# 蔡宏天-256ea0146-项目申请书

- 项目名称：AI 模型和数据集的生命周期管理系统
- 项目主导师：杨晓东
- 申请人：蔡宏天
- 日期：2025.05.28
- 邮箱：cht7janus@gmail.com

## 项目背景

### 项目基本需求

仓库地址：https://github.com/Alluxio/alluxio-py， 分支是 open_source_summer。

### 目的 1:解决目前无法对模型与数据集等数据或元数据进行统一生命周期管理的问题

为了解决模型与数据集及其元数据缺乏统一生命周期管理的问题，可以首先搭建一个集中化的管理平台，将模型注册中心（Model Registry）与数据集目录（Data Catalog）打通，实现从训练、评估到部署、监控和退役的全流程可视化管理。在这个平台中，每个模型和数据集都有唯一的 ID、版本号和完整的元数据（如训练环境、依赖关系、评估指标、数据来源等），并通过角色权限和审计日志保证操作可追溯；同时，流程节点（实验、验证、生产、归档等）都设有钩子，以便在关键环节进行自动化评估、合规扫描和告警通知。

此外，应引入灵活的版本控制机制和高效的 IO 策略：针对版本控制，采用“语义化版本+哈希值”命名、分支/标签管理和增量存储，结合 CI/CD 流水线实现模型的自动化晋升与回滚；针对 I/O 性能，利用分布式/边缘缓存、分层存储（热/温/冷层）、流式加载与零拷贝技术，以及统一的 IO 接口抽象层，实现模型与数据集的并行预热、本地缓存和按需拉取，从而在保证低延迟、高吞吐的同时节约存储成本。这样，团队便可在统一的平台上快速迭代、透明协作，极大提升模型上线效率与可维护性。

## 目标 2：解决框架兼容缺失问题

目前 alluxio-py 仅支持文件级别的基本 I/O 接口，缺乏面向模型参数，数据集等高阶资源的语义化封装。研发人员在训练脚本中往往需要先手动调用 alluxio-py 把模型或数据下载到本地，再分别使用 torch.load()、tf.data.TFRecordDataset() 等框架接口读取，导致：代码冗长、不统一，迁移到 Alluxio 后需大幅改动现有训练/推理脚本；无法复用 PyTorch DataLoader、TensorFlow tf.data 流水线的多线程 / 预取优化；路径管理混乱，难以与元数据中的版本、标签做关联。

在 feature/LancePyTorchTFAdapter 分支内，新建 alluxio.torch 和 alluxio.tf 子模块：**API 对齐**：提供 load_model(uri)、Dataset(uri, **kwargs) 等函数 / 类，使用户仅需把原有路径替换为 alluxio:// 前缀或调用一次包装器，即可无缝接入现有训练流程；**底层实现**：内部自动完成 Alluxio → 本地缓存 → 共享 mmap 的三级缓存策略，并实现 __len__、__getitem__、流式解码等接口，与 PyTorch torch.utils.data.Dataset 和 TensorFlow tf.data.Dataset 保持完全形态兼容；**性能目标**：加载延迟较原生 SSD 方案差距 < 5 %，并支持 DataLoader/Prefetch 在多线程或异步场景下的性能平滑扩展。

## 项目相关仓库

项目代码仓库：

https://github.com/Alluxio/alluxio-py

项目文档仓库：

https://github.com/Alluxio/alluxiopy/blob/main/README.md

Alluxio 是一个开源的分布式内存速度虚拟存储系统，它提供统一的接口来管理和访问不同存储系统的数据。Alluxio 主要用于大数据和机器学习场景中，通过在内存中缓存数据来加速数据访问，从而显著提高计算性能和效率。

我们的系统则侧重于数据与模型的全生命周期管理，在此基础上持久化元数据，并为多种深度学习框架（例如 PyTorch、TensorFlow 等）提供友好的调用接口。以下是系统在数据与模型管理方面核心设计的几个关键点：

1. 版本化与追踪
   我们为每一次数据集变更和模型训练结果都自动打上语义化版本（Semantic Versioning），并在元数据库中记录完整的变更日志、输入参数、训练配置和精度指标。这样可以随时回溯任意版本的数据或模型，保障实验的可复现性和可审计性。

2. 元数据持久化
   所有与数据和模型相关的元信息（如 schema、统计摘要、特征分布、超参数设置、训练日志等）都存储在高可用的元数据库中。通过 RESTful API 或者 SDK，用户可以实时查询任意数据集或模型的完整元数据，便于在不同项目或团队之间共享与复用。

3. 统一接口与多框架支持
   我们提供与 PyTorch、TensorFlow 等深度学习框架的原生对接插件。开发者只需调用统一的 load_dataset()、save_model()、version_model() 等高层 API，就能在后台自动完成数据加载、预处理流水线、模型存取与切换，彻底解耦业务代码与底层存储与元数据管理。

4. 高效存取与扩展性
   底层采用列式存储与分布式缓存相结合的方式，既支持批量大规模训练场景下的高吞吐读写，也能在在线推理时实现低延迟访问。结合自动分区和预取策略，系统能够根据使用热点动态调整缓存和存储布局，最大程度降低 I/O 开销。

此外，我们在数据血缘与治理以及监控与告警框架方面进行了如下扩展：

1. 数据血缘与治理
   为了实现端到端的数据可追溯性，系统在元数据库中记录每个数据集和模型从创建、预处理到训练、部署的完整流程信息，包括数据来源、处理脚本版本、特征工程方法、训练代码 Git 提交 ID 等。结合 Alluxio 的访问日志和缓存状态，在元数据库中为每一次读写操作打上唯一的引用标识，便于可视化地追踪某个模型预测结果所依赖的原始数据路径。通过定期导出和分析这些血缘信息，团队可以快速定位数据异常来源、评估某次变更对下游任务的影响。
2. 监控与告警框架
   利用基于 Grafana 的可视化面板，在各个层面（Alluxio Master/Worker、底层对象存储节点、元数据库实例）分别采集关键指标，例如吞吐量（IOPS）、延迟（Latency）、缓存命中率（Hit Rate）、磁盘与网络利用率、元数据库慢查询次数等，并在这些指标超过预设阈值时通过钉钉/邮件/短信等渠道推送告警。

# 技术方法及可行性分析

## 问题一：数据生命周期管理

## 问题分析：

当前系统仅把模型与数据集当静态文件存放，缺乏“注册→激活→废弃→

归档→删除”的全流程管理。旧版本长久滞留在高性能存储，推高成本；线上万一加载到错误版本，只能人工改路径回滚，既不安全又难审计；多 Worker 集群里又因各自缓存状态不一致，训练与推理服务常出现“找不到资源”或“加载旧模型”的混乱。

## 技术方案 1：基于 Lance 的统一生命周期管理

首先，我们在系统中引入一个统一的资源管理器（ResourceManager），用于对外提供资源的注册、获取、更新和删除等功能。ResourceManager 会根据资源类型（如训练集、预训练模型、推理模型等）和资源名称，动态实例化一个 LanceDataset 对象。LanceDataset 负责接收资源的元数据和原始数据路径，通过 Lance 的 Python SDK 将数据写入到 Alluxio 命名空间下的指定路径，例如 alluxio://lance_root/my_dataset。此时 Lance 会自动将数据存储为列式格式并生成第一个快照，系统会在元数据中记录该快照的创建者、时间戳以及初始依赖信息（例如训练代码版本号或特征流水线标识），并将其标记为“活跃”状态。之后，用户只需调用 ResourceManager.getResource(type, name) 即可获取到一个已经初始化好的 LanceDataset 实例，背后由 Lance 和 Alluxio 共同完成对数据的本地缓存与远程存储协调。

当需要将新数据版本上线时，上层业务首先将增量数据或全量更新内容写到临时目录中，然后调用 ResourceManager.updateResource(type, name, newDataPath) 通知系统进行升级。对应的 LanceDataset 会调用 lance.upsert 或 lance.write_table 方法，将新数据追加或覆盖到原有表中。由于 Lance 采用列式存储并内置快照机制，这一操作会自动在同一张表中生成新的快照，并将新快照标记为“活跃”，同时将先前版本的状态修改为“弃用”。依赖该数据的 Worker 或推理服务在下一次缓存刷新时，会感知到“活跃”快照指针的变化，进而通过 lance.load_table(path, version=n) 直接读取新版本，无需关心底层物理路径或文件名的变动。

如果某个历史版本确实不再需要继续提供给上层服务，但暂时还不能完全删除，就可以调用 ResourceManager.deprecateResource(type, name, version) 将其下线。此时 LanceDataset 会在相应表的元数据中把该版本标记为“弃用”，并且触发 Lance 与 Alluxio 之间的事件通知，让 Alluxio 尽快从缓存中清除该版本的数据块，以确保后续读取只会落到最新或指定的“活跃”版本。需要注意的是，被标记为“弃用”的快照仍然保留在后端存储中，只是在缓存层被剔除。这样做既能保证存储成本不会过度膨胀，又能在必要时快速回滚到那些版本。

对于长时间没有访问历史版本的场景，比如某个版本在三十天内都未被 Worker 或人工引用，就可以执行 ResourceManager.archiveResource(type,

name, version) 对该版本进行归档。实际操作是由 LanceDataset 调用 lance.vacuum(path, retention_seconds=<30 天对应秒数>)，让 Lance 在 Alluxio 热存储层中回收过期的快照，将其移动到挂载在 Alluxio 外部的对象存储等冷存储系统。即便数据被移至冷层，后续若有人需要访问该版本，Lance 会自动先将其从冷存储异步拉回到 Alluxio 缓存层，然后再返回给调用者。归档完成后，元数据会被更新为“归档”状态，表明该版本已进入冷存储但仍可检索。

当某次升级出现严重问题，需要快速回滚时，用户只需调用 ResourceManager.rollbackResource(type, name, toVersion) 指定一个之前的版本即可。LanceDataset 先调用 lance.load_table(path, version=toVersion) 将所需版本从冷存储或缓存准备好，然后通过元数据将该快照的状态修改为“活跃”，并将当前版本标记为“弃用”。随后的缓存刷新后，所有依赖该资源的服务都会自动加载到回滚后的快照，无需人工干预物理文件或重新配置路径。这样一来，整个回滚过程变得极其简便，只依赖于 Lance 和 Alluxio 的快照与缓存机制，而业务方无需了解底层存储的细节。

最后，如果某个资源彻底不再需要保留，可以调用 ResourceManager.deleteResource(type, name) 让系统永久清理其所有快照和元数据。此时 LanceDataset 会执行 lance.delete_table(path) 将该资源表下的所有版本从 Alluxio 热存储和后端冷存储中一并删除，并在资源注册中心移除对应记录。至此，该资源及其历史版本将不再占用任何存储或缓存空间。通过上述几段完整的流程描述，我们实现了从注册到更新、废弃、归档、回滚、删除的全生命周期管理，依赖 Lance 的列式快照能力和 Alluxio 的缓存协同，使资源管理对上层业务完全透明且高效。

## 问题二：Etcd 存储大规模模型参数与数据集的局限性

### 问题分析：

Etcd 天生面向轻量级配置和小规模 KV 存储，整个集群的数据量一旦接近甚至超过 8 GB，就会因 Raft 日志和快照频繁膨胀导致领导选举增多、写入延迟陡增，甚至可能出现节点不可用的情况；同时单次写入或读取数百 MB 以上的大对象会迅速吞噬内存和网络带宽，彻底拖垮集群性能，因此 Etcd 无法承载几 GB 甚至 TB 级别的大模型参数和数据集。

### 优化方案 1：

将模型权重和训练/推理数据集统一存放在高性能分布式对象存储（如 S3、MinIO、Ceph/RADOSGW、OSS 等），在轻量级元数据服务（如 MySQL、PostgreSQL 等）中仅保存文件路径、版本号与校验信息。业务进程加载模型时，先从元数据层获取最新索引，再并行或按需从对象存储拉取实际文件；在拉取过程中，通过 Alluxio 作为二级缓存层，将热点模型分片或整个数据集缓存在本地或内存，利用 Alluxio 的内存加速和零拷贝能力显著提升 I/O 性能，避免对后端对象存储的重复压力。该方案既能发挥对象存储的海量容量与高并发特性，又可保留 Etcd 级别的轻量级协调能力，同时借助 Alluxio 缓存大幅提高系统的可扩展性与稳定性。

## 问题三：提供与 PyTorch / TensorFlow 等深度学习框架相关 API

## 问题分析：

当前 alluxio-py 仅提供字节流或本地路径级别的 I/O 接口，深度学习脚本想要加载 Alluxio 中的模型和数据集，必须先手动把文件落地到容器，再调用 torch.load()、tf.data.TFRecordDataset() 等原生 API 进行二次读取。这样做不仅增加了额外的复制与清理步骤，还破坏了 PyTorch DataLoader 与 TensorFlow tf.data 在多线程、异步预取上的性能优势。更为突出的是，线上推理与离线训练往往共用一套资源索引，如果缺少对框架友好的加载方式，运维与研发不得不在路径、缓存策略和脚本改造之间反复权衡，最终导致版本迁移慢、重复代码多、错误率高。

## 技术方案 1：Alluxio FUSE 挂载方案

我们在深度学习框架内部引入一个 Alluxio-FUSE 适配器子模块：启动时由 FuseMountManager 自动检测并挂载 Alluxio 文件系统到一个本地目录（如 /mnt/alluxio），然后所有对文件的 open/read/write/close 调用都由 PosixFsAdapter 统一拦截——它只需在框架底层 I/O 接口注册点将用户传入的 Alluxio 路径（例如 alluxio:///models/resnet50.params）拼接为挂载目录下的实际路径并调用对应的 POSIX 系统调用。最后，通过在 TensorFlow 中实现自定义 FileSystem 子类、在 PyTorch 中封装 AlluxioDataset、或在

MXNet 中替换其 I/O 接口，确保所有的数据加载、模型参数读写、Checkpoint 存储都透明地走 FUSE 挂载路径，同时依赖 Alluxio 本地缓存机制和标准的文件描述符并发管理，无需修改框架核心即可实现高效稳定的分布式存储访问。

### 技术方案 2：实现一套 Python 的 SDK

深度学习框架通过 alluxio-py 或 Alluxio 提供的 Python SDK 在应用层直接调用 Alluxio 服务，而无需借助操作系统级别的 FUSE 挂载。具体而言，应在 PyTorch 环境中开发一个名为 AlluxioDataset 的自定义 Dataset 类，该类在其 **getitem** 方法中通过 alluxio_client.files().open(path) 接口获取字节流（BytesIO），并将此字节流传递给 torchvision 或自定义预处理流水线进行解码与转换。同理，在 TensorFlow 环境中需继承 tf.io.gfile.FileSystem 或使用 tf.data.Dataset.from_generator 方式，将对 TFRecord 或图像文件的读取请求在底层重定向至 alluxio_client.open 或 alluxio_client.read 接口，从而获取相应数据。通过上述机制，框架内部的 I/O 操作可直接由 Alluxio Python SDK 驱动，取代传统的本地磁盘访问路径。数据在初始化时以内存流形式加载后，框架能够并行启动多个线程或进程并发读取不同文件分片，也可对多个小文件进行合并流式读取，从而降低磁盘碎片化带来的性能损耗。该方案在 Python 代码层面完成适配，无需在操作系统级别配置挂载目录，也可规避 FUSE 用户态与内核态切换时产生的额外开销。然而，此方案需要针对不同版本的深度学习框架持续维护适配接口，同时需实现将 Alluxio 返回的字节流映射为框架可识别的文件对象，以保证 PyTorch 或 TensorFlow 在内部 I/O 流程中能够与 Alluxio 上的数据资源无缝对接。

## 项目进度规划

### 项目时间表

阶段 1：需求分析与设计（6 月 30 日 - 7 月 14 日）

• 6 月 30 日 - 7 月 2 日：项目启动会议，明确项目目标和需求。

• 7 月 3 日 - 7 月 7 日：熟悉 Alluxio 开源版项目的代码架构，功能用途已经编译运行方式等。

• 7 月 8 日 - 7 月 14 日：调研现有相关的解决方案和成熟的产品设计，完成系统的功能设计。

阶段 2：在本地文件系统上实现 AI 模型和数据集的生命周期管理完整的功能

（7 月 15 日 - 8 月 10 日）

• 7 月 15 日 - 7 月 31 日：实现基本的生命周期管理（注册、激活、更新、废弃、删除等功能）。

• 8 月 01 日 - 8 月 10 日：在本地环境中进行测试，并完成对额外需求的扩展与优化。

阶段 3：以 Alluxio 作为存储系统，实现 AI 模型和数据集的生命周期管理（8 月 11 日 - 8 月 31 日）

• 8 月 11 日 - 8 月 20 日：基于 Alluxio 完成基本生命周期管理功能的移植与实现。

• 8 月 21 日 - 8 月 31 日：进行 Alluxio 层面的测试，并对接口和性能进行扩展优化。

阶段 4：实现与 torch 和 tensorflow 可无缝衔接的 API 接口（9 月 01 日 - 9 月 15 日）

• 9 月 01 日 - 9 月 10 日：完成 Alluxio-FUSE 挂载方案的开发与联调。

• 9 月 11 日 - 9 月 15 日：完成基于 Python SDK 的 AlluxioDataset、TFRecordDataset 等框架适配接口，并进行功能验证。

阶段 5：测试与调优（9 月 16 日 - 9 月 25 日）

• 9 月 16 日 - 9 月 20 日：编写单元测试和性能测试。

• 9 月 21 日 - 9 月 25 日：根据测试结果进行调优，确保功能稳定，性能达到预期。

阶段 6：文档编写与项目总结（9 月 26 日 - 9 月 30 日）

• 9 月 26 日 - 9 月 28 日：编写项目文档，包括设计文档、用户手册和开发者指南。

• 9 月 29 日 - 9 月 30 日：项目总结会议，回顾项目成果，讨论后续改进计划。

项目里程碑

1. 需求分析与设计完成（7 月 14 日）

2. 数据生命周期管理在本地文件系统阶段完成（8 月 11 日）

3. 基于 Alluxio 的数据和元数据持久化优化与开发完成（9 月 1 日）

4. 深度学习框架 API 接口集成完成（9 月 15 日）

5. 全面测试与调优完成（9 月 25 日）

6. 项目文档与总结完成（9 月 30 日）