> - 轨迹片段1: `(输入: U, 输出: Q, 奖励: R_final)`
> - 轨迹片段2: `(输入: (U,P), 输出: A, 奖励: R_final)`
>
> `LightningRL` **的操作**: 将这条长轨迹**分解**成多个独立的训练样本。
>
> - **训练样本1**: `{ input: U, output: Q, reward: R_final }`
> - **训练样本2**: `{ input: (U,P), output: A, reward: R_final }`
>
> 现在请看这两个训练样本，它们的格式**完美地符合了单步骤RL算法的输入要求**！每一个样本都有一个清晰的 `(输入, 输出, 奖励)` 对。
>
> **应用标准RL算法**: 接下来， `LightningRL` 就可以把这些独立的训练样本直接"喂"给任何标准的单步骤RL算法（比如GRPO）。

2.4 系统设计：训练-Agent分离架构

- **解耦设计：** 引入了训练- Agent 分离架构（如图4所示），将计算密集型LLM生成（由RL框架管理，并暴露 OpenAI-like API）与轻量级、多样化且灵活的应用逻辑和工具（在传统编程语言中编写，由 Agent 独立管理和执行）完全解耦。

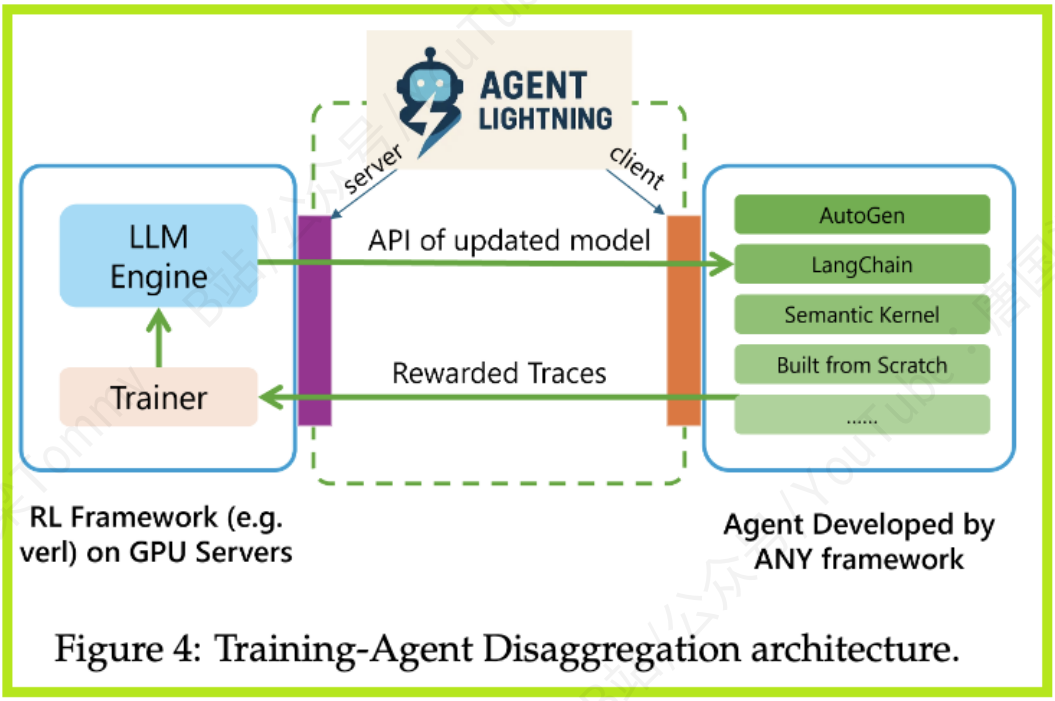


Figure 4: Training-Agent Disaggregation architecture.

- **两组件架构：** Agent Lightning 包含 Agent Lightning Server 和 Agent Lightning Client 。
  - **Lightning Server：** 运行在GPU服务器上，负责模型训练、任务调度。
    - **职责**：负责所有重活。它管理着需要优化的LLM，运行强化学习算法（如PPO），处理数据，更新模型权重。
    - **部署位置**：通常部署在拥有强大GPU资源的云服务器上。

- **Lightning Client**： 在用户环境中运行，负责执行Agent、收集数据。

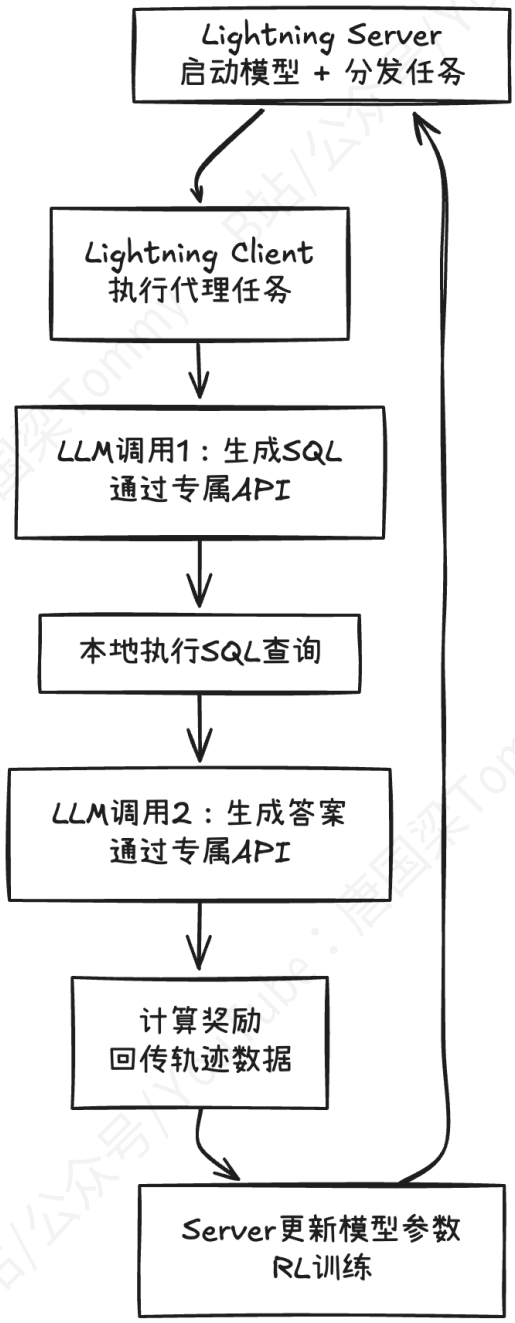
  - **职责**：负责所有应用逻辑。它运行开发者编写的Agent代码（比如用LangChain或AutoGen写的代码），调用工具，并与Server通信。
  - **部署位置**：可以运行在任何地方，比如一台普通的虚拟机，甚至开发者的本地电脑上。

举例说明：

案例：优化一个“数据库问答Agent”

假设我们有一个用LangChain构建的 AI Agent，它的任务是根据用户的自然语言问题，查询公司的销售数据库并给出答案。

任务：用户提问：“上个月北京分公司的A产品销售额是多少？”



第1步：启动与任务分发

- **在训练端 (Lightning Server)**：
  1. 你在GPU服务器上启动`Lightning Server`。
  2. Server加载了基础的`Llama-3.2-3B-Instruct`模型。
  3. 你上传了一个任务数据集，里面包含很多问题和标准答案，比如`("上个月北京分公司的A产品销售额是多少？", "58,000元")`。
- **通信开始**：
  1. Server从数据集中取出一个任务（问题和答案）。
  2. 它为这个任务**生成一个唯一的、临时的API端点**，例如`http://<server_ip>:8000/v1/chat/completions?task_id=xyz123`。
  3. Server将这个问题和这个专属API地址，一同发送给一个待命的`Lightning Client`。

第2步：Agent在客户端执行

- **在Agent端 (Lightning Client)**：
  1. Client收到了任务和API地址。
  2. 它调用本地的Agent函数，这个函数是用LangChain写的，流程是“生成SQL -> 执行SQL -> 根据结果生成答案”。
  3. **第一次LLM调用 (生成SQL)**：Agent需要调用LLM来把“上个月北京分公司的A产品销售额是多少？”转换成SQL语句。它**不会去调用OpenAI的官方API，而是调用刚才Server给它的专属API地址**（`...task_id=xyz123`）。
  4. **数据自动被捕获**：Server收到这个请求后，它就知道这是`task_id=xyz123`的第一次LLM调用。它用自己管理的Llama-3模型生成SQL：`SELECT SUM(sales) FROM sales_records WHERE product = 'A' AND city = '北京' AND sale_date >= '2025-07-01';`。**关键在于，Server在将SQL返回给Client之前，已经默默记录下了这次调用的输入（prompt）和输出（SQL）。**
  5. **工具调用**：Client的Agent收到SQL后，在**本地**调用数据库工具，执行查询，得到结果`58000`。这个过程完全不涉及Server。
  6. **第二次LLM调用 (生成答案)**：Agent现在需要根据查询结果`58000`生成自然语言答案。它再次调用那个专属API，输入类似“根据查询结果58000，回答用户问题”。
  7. **数据再次被捕获**：Server收到请求，生成答案：“上个月北京分公司的A产品销售额为58,000元。”同时，它再次记录下这次调用的输入和输出。

第3步：奖励计算与数据回传

- **在Agent端 (Lightning Client)**：
  1. Agent执行完毕，得到了最终答案。
  2. Client将Agent的答案“58,000元”与任务里的标准答案“58,000元”进行比对，发现完全正确。于是计算出最终奖励（Reward）为`1.0`。
  3. Client将这次任务的**完整执行轨迹（Trace）和最终奖励**打包，发回给Server。这个轨迹看起来像这样：

```
{
  "trace": [
    { "input": "...", "output": "SELECT ..." },
    { "input": "...", "output": "销售额为58,000元。" }
  ],
  "reward": 1.0
}
```

第4步：模型在服务器端训练

- **在训练端 (Lightning Server)**：
  1. Server收到了这条带有高奖励的成功轨迹。

2. 它内部的RL训练模块（LightningRL算法）启动工作，将最终奖励`1.0`分配给轨迹中的每一次LLM调用。
3. 然后，它使用这些`(输入, 输出, 奖励)`数据对`Llama-3.2-3B-Instruct`模型进行一次参数更新（梯度下降）。这次更新会使模型在未来更倾向于生成能够得到高奖励的SQL和答案。
4. 模型更新完毕。Server准备好处理下一个任务，此时它内部的LLM已经是“微调”过的新版本了。

这个循环不断重复，Agent通过在客户端的“实战”产生数据，服务器端的模型则根据这些数据不断“进化”。

三、系统架构设计

3.1 架构设计

Agent Lightning采用**分布式训练架构**：

- **Agent Lightning Platform**

- **Agent Frameworks**