> 这个过程就是一个**顺序决策（Sequential Decision-Making）**过程。Agent 在每一步都需要根据当前掌握的信息，做出一个“动作”（调用LLM生成内容），然后环境会发生变化（掌握了新信息），并可能得到一个“反馈”（比如发现某个景点关门了，这是一个负反馈）。

2.3 分层RL算法：LightningRL

该算法用于解决“宏观动作”（整个LLM输出）与“微观动作”（token生成）之间的优化问题，它将多轮次的Agent优化问题分解为多个单轮次问题。

> 整篇论文的算法核心，它巧妙地解决了一个关键难题：**如何用为“单次对话”设计的强化学习算法，来训练一个需要进行“多次对话”的复杂 Agent？**

- **步骤一：信用分配**。首先，将整个执行轨迹的最终奖励R分配给轨迹中的每一次LLM调用（即每个宏观动作）。在最简单的实现中，每次调用的奖励都直接设为R。

  > 局限性：这种策略虽然简化了训练过程，但对于更复杂的长序列决策或需要更精细控制的任务，可能无法提供最优学习信号。未来的工作方向是引入高层价值函数等更复杂的信用分配策略来估计每个动作的预期回报。

- **步骤二：分解与应用** 。接着，将多步骤问题转化为单步骤问题。即将每个`(input, output, reward)`的转换视为一个独立的单轮次RL任务。然后可以使用任何现成的单轮次RL算法（如PPO、GRPO、REINFORCE++）来计算token级别的损失并更新模型参数。

  > - **图3(b) - 过去的方法**: 以前的做法是把多次调用**拼接**成一个超长的序列，然后用一个复杂的“掩码（masking）”告诉模型哪些部分需要学习，哪些部分需要忽略。这种方法不仅实现复杂、容易出错，而且会产生非常长的序列，训练效率低下。
  > - **图3(c) - `LightningRL`**：`LightningRL` 的做法则优雅得多。它不拼接，而是**分解**。它通过“信用分配”模块将奖励赋予每个独立的调用（transition），然后将这些调用作为独立的样本进行训练。

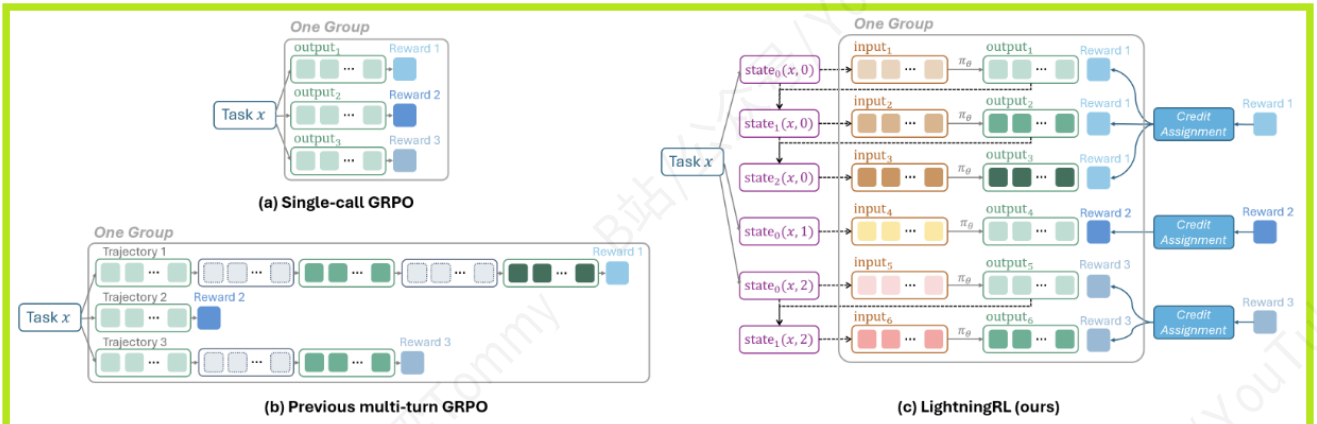

Figure 3: Illustration of the LightningRL algorithm. **(a)** Single-call GRPO, where the LLM generates a response to a task in one pass. Outputs for the same task are grouped together for advantage estimation. **(b)** Previous multi-turn GRPO. Each trajectory contains multiple LLM calls, with trajectories for the same task grouped for advantage estimation. Tokens not generated by the LLM are masked (gray dashed boxes) during optimization. **(c)** Our proposed LightningRL. Trajectories are decomposed into transitions, and transitions for the same task are grouped for advantage estimation. Each transition includes the current input/context, output, and reward. The input is part of the current agent state, with rewards computed by the credit assignment module.

举例说明：

假设一个RAG Agent 有两次LLM调用：

- **Call 1 (Query Generation)**: 输入`U` (UserInput)，输出`Q` (Query)。
- **Call 2 (Answer Generation)**: 输入`(U, P)` (UserInput, Passages)，输出`A` (Answer)。

整个任务完成后，我们得到了一个最终奖励`R_final`（比如答案的F1分数）。

步骤一：信用分配 - 解决"功劳归谁"的问题

问题: 最终的奖励`R_final`到底是谁的功劳？是第一次调用生成了一个好查询？还是第二次调用做了一个好总结？还是两者都有？这就是经典的"信用分配"问题。

`LightningRL` 的简化策略: 论文提出了一种非常简单但有效的策略，叫做**"相同值分配" (Identical Assignment)**。

它的思想是：既然我们无法精确区分每个步骤的功劳，那就干脆认为**最终的成功（或失败）是整个执行链条上所有步骤共同努力的结果**。

- **具体操作**: 将最终的奖励`R_final`分配给路径上**每一次**LLM的调用。
- **在RAG例子中**:
  - Call 1 (`Q`的生成) 获得的奖励被认为是`R_final`。
  - Call 2 (`A`的生成) 获得的奖励也被认为是`R_final`。

现在，我们原本只有一个最终奖励的执行轨迹，变成了一个每一步都有明确奖励的轨迹。

步骤二：分解与应用 - 将多步骤问题转化为单步骤问题

经过了信用分配，我们的执行轨迹现在看起来是这样的：

> - 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。

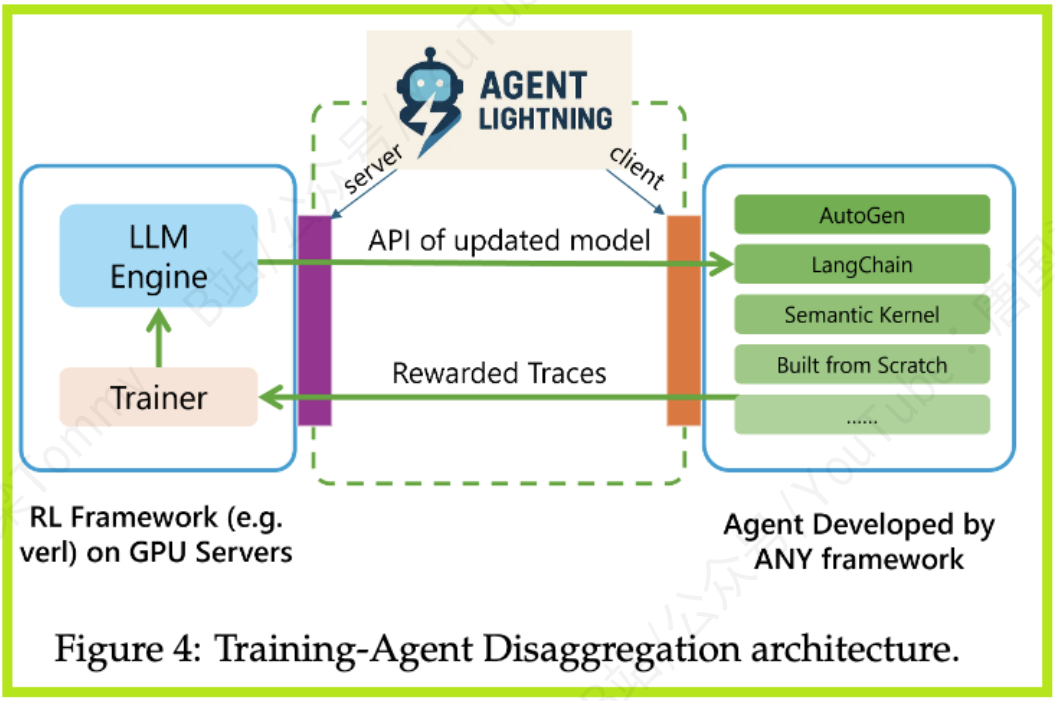

Figure 4: Training-Agent Disaggregation architecture.

- **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。