTO_TRACE` 已经归 core 所有，所以 server 净增 5 个新名称。
`ait.runner.environment-contract/v1`	4 条 runner 条目。它的外部仓库名称族里包含了 core 那个具体的 `AIT_EXTERNAL_CORE_REPO_ROOT` 输入，所以 runner 净增 3 个规范化名称。
公布总数	31 个唯一名称或有界名称族。 runner 那个有界的外部 Repository 名称族里含有 core 的具体诊断成员，所以只算一次。

运行时、仓库与身份

名称	归属、设置方、是否机密及行为
`AIT_NATIVE_ACTOR`	ait-core · 用户或运维 · 非机密。 记入本地溯源和远程请求的作者身份。
`AIT_NATIVE_SERVER_DATA`	ait-core 与 ait-server · 运维 · 非机密。 没有传 `ait-server --data` 时使用的持久 server 权威根目录，server 感知型的 core 操作也会用到。放在持久存储上。
`AIT_REPO_ROOT`	ait-cli · 用户或监管进程 · 非机密。 用于进程发现的显式仓库根。优先正常进入仓库目录；只有调用方无法确定工作目录时才用它。
`AIT_RUNTIME_DATA`	ait-core · 运维 · 非机密。 本地诊断、Snapshot 元数据、status 缓存和 server 感知型操作共用的运行时数据根目录。
`AIT_SERVER_TOKEN`	ait-runner · 运维 · 机密。 runner 发往配置好的 `ait-server` 端点的可选 bearer 凭据。server URL 和 worker 身份仍然是 runner 的显式命令行选项。

Agent worker 与集成凭据

Agent worker 先选定一份带类型的 worker 清单。下面这些环境值提供有界的 worker 引导参数和凭据--这些东西不适合直接放进普通的仓库内容里。

名称	归属、设置方、是否机密及行为
AIT_AGENT_CONFIG_PATH	ait-agent-worker · 用户或监管进程 · 非机密。 带类型的 Agent worker 清单路径。普通仓库的默认值是 .ait/agent-workers.json。
AIT_DISCORD_APPLICATION_ID	ait-agent-worker · 运维 · 非机密。 所选 worker 的 Discord 应用身份。
AIT_DISCORD_BOT_TOKEN	ait-agent-worker · 运维 · 机密。 所选 worker 的 Discord bot 凭据。
AIT_DISCORD_PUBLIC_KEY	ait-agent-worker · 运维 · 非机密。 用于校验 interaction 请求的 Discord 公钥。
AIT_LINE_CHANNEL_ACCESS_TOKEN	ait-agent-worker · 运维 · 机密。 所选 worker 的 LINE channel access 凭据。
AIT_LINE_CHANNEL_SECRET	ait-agent-worker · 运维 · 机密。 所选 worker 的 LINE 请求校验密钥。

名称	归属、设置方、是否机密及行为
AIT_OPENAI_API_KEY	ait-agent-worker · 运维 · 机密。 当所选 worker 走 OpenAI 边界时，提供给它的 OpenAI 凭据。
AIT_SLACK_APP_TOKEN	ait-agent-worker · 运维 · 机密。 所选 worker 的 Slack app 级凭据。
AIT_SLACK_SIGNING_SECRET	ait-agent-worker · 运维 · 机密。 Slack 请求签名校验密钥。
AIT_TELEGRAM_BOT_TOKEN	ait-agent-worker · 运维 · 机密。 所选 worker 的 Telegram bot 凭据。
AIT_TELEGRAM_OPENAI_API_KEY	ait-agent-worker · 运维 · 机密。 当该 worker 用单独的 key 时，Telegram 专用的 OpenAI 凭据。
AIT_TELEGRAM_WEBHOOK_SECRET	ait-agent-worker · 运维 · 机密。 Telegram webhook 校验密钥。

server CI 内存接纳

这三个变量只属于 server 持有的 CI 接纳环节。Task worktree 的 RAM 放置 属于仓库配置，在 `task_worktree.memory_root` 下面，不是进程环境配置。 它带类型的字段、macOS 上 8 GiB 的默认值、只读诊断和回落到持久磁盘的 规则，都写在 并行 Task 隔离。

名称	归属、设置方、是否机密及行为
AIT_NATIVE_SERVER_CI_RAM_MIN_AVAILABLE_BYTES	ait-server · 运维 · 非机密。 可选的确切空闲字节下限，在把 server CI 接纳到已验证的 RAM 根目录时生效。不设置就只做挂载校验，不额外加下限。
AIT_NATIVE_SERVER_CI_RAM_RECLAIM_TARGET_BYTES	ait-server · 运维 · 非机密。 可选的回收后空闲字节目标值。生效目标永远不会低于接纳下限。
AIT_NATIVE_SERVER_CI_RAM_ROOT	ait-server · 运维 · 非机密。 用于隔离 server CI 作业的绝对路径、经过校验的内存宿主根目录。它必须位于持久 server 权威之外。

存储与远程操作调优

这些是高级的有界控制项。取值非法时会走写明的回落值，或者在拥有它的边界上校验失败；它们不会改写已经存下来的对象。

名称	归属、设置方、是否机密及行为
AIT_OBJECT_PACK_CHUNK_MIB	ait-server · 运维 · 非机密。 object pack 写入分块大小，单位 MiB，取正数。不设置、非法或为零时用 8 MiB。
AIT_REMOTE_MUTATION_RESPONSE_DEADLINE_SECONDS	ait-core · 运维 · 非机密。 远程变更的响应截止时间。普通变更默认 10 秒，Task Land 默认 30 秒；取值小于等于零时，仅关闭这一项响应截止时间。
AIT_REMOTE_MUTATION_SETTLE_POLL_SECONDS	ait-core · 运维 · 非机密。 远程变更之后的对账轮询间隔。默认 0.25 秒。
AIT_REMOTE_MUTATION_SETTLE_WINDOW_SECONDS	ait-core · 运维 · 非机密。 变更之后的对账窗口。默认 5 秒。
AIT_TREE_PACK_CHUNK_MIB	ait-server · 运维 · 非机密。 Tree pack 写入分块大小，单位 MiB，取正数。不设置、非法或为零时用 8 MiB。

受管进程边界

下面这些值在子进程里能看到，但不是普通配置项。AIT 会用当前操作确切的 路径或归属令牌把它们覆盖掉。

名称	归属、设置方、是否机密及行为
`AIT_EXTERNAL_<NAME>_REPO_ROOT`	ait-runner 与 ait-core · AIT 注入名称族 · 非机密。 runner 创建的、指向某个已锁定外部仓库的路径。`<NAME>` 是非空的规范化大写 ASCII，可含数字或下划线；core 的诊断用的是具体的 `AIT_EXTERNAL_CORE_REPO_ROOT` 成员。
`AIT_RUNNER_ATTEMPT_ROOT`	ait-runner · AIT 注入 · 非机密。 当前这次执行尝试的确切 runner 私有根目录。runner 的父级位置用 `--attempt-root` 选定；环境里现成的值无法把子尝试引到别处。
`AIT_RUNNER_WORKSPACE`	ait-runner · AIT 注入 · 非机密。 当前这次执行物化出来的确切工作区。
`AIT_WORKSPACE_LOCK_OWNER_TOKEN`	ait-cli · AIT 注入 · 机密。 只传给共用当前工作区锁、且已通过校验的嵌套命令的不透明令牌。不要自己造，也不要打日志。

诊断与发布边界自动化

名称	归属、设置方、是否机密及行为
`AIT_PERFETTO_TRACE`	ait-core 与 ait-server · 用户或运维 · 非机密。 在开启 tracing 的进程中，选择性采集 Perfetto trace 的非空输出路径。
`AIT_SHARED_CARGO_TARGET_DIR`	ait-release · 自动化 · 非机密。 受支持的原生发布冒烟边界所用的显式共享 Cargo target 目录。它不是通用的运行时存储选择器。

哪些是有意排除的

发布清单里的占位符是由发布适配器解析的 argv 模板 token，不是环境输入。 工作流里局部的 CI env：
别名、测试夹具、故障注入开关、生成的 ID、报告
标签，以及只归仓库构建或部署脚本所有的值，同样不在已安装运行时约定的
范围内。这些东西应该在拥有它们的工作流或脚本旁边说明，而不是摆出来当 成用户必须理解的设置项。

这条范围规则让这份附录保持完整，又不至于把源码里每一个 token 都变成 对外的配置承诺。

相关

- 附录：`.ait/config.json`
- 并行 Task 隔离
- ait-server：远程权威与恢复

版本权威

1.1.1 源码权威

这份 PDF 是便于翻阅的公开参考，不是另一份产品契约。涉及发布权威时，请以下方确切的 1.1.1 源码、Release、分发契约和 CLI 解析器为准。

公开源码	https://github.com/weita2026/ait-native/tree/v1.1.1
公开提交	`d248b29d3d988867366287d91e57b6c5bace716b`
发布版本	https://github.com/weita2026/ait-native/releases/tag/v1.1.1
分发 SHA-256	`f2290e405520454e7926f09cdda6a1fe4be1b60300f03d4222918bedc9dd3efa`
CLI 解析器 SHA-256	`829ccf69699c28bcca872cd2b191d256ed8aef4b4e8ada0590fb1e0aacb38215`
AIT-CORE	`SNP-ED7593DBF982`
AIT-SERVER	`SNP-0CCD7DD2A077`
AIT-RUNNER	`SNP-35C9C133D2EE`
AIT-PYTHON	`SNP-756C731A4CC0`
AIT-NODE	`SNP-55E90D0A81F1`

如需当前在线版、搜索和特定路由更新，请打开ait-native.dev/technical/。