模块化LLM操作容器 (MLOC) 系统设计方案

1. 项目愿景与目标

目标: 创建一个名为 **MLOC (Modular LLM Operations Container)** 的统一容器化框架。该框架旨在简化和标准化 LLM 的训练、微调、推理和应用（RAG, Agent）流程。通过一份配置文件，即可在异构硬件集群（Kubernetes）中一键部署，并执行指定的 LLM 任务。

核心特性:

- 配置驱动: 节点的角色和任务完全由 YAML 配置文件定义。
- 模块化与可扩展: 核心功能（SFT, PPO, RAG 等）作为可插拔的模块，方便集成 TRL, LangChain, vLLM 等业界优秀开源库。
- 异构硬件感知: 能够识别节点硬件（如不同型号的 GPU），并支持任务的智能调度。
- 云原生与广域网适应性: 为 Kubernetes 设计，并针对节点可能分布在不同广域网的场景进行了优化。

2. 核心设计原则

1. 职责分离 **(Separation of Concerns)**:
    - **Orchestrator (主控节点)**: 负责“管理”，解析任务、分配工作、监控状态，但不执行具体的计算密集型任务。
    - **Worker (工作节点)**: 负责“执行”，专注于运行 SFT、推理等实际任务，状态和依赖由主控节点管理。
2. 无状态 **Worker**: 工作节点本身不保存持久化状态。模型、数据集、日志等都应由外部存储（如 S3、Hugging Face Hub、PVC）管理，这使得 Worker 可以被随时替换、增删，从而实现高可用和弹性伸缩。
3. 统一的通信接口: 节点间采用标准化的通信协议（如 gRPC 或 RESTful API + 消息队列）进行交互，降低耦合度。
4. 硬件抽象: 框架应提供硬件抽象层，使得任务逻辑与具体的硬件设备解耦。例如，通过一个 `DeviceManager` 来管理 GPU 分配。

3. 系统架构 (System Architecture)

具体化为 `Orchestrator/Worker` 架构。

组件详解:

1. **Orchestrator (主控节点)**:

    - **API Server**: 暴露 RESTful API，接收用户/外部系统的任务请求（例如，启动一个新的 SFT 任务）以及用量查询请求。
    - **Task Parser**: 解析 `task.yaml` 配置文件，理解任务的类型、依赖、数据源和资源需求。
    - **Scheduler**: 核心调度器。它会查询服务发现系统，找到符合条件的空闲 Worker 节点（例如，需要 A100 GPU 的 PPO 任务会寻找拥有 A100 的节点），然后将任务分发出去。
    - **State & Usage Monitor**: 监控 Worker 的心跳和任务进度，处理失败、超时等异常情况，并负责任务的重试或失败报告。同时，收集和聚合任务执行数据，用于用量统计。
2. **Worker (工作节点)**:
    - **Lifecycle Manager**: 容器启动时，向服务发现系统注册自己，并上报硬件信息（GPU 型号、显存大小等）。
    - **Task Listener**: 监听消息队列中的任务。一旦接收到属于自己的任务，便开始执行。
    - **Module Loader**: 根据任务类型（`sft`, `rag_inference` 等），动态加载对应的任务模块（Wrapper）。
    - **Execution Engine**: 这是实际执行任务的地方。它会调用 `vLLM`, `TRL`, `Hugging Face Transformers` 等库来执行计算。
    - **Resource Downloader**: **(关键)** 负责根据任务指令，从指定的中央代码库/数据源（如 Hugging Face Hub, S3, Git LFS）拉取模型权重和数据集。
    - **Result Uploader**: 任务完成后，将产物（如新的模型权重、评估结果、日志）上传到指定的目标位置。
3. 通信层 **(Communication Layer)**:
    - 推荐方案: **REST API +** 消息队列 **(e.g., RabbitMQ, Redis Pub/Sub)**
        - **REST API (Orchestrator)**: 用于接收来自用户的同步命令，如“部署这个任务”、“查询任务状态”、“获取用量报告”。
        - 消息队列: 用于 Orchestrator 和 Worker 之间的异步任务分发。这种方式解耦了两者，Orchestrator 只管发布任务，Worker 只管消费任务，非常适合高延迟、不稳定的广域网环境。
4. 外部依赖 **(External Dependencies)**:
    - 配置/服务发现: Kubernetes 的 `ConfigMap` 可以存储配置，`etcd` 或 `Consul` 可以用于服务发现和 Worker 注册。
    - 数据与模型存储: 强烈建议使用对象存储（如 AWS S3, MinIO）或专门的模型库（如 Hugging Face Hub, Git LFS）作为唯一的真相来源 (Single Source of Truth)。
    - 日志与监控: 使用 `Prometheus` 进行指标监控，`Grafana` 进行可视化，`ELK/Loki` 堆栈进行日志收集。

4. 建议代码结构 (Proposed Code Structure)

mloc/

```
├── .github/              # CI/CD workflows
├── configs/              # 任务配置示例
│   ├── sft_mistral_7b.yaml
│   ├── rag_inference_service.yaml
│   └── ppo_zephyr_7b.yaml
├── docker/
│   ├── base/Dockerfile     # 基础环境，包含CUDA, Python等
│   └── app/Dockerfile      # 应用层，安装项目代码和依赖
├── docs/                 # 项目文档
├── scripts/              # 部署、测试等辅助脚本
│   └── deploy_k8s.sh
└── src/
    ├── mloc/
    │   ├── __init__.py
    │   ├── main.py           # 容器入口，根据环境变量或参数决定启动为 Orchestrator 还是
Worker
    │   ├── orchestrator/
    │   │   ├── __init__.py
    │   │   ├── api.py        # Flask/FastAPI 应用 (包含 /tasks 和 /stats 路由)
    │   │   ├── scheduler.py    # 调度逻辑
    │   │   └── monitor.py      # 状态与用量监控
    │   ├── worker/
    │   │   ├── __init__.py
    │   │   ├── listener.py     # 监听任务队列
    │   │   └── executor.py     # 任务执行器，动态加载模块
    │   ├── modules/          # 核心任务模块 (Wrappers)
    │   │   ├── base_module.py  # 所有模块的抽象基类
    │   │   ├── sft_module.py   # 封装 TRL 的 SFTTrainer
    │   │   ├── rm_module.py    # 封装 Reward Model 训练
    │   │   ├── ppo_module.py   # 封装 TRL 的 PPOTrainer
    │   │   ├── rag_module.py   # 封装 RAG 流程
    │   │   └── agent_module.py # 封装 Agent 逻辑
    │   ├── integrations/       # 对外部库的进一步封装
    │   │   ├── vllm_engine.py
    │   │   └── hf_downloader.py
    │   └── common/           # 通用工具和定义
    │       ├── constants.py
    │       ├── schemas.py      # Pydantic 数据模型，用于配置和API
    │       └── utils.py
    ├── tests/              # 单元测试和集成测试
    └── setup.py            # 项目打包配置
```

5. 配置驱动 (Configuration Driven Design - `task.yaml`)

示例 (`sft_mistral_7b.yaml`):

```
apiVersion: mloc/v1
kind: TrainingTask
metadata:
  name: sft-mistral-7b-on-dolly
  owner: john-doe # 用于用量统计
  project: llm-research # 用于分组统计
  annotations:
    description: "Supervised fine-tuning for Mistral-7B on the Dolly dataset."

spec:
  # 任务类型，Worker会根据此字段加载对应模块
  taskType: "sft"

  # 资源需求，Orchestrator会根据此进行调度
  resources:
    replicas: 1 # 需要多少个Worker来执行
    hardware:
      cpu: "8"
      memory: "64Gi"
      gpu:
        type: "nvidia-a100-80gb" # 可以是具体型号，也可以是 "any"
        count: 4

  # 模型配置
  model:
    source:
      type: "huggingface" # 或 "s3", "local"
      identifier: "mistralai/Mistral-7B-Instruct-v0.1"
      revision: "main"
    # Adapter 配置 (LoRA, QLoRA)
    adapter:
      type: "qlora"
      r: 16
      lora_alpha: 32
      lora_dropout: 0.05
      target_modules: ["q_proj", "v_proj"]

  # 数据集配置
```

```
dataset:
  source:
    type: "huggingface"
    identifier: "databricks/dolly-15k"
    split: "train"
  preprocessing:
    # 数据预处理逻辑的配置
    prompt_template: "..."
    max_seq_length: 2048

# SFT 训练超参数 (直接映射到 TRL SFTTrainer 的 TrainingArguments)
hyperparameters:
  output_dir: "/artifacts/sft-mistral-7b-dolly" # Worker内部的临时路径
  num_train_epochs: 3
  per_device_train_batch_size: 4
  gradient_accumulation_steps: 2
  learning_rate: 2e-4
  logging_steps: 10
  fp16: true

# 任务产物输出配置
output:
  destination:
    type: "s3" # 或 "huggingface"
    bucket: "my-ml-models"
    path: "checkpoints/sft-mistral-7b-dolly-final"
  # 输出类型，可以是 adapter_weights, full_model, logs 等
  artifacts:
    - "adapter_weights"
    - "training_logs"
```

6. 工作流详解 (Detailed Workflow)

一个完整的端到端流程：

1. 部署 **(Deployment)**:
    - 管理员使用 `kubectl apply` 部署 MLOC 的 Orchestrator 和一组 Worker Pool (Deployment/StatefulSet)。Orchestrator 是一个单实例服务，Worker 可以是多个实例。
2. 初始化 **(Initialization)**:
    - **Orchestrator** Pod 启动，开始监听 API 请求。
    - **Worker** Pod 启动后，执行 `Lifecycle Manager`：

        - 检测自身硬件环境（`nvidia-smi` 等）。
        - 向 Orchestrator 或服务发现系统注册，报告自己的 ID、地址和硬件信息。状态变为空闲 (`IDLE`)。
3. 任务提交 **(Task Submission)**:
    - 用户通过 `curl` 或客户端调用 Orchestrator 的 API，提交上述 `task.yaml` 文件。
    - `POST /api/v1/tasks`
4. 任务调度 **(Scheduling)**:
    - Orchestrator 的 `Task Parser` 验证并解析 YAML。
    - `Scheduler` 查找符合 `resources.hardware` 要求的、状态为 `IDLE` 的 Worker。
    - 如果找到，`Scheduler` 将任务（可以简化为 JSON 格式）发布到消息队列的特定主题（Topic）中，并将 Worker 状态更新为 `RUNNING`。
5. 任务执行 **(Execution)**:
    - Worker 的 `Task Listener` 监听到新任务。
    - `Resource Downloader` 根据 `model` 和 `dataset` 配置，从 Hugging Face 或 S3 下载所需资源到本地临时目录。
    - `Module Loader` 根据 `taskType: "sft"`，实例化 `SFTModule`。
    - `SFTModule` 读取配置中的 `hyperparameters`，初始化 TRL 的 `SFTTrainer`。
    - 训练开始。`Execution Engine` 定期向 Orchestrator 发送心跳和进度更新。
6. 完成与返回 **(Completion & Return)**:
    - 训练完成后，`Result Uploader` 将 `/artifacts/` 目录下的产物（LoRA 权重、日志等）根据 `output` 配置上传到目标 S3 桶。
    - Worker 向 Orchestrator 发送“任务完成”消息。
    - Orchestrator 将任务状态标记为 `COMPLETED`，并将 Worker 状态重新设为 `IDLE`，等待下一个任务。

8. 数据、通信协议与广域网考量

当 Worker 节点分布在不同的数据中心或广域网（WAN）时，数据传输效率和通信稳定性成为关键挑战。MLOC 采用以下协议和原则来应对：

1. 数据拉取协议 **(Data Pulling Protocol)**:
    - 原则: Orchestrator 绝不直接向 Worker 传输大型数据（如模型权重、数据集）。所有的数据传输都由 Worker 主动发起。
    - 流程: Orchestrator 在任务指令中只包含数据的 元信息（Metadata），例如：
        - 模型：Hugging Face Hub 的 `repo_id` 和 `commit_hash`。
        - 数据集：S3 存储桶的 `bucket_name` 和 `object_key`。
        - 代码：Git 仓库的 URL 和 `branch/tag`。

    - 实现: Worker 内的 `Resource Downloader` 组件负责解析这些元信息，并使用相应的客户端（`huggingface_hub`, `boto3`, `git`）从最近的或指定的中央数据源安全地拉取数据。
    - 优势:
        - 带宽优化: 避免了 Orchestrator 成为网络瓶颈。
        - 可靠性: 利用了专业存储服务提供的高可用和断点续传能力。
        - 安全性: 可以为每个 Worker 配置独立的、有时限的数据源访问凭证。
2. 异步通信协议 **(Asynchronous Communication Protocol)**:
    - 中心节点: Orchestrator 作为唯一的中心通信节点，负责所有任务的协调和状态同步。Worker 之间不进行直接通信。
    - 消息队列: 使用消息队列作为 Orchestrator 和 Worker 之间的缓冲层，可以有效应对 WAN 的高延迟和不稳定性。即使 Worker 短暂断开连接，任务指令也不会丢失，Worker 重新上线后可以继续消费。
    - 心跳机制: Worker 通过定期向 Orchestrator 发送心跳来报告其存活状态和任务进度，Orchestrator 可以据此判断 Worker 是否失联并进行任务重调度。

9. 监控与用量统计

1. 系统监控:
    - 使用 `Prometheus` 收集时间序列指标，如 GPU 利用率、内存使用、任务队列长度等。
    - 使用 `Grafana` 对指标进行可视化，创建仪表盘。
    - 使用 `Loki` 或 `ELK` 堆栈聚合所有容器的日志，便于调试和审计。
2. 用量统计接口 **(Usage Statistics API)**:
    - 目的: 为系统管理员或用户提供查询资源消耗情况的接口，用于成本分摊、预算控制和性能分析。
    - 数据收集: Orchestrator 的 `State & Usage Monitor` 在任务开始和结束时记录关键信息，例如：
        - `task_id`, `user_id` (来自 `metadata.owner`), `project_id`
        - `start_time`, `end_time`, `duration`
        - `gpu_type`, `gpu_count`
        - 计算并存储 `gpu_hours` (`duration_in_hours * gpu_count`)

API 端点: Orchestrator 的 `API Server` 将暴露一个新的只读端点，例如：
GET /api/v1/stats

    -
    - 查询参数:
        - `user=<user_id>`: 查询特定用户的用量。
        - `project=<project_id>`: 查询特定项目的用量。
        - `start_date=<YYYY-MM-DD>` & `end_date=<YYYY-MM-DD>`: 按时间范围查询。

返回示例:


```
{
  "query_parameters": {
    "user": "john-doe",
    "start_date": "2025-09-01",
    "end_date": "2025-09-05"
  },
  "total_gpu_hours": 128.5,
  "total_tasks_completed": 15,
  "breakdown_by_gpu": {
    "nvidia-a100-80gb": 100.0,
    "nvidia-l40": 28.5
  }
}
```


-