AI智能体协同工作框架设计

在电力行业场景中，多个专业模型需要协同完成复杂任务。以下是一个详细的协同框架设计方案：

模型协同架构

1. 任务编排层

工作流编排引擎：

- 基于DAG(有向无环图)的任务流程定义
- 支持条件分支、并行执行和错误处理
- 任务依赖关系和数据流管理

实现技术：

- Airflow/Prefect/Temporal作为工作流引擎
- YAML/JSON格式定义工作流程
- 可视化编排界面支持拖拽式流程设计

2. 通信中间件

消息队列：

- 模型间异步通信
- 任务状态和结果传递
- 缓冲机制处理模型处理速度差异

共享存储：

- 模型间中间结果存储
- 大型数据对象传递(如图片、文档)
- 向量表示存储与检索

3. 协同方式

管道模式

```
文本理解模型 → 方案撰写模型 → 图片生成模型 → PPT生成模型
```

- 每个模型完成自己的任务后将结果传递给下一个模型
- 适合有明确顺序依赖的任务

Agent模式

- 中央协调器管理多个模型Agent
- 每个Agent具有特定能力(文本理解、图像生成等)
- 协调器根据任务需求动态调用Agent
- 支持反馈循环和多轮交互

实现方案

示例：自动生成电力项目汇报PPT

```
1. 协调器接收任务：「生成XX电站季度运营报告PPT」
2. 文本理解模型解析任务要求(主题、重点、风格)
3. 数据分析模型处理电站运营数据
4. 方案撰写模型生成报告大纲和内容要点
5. 协调器将内容分发给专业模型：
   - 图表生成模型创建数据可视化
   - 图像生成模型创建配图
   - 文本撰写模型完善各章节内容
6. PPT生成模型整合所有资源，应用模板
7. 审核模型检查最终结果
```

技术实现细节

1. 统一接口标准

```
{
  "task_id": "unique-id",
  "model_id": "text-writer-001",
  "input": {
    "context": "...",
    "requirements": "..."
  },
  "output": {
    "text": "...",
    "metadata": {}
  },
  "status": "completed"
}
```

2. 状态管理

- 任务状态跟踪(等待、执行中、完成、失败)
- 中间结果缓存
- 断点恢复机制

3. 错误处理与回退

- 模型执行失败时的回退策略
- 替代方案自动启动
- 人工干预接口

协同优化策略

1. 上下文传递

为确保一致性，每个模型需要获取足够上下文：

- 全局任务目标
- 已完成部分的结果
- 下游模型的需求

2. 参数共享机制

设计参数共享协议：

- 统一元数据格式
- 关键词和实体标记
- 风格和语调定义

3. 反馈循环

建立模型间反馈机制：

- 下游模型可请求上游模型澄清或补充
- 最终用户反馈可传递给相关模型
- 质量评估结果用于模型自我优化

实现示例代码框架

```
# 协调器核心逻辑
class TaskCoordinator:
    def __init__(self):
        self.models = {
            "text_understanding": TextUnderstandingModel(),
            "proposal_writer": ProposalWriterModel(),
            "image_generator": ImageGeneratorModel(),
            "ppt_creator": PPTCreatorModel()
        }
        self.task_queue = MessageQueue()
        self.result_store = SharedStorage()

    def execute_workflow(self, task_definition):
        workflow = self._parse_workflow(task_definition)
        context = {"task_id": generate_unique_id(), "objective": task_definition["objective"]}

        for step in workflow:
            model_id = step["model"]
            input_data = self._prepare_input(step, context)

            # 执行模型
            model = self.models[model_id]
            result = model.execute(input_data)

            # 存储结果
            self.result_store.save(f"{context['task_id']}:{step['id']}", result)

            # 更新上下文
            context[step["output_key"]] = result

        return self._compile_final_result(context)
```

部署架构

容器化微服务：

- 每个模型作为独立服务部署
- API网关统一管理模型服务
- 服务发现支持动态扩展模型池

资源管理：

- GPU资源动态分配
- 优先级队列管理任务执行顺序
- 自动扩缩容根据负载调整资源

监控与优化

性能监控：

- 模型执行时间跟踪
- 资源利用率监控
- 瓶颈识别

质量保障：

- 输出一致性检查
- 自动评估机制
- 人工审核接口

需要更详细的某个协同模式或具体实现方案吗？