科研创意系统技术架构与模块设计

白皮书

本白皮书提出一个具备**感知**、**推理**、**行动**、**规划**、**记忆**能力的科研创意多 Agent 协同平台的技术架构与模块设计方案。该系统支持*慢思考*（逐步深思熟虑）、*黑板机制*（共享工作记忆）、*链式思维*（Chain-of-Thought 推理链）和*协同反馈*（多 Agent 互评互助），以产出带有评分和排序的创意构想或实验方案列表。本方案面向科研创新场景，旨在整合多个专长各异的智能 Agent 协同工作，探索复杂问题领域的创新解法。

1. 系统架构设计

系统总体架构采用**多 Agent 协同**与**黑板系统**相结合的设计。图 1 展示了架构的概要示意：

图 1：多 Agent 科研创意系统架构示意。主 Agent（Lead/Coordinator）负责任务分解与调度，经由中心黑板与各种专用子 Agent 交互，包括信息检索 Agent、建模 Agent、验证 Agent、批判 Agent 等。黑板作为共享知识库，存储问题、部分解、假设和交互消息，用于 Agent 之间的异步协作。

主 Agent (协调者)：系统的核心调度者，相当于人类团队中的组长或专家组主持人。主 Agent 负责解析用户的科研创意需求或问题陈述，基于此**规划求解策略**和**任务分解**，并通过黑板发布子任务或直接唤起相应的子 Agent 执行。主 Agent 持续监控黑板上的进展，整合各 Agent 提供的部分成果，适时调整策略。主 Agent 的规划过程中引入*慢思考*机制：即不急于给出单步答案，而是进行多轮推理，在黑板上记录**链式思维过程**（如假设、推理步骤）以供所有 Agent 参考和监督。这保证了复杂问题求解的全面性和逻辑性。

子 Agent（知识源）：围绕主 Agent 部署多种类型的智能 Agent，扮演不同的专家角色，每类 Agent 擅长解决问题的某一方面。例如：

- *信息获取 Agent*：负责从文献、数据库或互联网检索信息（如“文献 Agent”专门查找学术论文，“专利 Agent”检索专利技术等）。
- *建模 Agent*：负责科学模型计算或模拟（如“DFT Agent”进行量子化学计算，“热力学 Agent”进行热力学分析建模等）。
- *验证 Agent*：负责验证假设或实验方案的可行性（例如运行代码实验、检查逻辑一致性，或调用实验室设备模拟等）。

- *批判 Agent*：负责对其他 Agent 提出的方案和结论进行审查和批判性评估，提出改进建议。
- 此外还有其他可能的专用 Agent，如数据分析 Agent、可视化 Agent、报告生成 Agent 等，根据需要扩展。

各知识源 Agent 通过读取和监听黑板上相关内容，判断何时应用自己的专长“登场贡献”。这一机制类似专家团队在公共黑板前协作：**黑板**上写下问题后，不同专家密切关注，当某个部分出现自己能解决的机会时，该专家就上前填上一笔。这种设计允许各 Agent 松耦合地**异步协作**，不需要点对点直接通信，就能逐步累积解题线索。黑板上信息的**事件驱动**更新会触发相关 Agent：“看到”自己擅长的问题出现时主动响应。例如，当黑板上出现“需要检索文献 X 领域”事件时，文献 Agent 会被唤醒执行检索并把结果写回黑板。

黑板系统：黑板是系统的**共享知识库**和**交流中心**，存储问题描述、当前状态、部分解、知识源 Agent 发布的中间结果、评论等。黑板采用**发布/订阅机制**实现事件驱动：每当主 Agent 或子 Agent 往黑板发布新信息（如提出子任务、写入发现事实、更新假设等），都会生成特定类型的事件。各 Agent 根据订阅的事件类型或关注的黑板内容模式决定是否读取并介入处理。由于所有 Agent 都以黑板为中心协作，系统避免了复杂的点对点通信拓扑，每个 Agent 专注于从黑板“拉取”自己需要的信息并“发布”结果。黑板机制还能记录 **Agent 的思维链**和决策依据，作为持续的**记忆**存储供后续步骤参考。这相当于一个全局的工作记忆，使得 AI 系统拥有*持久化的上下文*，避免短视行为。

控制与调度：为防止多个 Agent 同时争用黑板造成冲突，架构中引入**控制壳 (Control Shell)**或称调度器，由主 Agent 或专门的控制模块实现。控制壳监视黑板状态并决定何时、由哪个 Agent 执行下一步操作，相当于人类专家会议中的主持人，防止所有专家“一窝蜂”上前争抢黑板而产生混乱。调度策略可以基于事件优先级队列实现：黑板维护一个待处理事件/任务

的优先队列，按照重要性或提交顺序排列。当有空闲 Agent 时，从队列取出最高优先的事件指派给对应类型的 Agent 处理。如果多个 Agent 尝试同时写入黑板，控制模块也可锁定黑板或合并结果以冲突检测与消解。例如，如果两个 Agent 提出相互矛盾的假设，系统可触发批判 Agent 分析冲突并在黑板上给出裁定或让主 Agent 调整计划。

该架构支持**并行**与**异步**的协同：多个 Agent 可以各自处理不同子问题，透过黑板汇聚成果，从而加速复杂任务的求解。同时，黑板作为统一数据源确保所有 Agent 共享一致的世界观和最新进展，从而实现协同一致性。另外，通过黑板记录决策过程和中间数据，系统也方便引入人工监控与介入点：开发者或用户可以查看黑板上的思维链日志，必要时**人工插入提示**或更正信息，然后让 Agent 继续运行。这有助于提高系统可靠性和透明度。

2. 模块设计与功能说明

本节详细描述各模块（主要 Agent 和子 Agent 类别）的职责，以及**主 Agent 的任务规划/调度**机制和**黑板事件系统**的实现。

2.1 多 Agent 模块分类与职责

主 Agent（Planner/Coordinator）：负责**理解用户提示**（科研问题或创意需求），将其**拆解成子任务**并规划解题步骤。主 Agent 会率先在黑板上写入问题规范和初步计划，使其他 Agent 了解任务背景。例如，主 Agent 收到用户要求“设计一种高效催化剂合成实验方案”时，可能将任务分解为：“检索现有催化剂材料文献”、“提出候选材料理论模型”、“模拟材料热力学性质验证”、“设计实验步骤”、“评估创新性”等子任务。主 Agent 据此依次激活相应子 Agent 或发布任务事件到黑板，并**监听子 Agent 反馈**来动态调整计划。当所有子任务都有结果后，主 Agent 负责编译整合各部分成果，生成最终的创意方案列表并评分排序输出。

主 Agent 的**调度策略**灵活运用串行和并行机制：对于互相独立的子任务，可并行触发多个 Agent 同步工作；对于有先后依赖的任务，则按序等待关键结果再推进下一步。主 Agent 同时承担一定**元认知**功能：监控整个求解过程是否偏离方向，遇到僵局时可调整策略（例如改变问题分解方式或启用批判 Agent 寻找盲点）。主 Agent 的内部实现可以基于大型语言模型，通过精心设计的提示模板让其具备**自我链式思考**能力，即在做规划时自行生成多步思考过程并在黑板记录推理链，从而指导后续行动。这类似 Anthropic 的研究 Agent 中，让 LeadResearcher 先“思考”并保存计划，再启动子 Agent 执行。

信息获取类 Agent：负责从各种数据源获取信息的 Agent 集合，涵盖：

- *文献 Agent*：检索学术论文、书籍等文献资料，提供研究背景和最新进展摘要。
- *专利 Agent*：检索专利数据库，寻找相关技术方案的专利信息。

- *网络 Agent*：通过互联网搜索引擎获取新闻、网页数据，或调用 API 获取实时数据等。
- *数据 Agent*：查询内部数据库或开放数据集，获取实验数据或统计资料。

这类 Agent 接收主 Agent 发布的检索任务（如黑板事件："需要关于 X 的文献综述"），使用自身工具 (如文献数据库 API) 执行查询，将结果 (摘要、引文等) 写回黑板。信息获取 Agent 应遵循统一的输出格式，便于后续处理。例如文献 Agent 返回时可以包含**来源引用**，以便系统最终输出包含出处的陈述。

建模与分析类 Agent：在科研场景中，很多创新需要借助理论模型或计算工具验证，可由建模类 Agent 承担：

- *DFT Agent*：调用量子化学密度泛函理论(DFT)计算软件，对候选材料或分子进行能量计算和性质预测。
- *热力学 Agent*：基于热力学和动力学模型，计算反应可行性、平衡常数、转化率等。
- *模拟 Agent*：运行数值模拟（如分子动力学、有限元分析）验证方案在近似环境下的表现。
- *数据分析 Agent*：对已有实验数据或文献数据进行统计分析，寻找规律支撑创意。

这些 Agent 通常通过编程接口调用复杂的外部工具或本地图形计算库，其工作往往耗时较长且需要异步执行。主 Agent 会根据需要触发建模 Agent，并可能在黑板上标记其任务进行中。建模 Agent 完成后，将计算结果、模型结论写入黑板，并附上必要的说明（例如"DFT 计算表明材料 A 的带隙为…，满足高催化活性要求"）。在黑板事件系统中，可为此设计**任务完成**事件，通知后续 Agent 结果已可用。

验证与实验设计 Agent：负责对提出的创意方案进行验证和细化：

- *验证 Agent*:对理论上的方案进行可行性验证,方式包括交叉验证信息、一致性检查等。例如验证 Agent 可检查不同来源的信息是否矛盾，或根据已有知识规则评估方案是否现实（比如所需条件是否可实现）。
- *实验设计 Agent*：将概念方案转化为具体可执行的实验流程，包括实验步骤、所需材料设备、测量指标等细节，确保创意具有可操作性。
- *安全/合规 Agent*：在某些应用中，可以有 Agent 检查方案的安全性、伦理和合规性，防止不合规或危险方案。
- *报告生成 Agent*：将最终方案和论证整理成文档或报告格式，方便人类阅读。

验证 Agent 通常在黑板上监视其他 Agent 的输出，当检测到潜在错误、不一致或值得质疑之处时，发布批评或警告信息。例如验证 Agent 可能发现两个模型 Agent 给出的结果矛盾，便在黑板上发出冲突事件提示主 Agent 注意。实验设计 Agent 则在黑板上看到“方案骨架”后开始补充细节，输出一份结构化的实验方案草案。

批判 Agent（Critic Agent）：批判 Agent 专门用于**评价和改进**其他 Agent 的工作成果。它相当于 AI 团队中的审稿人或导师，对阶段性结果给出客观评论和改进建议：

- 批判 Agent 会持续关注黑板上的主要中间结论和最终方案草稿，对其进行审阅。例如检查推理逻辑是否严谨，是否存在知识盲区，结论是否有过度推断等。
- 对发现的问题，批判 Agent 直接在黑板上留言反馈，并可能打低分或者建议进一步调查某方面。例如批判 Agent 可能指出：“方案中假设 X 缺少文献支持,建议文献 Agent 补充相关研究”。
- 批判 Agent 也可充当**评分者**之一，对每个创意方案按照预定维度打分（见第 3 节），为最终排序提供参考。

批判 Agent 的实现可以是基于 LLM 的评审 Prompt，具备一定领域知识和批判思维。通过它的存在，系统形成一个**自我审阅**循环，类似人类头脑风暴后进行 peer review，提高创意质量和可靠性。

2.2 主 Agent 的规划与调度机制

主 Agent 作为核心调度者，其关键在于**理解任务、分解任务并高效调配 Agent**。当接收到用户提示后，主 Agent 典型的工作流程为：

1. **任务解析**：利用大模型对用户输入进行语义分析，提取关键目标和约束。例如用户要求可能隐含领域、目标指标等，主 Agent 首先明确这些要素并在黑板上记录对任务的理解和解决思路 (这一步往往以 Chain-of-Thought 形式在内部生成，但最终概要写入黑板以共享）。
2. **子任务分解**：根据任务复杂度和不同方面，将问题划分为可并行或顺序的子问题。主 Agent 会在黑板上发布这些子任务项，格式可以类似“任务 1: [类型] 查找…；任务 2: [类型] 计算…”。每个任务都标明类别以便相应 Agent 认领。例如“任务 1: 信息检索—查找当前最有效的催化剂材料”和“任务 2: 模拟计算—评估这些材料的热稳定性”。
3. **调用子 Agent**：针对每个子任务，主 Agent 选择适合的 Agent 执行。如果采用直接调用模式，主 Agent 会通过系统接口触发该 Agent 运行并传入任务参数；如果采用黑板驱动模式，则仅需在黑板上正确发布任务描述，订阅了该任务类型的 Agent 自行运行。实际实现中两种方式可以结合：对于关键任务主 Agent 直接调用（确保及时执行），对一般任务则依赖 Agent 监听黑板事件异步响应。
4. **同步与等待**：主 Agent 在触发子任务后，可以选择等待所有并行任务完成，也可以实施**迭代式规划**——即收到部分结果就提前分析并规划下一步。例如 Anthropic 的多

Agent 研究系统中，LeadResearcher 在收到第一批 Subagent 结果后就判断是否还需额外信息，决定是否创建更多 Subagent。本系统主 Agent 也应具备这种动态调整能力：每当黑板上有重要更新（事件），主 Agent reevaluate 全局计划，看是否需新增任务、修改顺序或提前终止某分支。

5. **汇总与产出**：当所有必要子任务完成后，主 Agent 汇总黑板上的知识和方案，由 *Aggregation 模块*将分散信息整合成完整的创意方案列表。主 Agent 在此基础上调用评估机制 (可调用批判 Agent 或评估 Agent) 对方案进行评分排序，然后输出给用户。输出时可以附带简要说明每个方案的亮点或依据。

整个过程中，**黑板事件系统**扮演了主线调度的通信骨干。需要重点设计以下机制：

- **事件类别**：定义清晰的事件类型来标识黑板上发生的变化，如“任务发布”、“任务完成”、“信息更新”、“冲突警告”、“方案草案生成”、“方案评审反馈”等。每类事件对应特定处理逻辑。例如“任务完成”事件由主 Agent 或发布该任务的 Agent 监听，用于推进后续步骤；“冲突警告”事件通常触发批判 Agent 介入分析。
- **事件过滤与订阅**：每个 Agent 在初始化时声明感兴趣的事件类型或黑板内容模式。例如文献 Agent 订阅“检索请求”事件，批判 Agent 订阅“方案草案生成”事件等。黑板实现上可以为每类事件维护订阅者列表，也可以让 Agent 周期性扫描黑板变化。为了效率更高，推荐采用发布/订阅模型配合**消息队列**（如 Redis streams）来实时通知。
- **优先级与调度**：不同事件重要性不同，需要引入优先级队列。在大量事件同时发生时，控制壳先处理高优先事件。例如“冲突警告”应优先于新的普通任务，以尽快解决问题。同理，主 Agent 可以根据任务紧迫性调整对不同子任务的等待时间。如果某低优先任务超时未完成，主 Agent 可能选择继续后续流程而不等待，以提高效率（或派发给备用 Agent 重试）。

- **冲突检测**：黑板作为共享空间，有可能出现不同 Agent 写入内容相互矛盾。例如两个 Agent 提出不同的实验方案路径。此时黑板可标记潜在冲突，由主 Agent 或批判 Agent 分析处理。冲突检测也包括资源冲突（两个 Agent 试图同时修改同一黑板条目）。可以通过乐观锁或版本控制解决并发写入，也可以在架构上规定某些敏感数据只有主 Agent 能最终确认写入。
- **事件驱动与持续运行**：整个系统按照事件驱动循环运行，各 Agent 闲时等待新事件而非占用资源。这样设计具有良好的可扩展性和鲁棒性：当任务规模扩大，只需增加 Agent 实例并行消费事件队列即可，系统性能随 Agent 数量线性扩展；当某 Agent 失败或卡顿时，不会阻塞主线，控制壳可以超时检测并重新派发任务或替换 Agent，从而具备容错能力。

图 2：多 Agent 系统的事件驱动工作流程示意（参考 Anthropic Research 架构）。主 Agent（LeadResearcher）收到用户查询后进入迭代研究循环：首先在 Memory（黑板）中存储研究计划，然后创建多个专门 Subagent 并行检索不同方向的信息。各 Subagent 独立执行搜索和工具调用，将发现结果返回主 Agent。主 Agent 汇总信息，判断是否需要进一步研究（若是则生成新任务或 Subagent，如图中省略的循环箭头），否则进入收尾阶段：调用 CitationAgent 为报告添加引用，最终生成带出处的研究报告答复用户。

图 2 所示流程体现了**黑板记忆**与**链式思考**的作用：主 Agent 将初始计划写入共享 Memory 以便随时检索（防止上下文窗口限制导致遗忘），并在每轮迭代中利用“思考-行动-反馈”的循环逐步完善方案。这样的慢思考迭代机制确保系统不会匆忙给出答案，而是充分调查多方面证据。在我们的设计中，黑板 Memory 由数据库和缓存支撑（详见第 4 节），可以永久保存有价值的中间产物，后续类似任务还能复用，真正实现**知识积累**和**持续学习**。

2.3 黑板事件系统示例

为更具体地说明黑板事件系统如何运作，下面举一个简化的示例场景：

用户问题：“提出一种提高锂电池能量密度的创新方案。”

Step 1 – 主 Agent 在黑板发布问题和初步任务：

[问题] 如何提高锂电池能量密度?

[计划] 子任务：

1. 文献调研当前锂电池能量密度瓶颈（信息检索）
2. 提出可能的材料/结构创新思路（创意生成）
3. 模拟验证该思路在容量上的提升（建模计算）

发布事件：任务发布: 文献调研锂电池瓶颈

Step 2 – 文献 Agent 订阅到该事件，从资料库检索相关综述文章，将结果写入黑板：

[文献结果] 当前瓶颈在于正极材料比容量有限，例如 NMC 材料克容量~200 mAh/g。

发布事件：信息更新: 文献调研结果可用

Step 3 – 创意 Agent（可理解为一个擅长头脑风暴的新想法产生 Agent）检测到有文献结果且任务 2 等待执行，于是读取瓶颈信息，产生一个创意：

[创意方案] 引入高电压富锂正极材料并优化电解液，提高能量密度。

发布事件：方案草案生成: 富锂正极+电解液改进

Step 4 – 批判 Agent 订阅到“方案草案”事件，审查方案合理性。它发现方案涉及高电压可能导致电解液分解，指出这一问题：

[批判] 富锂正极提高能量但高电压下电解液稳定性存疑，需验证。

发布事件：批判反馈: 需验证电解液稳定性

Step 5 – 主 Agent 看到批判反馈，决定增加一个验证子任务，于是在黑板新增：

[计划更新] 增加任务 4: 模拟电解液在高电压下的稳定性（建模 Agent）

发布事件：任务发布: 模拟电解液稳定性

Step 6 – 建模 Agent 监听到模拟任务，调用化学模拟工具，完成后写回结论：

[模拟结果] 模拟显示常规电解液在 4.5V 下快速分解，需换用高稳定性电解液配方。

发布事件：信息更新: 模拟结果可用

Step 7 – 创意 Agent 订阅到新的模拟结果，更新方案：

[创意方案 v2] 使用富锂正极+高稳定电解液（如含氟溶剂）组合，预计能量密度提升 20%且循环稳定。

发布事件：方案草案生成: 富锂正极+含氟电解液

Step 8 – 批判 Agent 再次审视新方案，认为合理，打分 8/10 并无重大批评，发布：

[批判] 新方案考虑了电解液稳定性，创新度高，可行性良好。

[评分] 方案评分=8/10

发布事件：批判反馈: 方案通过

Step 9 – 主 Agent 检测到所有任务完成且方案通过审核，于是进入收尾：汇总最终方案，附上根据评分排序的列表（此示例只有一项方案），生成回答给用户，包括引用的文献与模拟数据来源。

通过上述场景可以看到，黑板事件系统使各 Agent 各司其职又衔接有序：任务发布->信息更新->方案生成->批判反馈形成闭环，最终由主 Agent 收敛输出结果。

3. Agent 评估与优化机制

在多 Agent 协同环境中，引入**评估与优化机制**有助于持续提升系统性能和创意质量。本系统为每类 Agent 建立一套**评估维度体系**，并设计 AI 驱动的反馈与自我改进循环，实现系统的**自适应进化**。

3.1 Agent 评估维度

我们从协作效率、结果质量和创新性等方面构建多维度的评估指标。主要评估维度包括：

- **知识贡献度**：衡量该 Agent 的输出对最终方案的贡献程度。可通过追踪黑板上的引用关系实现：例如最终方案引用了哪些 Agent 提供的信息或中间结论，引用次数或重要性作为贡献度评分。贡献度高的 Agent 应是提供关键见解、推动方案进展的。
- **准确性**：衡量 Agent 输出内容的可靠与正确程度。对于信息 Agent，准确性指检索内容的真实性、权威性；对于建模 Agent，指计算结果精度和可信度；对于创意 Agent，则指提出的方案是否合乎科学原理、不违背常识。准确性可通过事后验证（如与权威数据对比）或由批判 Agent 评判得到。
- **响应效率**：包括**响应时间**和**资源利用率**两方面。响应时间考察 Agent 从接收任务到完成输出所用时间，占用的计算资源也纳入考量（例如 CPU/GPU 时间、API 调用次数等）。系统应记录每次任务的耗时和资源，用以评估 Agent 的效率。比如文献 Agent 检索同样信息用时较长，则效率评分低。
- **协作度**：考察 Agent 在团队协作中的配合程度，包括信息交流的及时性、结果格式规范度、是否遵循黑板协议等。良好的协作度体现为 Agent 输出结构清晰易被他人利用，能正确触发后续 Agent 行为，且不会频繁引入误导信息打断流程。
- **创新度**：尤其针对创意/推理类 Agent，评估其提出想法的新颖性和独创性。创新度可由**多 Agent 交叉投票**或引入评价 Agent 来主观打分。例如，批判 Agent 可以一个维

度专门给出“创意新颖性”评分。如果有多个方案，也可比较该 Agent 方案与已有方案在思路上的差异度。理想情况下，多 Agent 协作应当能产生**涌现性的新见解**（emergent novel insights），即通过共享智能涌现出单个 Agent 无法独立产生的创新。

此外，还有一些全局维度如**鲁棒性**（是否容易出错）、**安全合规**（输出有无违规内容）等，可根据具体应用需求加入评估体系。

3.2 评估反馈机制

评估机制由**主 Agent 或专门的评估 Agent** 执行，对各 Agent 定期或每次任务完成后进行评价打分，并将反馈结果用于优化：

- **日志记录与分析**：系统维护详尽的运行日志，包括黑板交互记录、Agent 响应时间、错误情况等。评估 Agent 定期扫描日志，从中提取评估指标数据。例如计算每个 Agent 平均响应时长、产出错误率，累计贡献的有效信息量等。对定量指标，可直接归一化成评分；对定性指标（如创新度），则通过预设规则或让 LLM 根据日志内容打分。
- **即时反馈**：对于明显的问题，评估 Agent 可以即时反馈给相关 Agent。例如发现某 Agent 多次提供相同信息造成冗余，可发送提醒；或发现某 Agent 输出错误率高，可以降低其在任务分配中的优先级。即时反馈也可写入黑板，供主 Agent 和其他 Agent 参考，从而在后续过程里调整对该 Agent 的信任度。
- **周期考评**：在一个任务完成后，主 Agent 会综合各维度对参与 Agent 做出一次任务内评价，并存储到 Agent 档案中。长期来看，可以计算 Agent 的历史平均表现，识别“高绩效”Agent 和“瓶颈”Agent。比如，假设文献 Agent A 平均贡献度远高于 Agent B，则系统在未来检索任务更倾向调用 A。

- **协同反馈**：评估不光来自中心，Agent 之间也可互评。例如批判 Agent 对所有 Agent 的表现给出评语和分数，这些内容记录在黑板或日志中。这种**多 Agent 反馈**形成更全面的评价视角，有研究表明利用多智能体互相批评可以提升整体表现。

所有评估结果最终汇总由主 Agent/评估 Agent 审阅，并用于指导系统优化。需要注意平衡及时反馈和不干扰正常协作：评估 Agent 应尽量在不干扰任务执行的情况下收集数据、提供反馈，重大问题才中断流程处理。评估周期也可以根据任务长短动态调整。

3.3 自主优化与自我进化

有了对 Agent 的客观评估，下一步是**自动优化**机制，使系统能根据评估结果自我改进，实现*闭环进化*。主要策略包括：

- **Prompt 优化**：许多 Agent（尤其基于 LLM 的 Agent）的行为由 Prompt 决定。系统可以利用评估反馈自动调整 Prompt。例如，如果批判 Agent 反馈某 Agent 经常曲解任务，那么可以改进该 Agent 的提示词，明确其职责和输出格式。这种 Prompt 迭代可通过 AI 辅助完成：让主 Agent 或专门的 Prompt 优化 Agent 读取低绩效 Agent 的历史对话，生成改良的提示模板供开发者审核应用。
- **策略调整**：主 Agent 可以根据评估结果调整调度策略。例如降低表现不佳 Agent 的并发数量，或者在任务拆解时引入冗余（多个 Agent 并行执行同任务取最优）来缓解个别 Agent 失误的风险。如果发现某类任务始终由某 Agent 完成质量最高，主 Agent 可直接指定其为首选处理者，减少资源浪费。
- **Agent 替换或重组**：对于持续低效或不胜任的 Agent，系统可尝试**替换**或增加新的 Agent 模块。例如，如果开源的模型 Agent 效果不佳，可以切换调用更强大的 API 服务；或者引入第二个不同架构的 Agent 与原 Agent 竞争，从中择优。由于架构基于标准接口协议 (见第 6 节)，替换 Agent 相对容易，只需注入符合接口的新 Agent 即可。

- **自动代码改进**：进一步的自我进化是让 AI **自行改写 Agent 代码**以优化性能。这类似 Auto-GPT 展现的能力："Auto-GPT 可以阅读、编写并执行自己的代码，从而改进其程序"。在本系统中，也可配置一个"开发者 Agent"或由主 Agent 临时充当编程助手：当发现某 Agent 效率瓶颈或 Bug 时，自动生成修改该 Agent 代码的补丁或优化建议代码。例如如果某 Agent 排序算法低效，AI 可以重写为更优算法并测试验证。出于安全考虑，这种自动代码改进一般需要在人监督下执行，但在受控环境下可实现一定程度的**自我修复**和**优化**。
- **学习新的知识**：自我进化还包括知识更新。如果评估发现 Agent 知识盲区多（比如批判 Agent 指出某些领域知识不足），系统可以触发"学习 Agent"检索新资料充实知识库，或 fine-tune 相关 LLM 使其在后续任务中表现更佳。随着每次任务的进行，系统不断积累新知识，未来 Agent 可以调用这些内部知识，从而提升回答质量和创新概率。

通过以上机制，平台形成闭环：**评估→反馈→优化**。每轮任务后系统都会变得更智能、更高效。这种自我进化过程可以视为 AI 团队的"经验学习"，随着时间推移逐步逼近最优协作状态。当然，也要设置约束避免无效或有害的优化（例如 AI 错误修改了关键代码）。因此在人监督下逐步开放自主改进权限是必要的安全措施。

4. 开发框架与部署建议

实现上述多 Agent 协同平台，需要选择适当的前后端技术框架来满足**实时交互**、**异步并发**和**数据持久化**等需求。以下是本方案的开发框架和部署环境建议：

前端架构：采用 **React** 结合 **Tailwind CSS** 构建富交互界面。前端主要职责包括：可视化展示 Agent 的思维链和黑板内容、反馈各 Agent 状态、以及提供人工干预操作接口。React 适合构建动态 UI 组件来显示多 Agent 的对话和流程，例如：

- **黑板视图**：一个面板实时显示黑板上的关键条目和事件流（类似日志流），更新插入的新任务、信息、批判意见等，使用户直观了解系统当前思考到了哪一步。
- **Agent 思维链展示**：以对话形式或树状结构展现 Agent 的 Chain-of-Thought。例如主 Agent 的规划步骤、批判 Agent 的评论，都以气泡对话框显示，并标注角色和时间。这样方便开发者或用户追踪推理过程。
- **多 Agent 协作流程图**：使用 Tailwind CSS 的灵活布局，可以将 Agent 绘制成节点图，连线表示信息流。甚至可以引入 SVG 或 Canvas 绘图，动态呈现 Agent 之间消息传递的流程图，从而提供“所见即所得”的协作概览（如图 2 那样的流程示意）。
- **人工介入控件**：界面需提供暂停/继续按钮、任务插入面板等，让人类在必要时干预。例如在黑板视图的每个事件旁放一个“人工回复”按钮，允许用户对此提供备注或提示，系统将其作为特殊 Agent 输入处理。

Tailwind CSS 提供的实用原子样式有助于快速设计出整洁的仪表盘风格界面。前端还需通过 WebSocket 与后端保持长连接，以实时获取 Agent 产生的新内容并更新 UI，保证信息同步。

后端架构：采用 Python 的轻量级 Web 框架 **FastAPI** 构建后端服务，并结合 **WebSocket** 实现**双向通信**。FastAPI 高性能且易于编写异步代码，适合承载多个 Agent 的并发执行和 IO 操作。后端主要模块包括：

- **Agent 管理模块**：维护当前加载的所有 Agent 对象、其状态（空闲/忙碌）等。可使用依赖注入或全局注册表，方便主 Agent 动态调用特定 Agent 的服务。
- **调度与事件模块**：基于任务和事件的调度中心，监听黑板事件队列并按照优先级分发给 Agent 处理。可以利用 **Celery** 作为任务队列框架：每当有事件需要 Agent 处理时，发布 Celery 任务，由对应的 Agent worker 异步执行。Celery 擅长分布式任务调度和并行执行，结合 **Redis** 作为消息代理，实现高效的事件发布/订阅机制。
- **WebSocket 推送**：后端通过 WebSocket 将黑板的新事件推送给前端，实现实时更新 UI。例如某 Agent 完成任务写入黑板时，后端收到事件后立刻通过 WebSocket 向前端广播“某某 Agent 输出了结果 X”，前端即可渲染。
- **API 接口**：FastAPI 同时提供 RESTful API 接口，支持外部应用或开发者查询系统状态、提交新任务等。这些 API 也用于前端初始加载历史数据（如黑板日志）。

数据管理：采用 **PostgreSQL** 数据库配合 **Redis** 缓存，实现持久与高速存储兼顾。PostgreSQL 用来：

- 存储黑板长期内容（如每次任务的问题、最终方案、关键中间结果）形成知识库，可供日后检索（实现 Agent 的长期记忆)。
- 保存 Agent 评估记录、评分日志，以便分析和持续学习。
- 存储用户交互和反馈，以改进模型。

Redis 则用于：

- 缓存黑板的当前工作集，如最近的事件列表、待处理任务队列等，加速读写频繁的数据。
- 作为 Celery 的 Broker 和 Result Backend，支撑任务队列的迅捷通信。
- 实现发布/订阅：利用 Redis 的 Pub/Sub 或 Streams，将黑板事件作为消息发布，Agent 模块作为订阅方，解耦事件分发逻辑。

大模型接入：系统的大部分智能推理由大型预训练模型（LLM）提供。建议使用 **DeepSeek API** 作为大模型后端。DeepSeek API 与 OpenAI 接口兼容，支持各种中文和英文大模型调用。通过自建或托管的 DeepSeek 模型，可获得稳定的推理服务，且根据需要选择不同型号（推理型、创意型等）。我们将基于 DeepSeek API 构建 **Prompt 模板体系**：

- 针对每类 Agent 设计一套 Prompt 模板，明确其角色、职责和输出格式要求。例如批判 Agent 的 Prompt 强调“你是一个严谨的科研审稿人，负责找出方案中的问题并指出”；文献 Agent 的 Prompt 包括“阅读以下摘要，返回关键结论并列出引用”。
- 模板中预置若干 **Few-shot 示例**，提高模型输出稳定性和符合格式的程度。
- 使用 DeepSeek 的上下文缓存和 Chain-of-Thought 优化功能，确保模型在代理任务中能保持一致的思路、不轻易遗忘前文。
- 统一管理这些 Prompt 模板，方便日后根据评估反馈做更新迭代。

通过 DeepSeek API，开发者也可对模型进行**微调**或使用私有模型，有助于提升在特定科研领域的表现。部署时，可将大模型服务与主系统分离（例如容器化部署），通过 API 调用，这样可以弹性地扩展模型算力且保持前后端解耦。

部署与运行：推荐使用 Docker 容器部署各组件，实现可移植的微服务架构：

- 前端 React 应用一个容器，后端 FastAPI+Celery 一个容器，Redis 和 PostgreSQL 各一容器。通过 Docker Compose 或 Kubernetes 编排，方便伸缩。
- Celery 可以运行多个 worker 实例，实现真正的**并行多 Agent** 执行。当任务激增时，水平扩展 Celery worker 容器即可增强并发处理能力。
- 利用 Git 版本控制 Prompt 模板和 Agent 代码，搭配 CI/CD 流水线，快速迭代部署优化（尤其当有自动代码优化时，更需要测试-部署流程保障安全）。

安全方面，需要对外接口进行适当的权限控制，防止滥用。同时监控系统资源，特别是 LLM 调用次数和并发，以免因超负荷导致响应变慢或费用飙升。

5. 开发计划（3 个小团队协作）

为了高效推进研发，我们建议将团队划分为三个小组，分别侧重前端、后端和智能 Agent 开发，各组紧密协同并遵循里程碑计划交付 MVP 迭代。

团队 1：前端交互与界面实现

职责：打造直观的多 Agent 协作可视化界面，确保用户和开发者能清晰地看到 Agent 的思维过程和黑板动态，并提供必要的控制接口。具体任务包括：

- 架构设计前端单页应用（SPA），搭建 React 项目结构和状态管理方案（如 Redux 或 Context API）以管理黑板数据流。
- 实现黑板事件流 UI：类似日志控制台的组件，实时追加新事件。需设计不同事件类型不同的图标/颜色以增强可读性。
- 实现多 Agent 对话视图：将黑板上不同 Agent 的发言转换为对话气泡，支持折叠展开细节的功能，让用户按需查看推理细节。
- 绘制协作流程示意：开发自定义组件用图节点表示 Agent，用连线和箭头表示消息交互。支持高亮当前活动 Agent，突出系统运行进程。
- 构建人工干预面板：包括暂停/恢复按钮，人工在黑板发布信息的输入框等。需确保这些操作通过 WebSocket 发送指令到后端。
- 样式与响应式：使用 Tailwind CSS 快速迭代设计，保证界面在不同分辨率下布局合理。由于信息量大，要精心设计排版避免界面杂乱，同时提供过滤或搜索功能（例如只显示某个 Agent 相关的消息）。
- 前后端接口联调：通过 WebSocket 订阅后端推送的数据，并根据事件类型更新对应 UI 元素；实现 REST 接口调用以获取历史记录或提交用户插入的内容。

产出：交付一个用户界面友好的 Web 应用，使得整个多 Agent 系统的运行态势一目了然。即使非开发人员，也能通过界面理解系统提出创意的过程，并可以在关键点介入。

团队 2：后端调度系统与 Agent 管理框架

职责：实现多 Agent 协同的后端大脑，包括主 Agent 调度、事件监听和黑板队列管理，以及与前端和大模型服务的连接。主要任务：

- 搭建 FastAPI 服务结构，设计各 API 路由（如/start_task, /get_status 等）和 WebSocket 通信流程。保证后端能够将更新高效地广播给前端连接。
- 开发黑板数据模型和事件队列：定义数据库模型（PostgreSQL）用于持久存储黑板条目；配置 Redis 及 Celery，将黑板事件发布到 Redis Stream，让 Celery 异步 consume 处理。实现基本的发布/订阅代码，封装黑板读写操作的方法。
- 编写主 Agent 调度逻辑：这部分可能用一种管理类实现，包括任务分解函数、调度策略函数等。要处理事件分发、超时重试、优先级排序等复杂逻辑，并调控 Celery 任务的生产消费。
- 管理 Agent 生命周期：加载各 Agent 模块，可能采用插件机制扫描预定义目录下的 Agent 类并注册。确保每个 Agent 有唯一标识，调度器能按名称或类型找到它们。处理并发调用时的锁或队列分发，以免同一 Agent 实例被并发占用（或者采用每次任务启动一个 Agent 实例的模式）。
- 与 LLM 服务对接：封装调用 DeepSeek API 的客户端模块，供各 Agent 轻松调用大模型。如批判 Agent 可以调用 LLMClient.generate(prompt)得到输出。需要设计统一的调用接口以及错误处理、重试机制。

- 整合评估模块：虽然评估 Agent 主要由团队 3 开发，但团队 2 需要在调度器中加入钩子，将 Agent 任务完成的结果传递给评估流程。如任务结束后将日志片段发送给评估 Agent 评分。
- 搭建测试环境：模拟几个简单 Agent 在 Celery 上跑通，测试黑板事件能否正确流转，调度逻辑如预期运作。尤其要测试并发场景下资源占用和队列积压情况，并调整参数（如 Celery 并发数、预取数等）。

产出：一个稳健的后端系统，能够正确调度多个 Agent 协同工作。黑板队列机制应通过测试验证：发布任务->Agent 执行->产出结果->触发后续，整个链路流畅无误。此后端将作为核心框架支撑上层 Agent 能力的添加。

团队 3：Agent 实现与 AI 评估机制

职责：开发具体的 Agent 功能和 AI 评估、自我优化模块，相当于填充系统智能的“魂”。工作包括：

- 编写各类 Agent 逻辑：按照模块设计部分列出的分类，实现文献 Agent、批判 Agent、建模 Agent 等的功能。可先采用 **MVP** 策略，每类实现一个基本功能的子类。例如文献 Agent MVP 版本可简单调用维基百科 API，批判 Agent MVP 可对字符串搜寻特定关键词来模拟批判。随着 MVP 版本迭代，再逐步替换为真正的 LLM 调用或复杂计算。每个 Agent 需要定义：从黑板获取输入->处理->输出到黑板 的流程，并遵守统一接口。
- 构建 Prompt 模板库：针对需要 LLM 的 Agent（几乎大部分 Agent 都会用 LLM）编写 Prompt 模板并调优。可以借助 OpenAI GPT-4 或 DeepSeek 试跑 Prompt 效果，逐步完善。在代码中将这些 Prompt 定义为常量或配置，方便后续修改。支持多语言混杂情况下，也要确保 Prompt 提示清晰明确。

- 实现评估 Agent：开发一个专门的评估 Agent 或静态评估模块。它读取日志和结果，对各 Agent 按照第 3 节指标打分。评估 Agent 本身也可用 LLM 实现（Prompt 设计成“你是系统监察员，阅读以下对话和结果，给每个 Agent 评分…”）。评分结果要能够写回黑板或者存入数据库，以供主 Agent 参考。
- 设计反馈策略：基于评估结果，撰写代码实现一些自动优化行为。例如，当发现某 Agent 评分连续偏低，自动生成一条黑板消息建议更换 Agent 或改进 Prompt；或者直接调用一个函数去更新该 Agent 的 Prompt 模板（若采用强化学习或其他自动 Prompt 方法）。对于代码自优化，可以先搭建接口，例如写一个函数 optimize_agent(agent_id)，内部暂时简单打印建议，未来逐步完善成 AI 自动修改代码并热重载 Agent。
- 集成外部工具：为实现 Agent 功能，可能需要对接一些外部库或 API。团队 3 需要处理这些细节，例如 DFT Agent 需要集成量子化学计算软件的命令行调用、解析结果，热力学 Agent 可能需要 Scientific Python 库做计算。这些工具的管理（安装、调用、错误处理）都在本团队职责范围。
- 自测试与 Benchmark：设计若干测试用例，模拟典型科研问题，检查 Agent 序列是否能得到合理输出。例如输入一个简单科研问题，让系统跑完全流程，看各 Agent 是否按照逻辑出动，黑板信息是否完整。利用这些测试不断调试 Agent 实现的正确性和有效性。

产出：一套基本完善的 Agent 集合，系统已具备产生有意义创意方案的能力。评估和反馈闭环初步打通，比如可以演示某 Agent 做错事后系统调整 Prompt 的过程。团队 3 的工作使系统从一个框架变成真正智能的科研助理。

阶段划分计划

开发过程可以按 **MVP 版本**逐步推进，每个阶段都有明确目标交付：

- **MVP 1.0**：核心 Agent 框架成型。完成主 Agent 和至少 2-3 个简单子 Agent 的端到端打通（例如文献 Agent 和简单批判 Agent）。黑板和调度基本可用，实现一个简单案例演示（不要求高智能，但流程跑通）。验证多 Agent 交互机制工作，为后续扩展打下基础。
- **MVP 2.0**：引入黑板完整的事件系统和批判机制。扩展 Agent 种类（比如增加建模 Agent、验证 Agent 的雏形），完善黑板优先级队列、订阅发布、冲突处理等逻辑。此阶段重点让系统具备**批判反馈回路**：批判 Agent 能发现问题并促使主 Agent 或其他 Agent 修正。演示输出初步带评分的方案列表。开始关注性能问题，如并发处理和响应速度调优。
- **MVP 3.0**：加入 AI 评估和自我优化闭环。实现评估 Agent 或功能模块，能够在任务结束时给出各 Agent 表现评分，并对低效环节自动优化（例如动态调整 Prompt、替换 Agent 实现）。增加更多 domain-specific Agent（如高级文献 Agent 接入语义学者 API、更强的模拟 Agent 等），提升系统解决更复杂科研问题的能力。最终输出结果质量和多样性明显提高，每条方案有清晰评分排序。MVP3 应能在较复杂的真实科研课题上给出有参考价值的创意方案。

各团队在每个阶段交付各自部分，并通过集成测试验证系统功能符合预期。在 MVP3 完成后，系统基本满足功能要求，可以进入进一步强化和扩展阶段，例如增加 GUI 优化、人机交互细节、更多 Agent 插件等。

6. 其他补充设计

在上述架构和模块之外，为了打造一个健壮且易用的多 Agent 科研创意平台，还需考虑以下设计细节：

创意方案输出与评分排序：主 Agent 在整合各 Agent 成果后，需要生成**最终创意/实验方案列表**。为了增强说服力和易读性，输出机制上应：

- 确保每条方案都附有简明的**描述**、**优点分析**和**潜在问题**说明。如果批判 Agent 提出过改进意见，也可体现在方案描述里，使阅读者了解考虑过的因素。
- 输出按评分高低排序，并明确显示每项评分（如创新度 8/10、可行性 7/10 等维度评分）。评分可以采用星级、数值等形式直观呈现。
- 引用出处：对于方案涉及的关键数据或文献，提供引用标注（例如脚注或引用编号)。这可参考 Anthropic CitationAgent 的做法，在最终报告里将所有断言和来源关联。用户点击可查看详细来源信息，从而增强方案可信度。
- 数量控制：如果生成了很多方案，可能只选取前 N 个（比如 5 个）输出，以免用户不胜其烦。同时可以按不同类别归组方案（假如有类似方案或不同技术路线），提高条理性。

Agent 交互协议与 Prompt 规范：为让各 Agent 协同一致，系统需定义统一的交互协议，包括：

- **消息格式**：约定黑板上记录的信息格式，如使用 JSON 结构表示任务和结果。例如文献 Agent 输出可以是：

{ "agent": "文献 Agent", "type": "info", "content": "NMC 正极材料容量~200 mAh/g", "source": "文献[1]" }

这样方便其他 Agent 解析利用。如果采用自然语言，则也应遵循一定模板（如以[文献结果]开头表示）。

- **命名规范**：为不同 Agent 的黑板发布加标签，明确来源，避免混淆。此外统一任务命令的描述风格，如“任务 X: [类别] ...”的形式，便于解析。命令、事件的命名要简洁明确，比如使用固定前缀[任务]、[信息]、[批判]等。
- **Prompt 风格规范**：每个 Agent 的 Prompt 需遵循整体风格，不可自相矛盾。例如都采用礼貌、中立的语气陈述，不使用第一人称个人的口吻（除非设计需要）。批判 Agent 的批评也要建设性。通过规范 Prompt，保证多 Agent 对话展现一致的专业风格。
- **接口函数**：如果 Agent 开放内部方法供主 Agent 直接调用（比如某 Agent 有特定计算函数），需要定义好接口契约，包括参数类型、返回格式、错误码等。这类似传统软件模块接口，但对于 LLM Agent，大多通过黑板文本交互即可，无显式 API。然而对于涉及工具调用的 Agent，可以封装标准 API 接口（如一个 simulate(input) -> result 函数供主 Agent 使用）。

扩展性与定制机制：平台应尽可能**模块化**和**可扩展**，以便未来添加新 Agent 类型或更换底层模型：

- **插件式 Agent 注入**：可以设计一个 Agent 注册机制，允许开发者将自定义 Agent 模块放入预定义目录，系统启动时自动加载。这些 Agent 需实现基类接口，如有 process(blackboard)方法。通过读取配置文件或扫描模块，新 Agent 能够无缝接入系统。在 UI 上也可相应增加显示其活动。
- **配置驱动**：将 Agent 的种类、黑板事件类型、调度策略等做成配置文件（如 JSON 或 YAML），避免硬编码。这使得非程序员也能通过修改配置增加 Agent 或调整流程。例

如新增一个“经济性评估 Agent”只需在配置里声明其类名、订阅事件、优先级等，系统启动时自动集成。

- **模型替换**：使用抽象的 LLM 接口，以便更换模型提供方而不影响逻辑。例如封装 DeepSeek API 调用，如果未来换用 OpenAI 或本地模型，只需替换这一层实现。对于计算类 Agent，也可通过策略模式持有不同后端实现（例如本地模拟 or 云端服务），根据配置选择。
- **水平扩展**：确保系统能通过增加计算节点扩展容量。黑板和消息队列要支持分布式（Redis 本身可集群，PostgreSQL 亦可扩展读写分离）。Celery worker 可以跨多台机器运行，实现大规模 Agent 并发处理。测试需要验证当 Agent 数目和任务量增大时，系统性能线性扩展、无明显瓶颈。
- **故障隔离**：扩展性也意味健壮性。每个 Agent 失败（比如第三方 API 超时）不应拖垮系统。需设计超时和异常处理，每个 Agent 任务有 try-catch，出错时在黑板记录错误事件并由主 Agent 决定如何处理（重试、跳过等）。同时防止单个 Agent 占用过多资源，例如对 LLM 调用频率设限，避免个别 Agent 死循环调用耗尽配额。

综上，本白皮书描述的多 Agent 科研创意系统通过巧妙的架构和模块设计，实现了从问题感知、分解，到多智能体黑板协作求解，再到结果评审优化的全流程自动化。各模块各司其职又紧密配合，结合慢思考和链式推理策略，使 AI 具备深入复杂科研问题的能力。评估和自优化机制保证系统越用越智能，持续迭代进步。在合理的前后端框架支持下，该系统具有良好的可视化可控性和扩展潜力。我们相信，借助这一平台，未来能够产出更多富有新意且切实可行的科研方案，极大助力人类的创新研究工作。

附录：科研 Agent 系统开发团队实施方案

团队 2 与团队 3 的任务划分

图：黑板体系结构示意。中央黑板存储共享数据，调度控制器按照黑板状态激活相应知识源（Agents）贡献部分解。各知识源独立执行其专长算法并将部分解决方案更新到黑板上。这一模式启发了团队 2 和团队 3 的分工。

团队 2：后端调度系统与 Agent 管理框架

- **调度控制模块**：设计并实现系统的核心调度器（Controller），负责根据黑板中当前状态选择合适的 Agent 执行任务。调度器需要监测黑板上的事件/数据变化，按规则激活一个或多个 Agent 处理，并支持冲突解决机制（当多个 Agent 产生冲突或重复的解决方案时，调度器协调取舍）。
- **Agent 管理组件**：构建统一的 Agent 注册与生命周期管理框架。提供 Agent 的注册/注销接口，维护每个 Agent 的唯一标识、状态和元数据（类似 JADE 框架的 Agent 管理系统确保每个 Agent 有唯一 ID 并受控管理）。实现 Agent 的启动初始化（调用其 init()）、定期心跳或状态监控，以及在必要时暂停、恢复或终止 Agent 的能力。
- **黑板数据存储与通讯机制**：开发**黑板**（Blackboard）模块，用于在 Agents 之间共享数据和消息。黑板应支持并发读写控制，提供发布/订阅接口：Agent 管理框架可将事件写入黑板，供感兴趣的 Agent 读取；调度器也可将任务结果写回黑板。黑板记录对象需要定义统一格式（如含记录 ID、来源 Agent、时间戳、内容等），方便 Agent 按类型或主题订阅筛选。
- **系统初始化与配置**：实现框架的初始化加载逻辑，例如从配置文件读取需要启动的 Agent 列表、依赖关系和初始任务，将初始问题和数据写入黑板。确保在系统启动里程碑完成时，框架能够加载所有 Agent 并开始调度循环。

- **日志和监控**：集成日志记录模块，捕获调度决策、Agent 执行结果、黑板更新和异常错误情况，供日后分析和调优。建立基础的监控指标（如队列长度、Agent 响应时间）方便评估系统性能和为团队 3 的 AI 评估机制提供数据支撑。

关键里程碑交付：

1. *架构设计文档* – 确定黑板交互模式、调度算法和接口规范，输出设计文档和时序图。
2. *框架基础功能完工* – 实现黑板模块、调度器和 Agent 管理的基础代码，可以加载至少一个示例 Agent 运行端到端流程（输入事件经黑板触发 Agent 处理并输出结果）。
3. *多 Agent 协调与冲突处理* – 增强调度器支持并行多 Agent 执行和冲突解决策略，黑板模块支持复杂数据结构和订阅过滤。交付包括关键用例的集成测试报告。
4. *稳定版发布* – 完成日志监控集成和错误处理机制，代码经过充分单元和集成测试，准备交付团队 3 全面对接。

团队 3：Agent 实现与 AI 评估机制

- **Agent 标准接口与基础类开发**：依据团队 2 提供的框架，设计 Agent 的抽象基类（或接口协议）并实现其通用功能。定义如 init()（初始化配置）、process(event)（处理调度器分配的事件）和 on_event(event)（响应异步事件回调）等生命周期方法，确保所有自定义 Agent 遵循统一签名和行为。参考 JADE 等多 Agent 框架，Agent 启动时会调用初始化方法设定其行为和状态。基础类还可提供日志记录、错误捕获等通用功能，减少重复编码。
- **具体 Agent 模块开发**：根据系统需求实现多个具体的 Agent。每个 Agent 封装特定的知识或算法，在收到黑板上的相关事件/数据时执行。例如实现**数据采集 Agent**（从黑板原始数据推导高层信息）、**推理 Agent**（应用 AI 模型产出决策建议）、**协调 Agent**（整合其他 Agent 的部分结果形成最终方案）等。确保每个 Agent 实现其

process/on_event 逻辑，将处理结果封装为黑板记录对象输出。团队 3 需与团队 2 配合，对接黑板读写接口和事件触发机制，确保 Agent 能正确获取输入并发布输出。

- **AI 评估机制模块**：研发**评估器**组件，用于自动评估 Agent 系统的行为和性能。这可能包括一个专门的**评估 Agent**，持续监听黑板上的最终结果或中间过程，根据预先定义的指标进行评分反馈。例如，对于给定输入问题，评估 Agent 比较多个 Agent 产生的方案优劣，或者验证最终解决方案是否满足成功条件（可通过黑板记录标记成功/失败）。评估机制也可以收集每轮迭代所用时间、协作次数等指标，用于后续优化。关键交付是**评估报告生成工具**，在模拟测试中输出各 Agent 表现和系统整体指标。
- **测试用例与模拟环境**：团队 3 负责编写针对各 Agent 的单元测试和黑板交互的集成测试。利用框架提供的钩子，可以构造模拟黑板事件序列，测试 Agent 在各种情形下的反应和错误处理。例如提供 mock 的事件字典作为输入，断言 Agent 输出的黑板记录符合预期格式与内容。通过这一机制验证 Agent 逻辑正确性，也为 CI 流程中的“黑板模拟测试”提供测试脚本。
- **关键里程碑交付**：
    1. *Agent 接口定义完成* – 提交 Agent 抽象类/接口的设计说明和示例，实现基础类代码并通过团队 2 审核对接。
    2. *核心 Agent 原型实现* – 至少实现一两个核心 Agent（含基础的 AI 算法）和一个简单评估 Agent，并在框架上跑通一个端到端案例。
    3. *完整 Agent 集成* – 实现所有计划中的 Agents 功能代码，完成评估机制模块。与团队 2 共同调试所有 Agent 在黑板上的协作，交付完整的多 Agent 系统演示。

4. *性能与评估报告* – 优化 Agent 逻辑和评估指标，完成多次运行的性能评测。提交 **AI 评估报告**，分析系统的正确性和效率，并提出可能的改进建议。

代码库管理方案（GitHub 单仓库）

为了便于模块集成和团队协作，采用单一 Git 仓库（monorepo）托管整个 Agent 系统的代码。所有团队 2 和团队 3 的代码放在同一仓库下，通过清晰的目录划分实现模块隔离：

agent-system/

├── backend/ # 后端调度框架（调度器、黑板、Agent 管理等）

├── agents/ # 各 Agent 实现代码（每个 Agent 一个子目录）

├── common/ # 公共模块（共享的工具、Agent 基类、数据模型等）

├── tests/ # 测试用例（单元测试、集成测试按模块归类）

├── docker/ # Docker 构建相关文件（Dockerfile、配置等）

└── docs/ # 文档（架构设计、接口规范、开发指南等）

模块组织原则：backend 和 agents 目录对应团队 2 和团队 3 的主要工作领域，各自拥有清晰边界。例如，Agent 基类定义可置于 common/agent_base.py，由团队 2 维护，但具体 Agent 实现在 agents/子目录由团队 3 维护，实现对基类的继承扩展。采用单仓库集中管理有诸多优点：它提供了**高可见性和代码共享**，所有团队成员都能方便地查阅和复用彼此的代码，避免不同仓库知识隔离。同时，单一仓库便于**标准化工具链**的配置，对主干分支实施统一的代码策略和审查流程。

协作方式：各团队在各自目录下并行开发，通过分支策略防止冲突。例如，团队 2 可以在 backend-dev 分支开发调度框架，团队 3 在 agents-dev 分支开发 Agent，实现阶段性完成后合并回主分支。由于 monorepo 让所有模块处于同一版本控制视图，实现跨模块的原子性更改成为可能——一次 Pull Request 即可同步修改多个子模块以完成接口对接或功能迭代，这提升了多团队协作开发的效率。仓库应设置分支保护策略，如只有通过必要检查和审查后才

能合并到 main。通过在单库内维护一致的依赖管理和构建脚本，也确保了不同组件之间的兼容性和统一的部署流程。

基于 GitHub Actions 的 CI/CD 方案

我们将在仓库中配置 **GitHub Actions** 工作流，实现持续集成（CI）和持续部署（CD）的自动化。

工作流概览：每当有代码提交（Push）或拉取请求（PR）创建/更新时，触发 CI 流水线。CI 流程将**编译/构建代码**（如需要）、执行**测试**并进行**代码质量检查**，确保新改动能集成到主干而不破坏现有功能。随后，CD 流程在将代码合并到主分支或发布标签时触发，自动构建 Docker 镜像并部署更新的服务版本到目标环境。具体包含以下阶段：

1. **安装依赖与环境设置**：使用 GitHub 提供的虚拟机，加载所需的语言运行时（如 Python 或 Node）、安装项目依赖包等。针对 monorepo 多模块的情况，可并行设置多个作业。例如，分别设置 backend 和 agents 的测试作业，或者不同语言环境的作业矩阵。
2. **静态代码检查**：运行代码风格和质量检查工具，如 Lint（Pylint/Flake8、ESLint 等）和类型检查（mypy/TypeScript Compiler）。这能自动发现语法错误、不符合风格指南的代码，并在 PR 中反馈问题。静态检查作业快速失败可以阻止低质代码进入主库。
3. **单元测试**：执行各模块的单元测试套件。使用 pytest 或 unittest 运行 tests/目录下测试，用覆盖率工具收集覆盖率数据。此阶段验证函数级别的正确性，所有测试须通过且覆盖率达到规定阈值（例如 80%以上）才视为成功。
4. **集成测试**：在通过单元测试后，运行跨模块的集成测试场景，确保调度器与 Agents 协同工作正常。例如，启动一个模拟的黑板环境，预置一系列事件，由调度器触发多个 Agent 处理，最终验证黑板上的输出是否符合期望（**黑板模拟测试**）。可通过 GitHub Actions 的服务容器功能，启动临时的消息队列或数据库容器，模拟系统运行环境，进行更真实的端到端测试。集成测试脚本会调用框架公开的接口（如向黑板写入事件）并观察各 Agent 响应，检查系统整体行为。

5. **安全与依赖检查**（可选）：集成 Dependabot 或其他安全扫描动作，定期检查依赖库的漏洞，并在 CI 中运行 SAST（静态应用安全测试）工具，及时发现安全隐患。
6. **Docker 镜像构建**：当代码通过以上检查且合并主分支时，CI 触发 CD 流程构建 Docker 镜像。借助官方的 Docker Buildx Action，使用仓库中的 Dockerfile 将应用打包为容器镜像（包含后台框架和所有 Agents）。镜像打包完成后，通过登录凭据自动推送到镜像仓库（如 Docker Hub 或 GitHub Packages）。
7. **部署与发布**：最后，触发部署步骤（可在特定分支或打标签时执行）。例如，在推送镜像后，通过 SSH 或 Kubernetes Action 将新镜像部署到测试服务器；或使用 Continuous Deployment 工具实现自动更新。如果是研究性质项目，也可在 CI 完成后人工确认再部署。发布流程应包括运行数据库迁移脚本（如有）、重启服务、以及在 GitHub Releases 页面记录新的版本说明等。

CI/CD 工作流实现：在仓库的.github/workflows/目录下添加配置文件（YAML）。例如，一个 CI 工作流定义名为“CI”，包含上述 lint、测试、构建阶段，在 PR 提交和 main 分支 push 时运行；另一个 CD 工作流在创建版本标签时触发，进行部署。我们会充分利用 GitHub Actions 的并行执行和缓存特性，加快流水线速度。如依赖安装阶段启用依赖缓存，Docker 构建阶段缓存中间层等。整个流水线设计遵循**快速反馈**原则：静态检查和单元测试先于较慢的集成测试和构建，以尽早发现问题。一旦某步骤失败，后续步骤将跳过并通知开发人员修复。

通过上述 CI/CD 方案，代码变更将被严格验证并快速部署。一方面，每次合并都会运行完整测试，确保各团队的改动兼容整个系统；另一方面，自动构建部署减少人工错误，使研发团队能够专注于开发本身。

Agent 标准接口定义方式

为了确保不同 Agent 模块可被统一管理和调度，我们需制定标准的 Agent 接口规范，包括类结构、方法约定、数据协议和错误处理方案。具体设计如下：

- **抽象基类/接口协议**：定义一个 AgentBase 抽象类（或等价的接口）作为所有 Agent 的父类，团队 3 据此实现具体 Agent。该基类声明 Agent 的核心方法：例如 init(config)用于初始化，process(event)用于处理调度器分配的任务事件，on_event(event)用于响应异步事件通知。这些方法在基类中可以提供默认行为或抽象定义。调度框架（团队 2）只与 AgentBase 类型交互，从而对具体 Agent 实现细节解耦。类似 JADE 等通用 Agent 框架，每个 Agent 启动时框架会调用其初始化方法来设置自身状态，然后在运行时通过框架回调相应的方法处理消息。我们的 AgentBase 也可包含诸如 shutdown()（关闭善后）等方法，统一 Agent 生命周期管理。接口协议需要在文档中清晰注明各方法用途、调用时机和期望行为，方便团队 3 遵循实现。
- **生命周期管理方法**：
    - init(config)：**初始化。**在 Agent 创建或启动时由框架调用。传入配置参数（如从配置文件读取的该 Agent 专属配置），Agent 应在此完成资源准备、模型加载、初始状态设定等工作。init 应返回初始化是否成功的状态，供框架判定 Agent 是否可用。
    - process(event)：**主处理函数。**由调度器在适当时机调用，传入一个事件对象（详见下文输入协议）。Agent 执行其核心逻辑，基于事件内容进行计算、决策，然后返回结果（或直接将结果写入黑板）。process 通常是同步阻塞的，即拿到一个事件就执行到底。调度器可在调用前后记录日志、计算超时等。对于需要长时

间运行的任务，考虑将 process 设计为可重入或可暂停，以便框架进行超时处理或负载均衡。

  - on_event(event)：**事件回调。**当黑板上有新的事件发布且该 Agent 对其感兴趣时，由框架异步调用此方法。不同于 process 由调度器主动分配任务，on_event 更像一种**订阅通知**：Agent 事先向黑板或调度器登记自己关注的事件类型，当相关事件出现时框架调用 Agent 的 on_event 进行处理。这一机制适合于事件驱动的 Agent，例如监控型 Agent 对错误事件的处理。on_event 实现可轻量处理或将事件转化为内部任务，再调用 process。框架应确保对同一 Agent 的并发调用有序化（例如同一时刻不同时调用两个 process/on_event）。

- **标准输入输出协议**：统一 Agent 交互的数据格式，确保团队 2 和团队 3 有共同约定。**输入**采用**事件字典**（或对象），包含必要字段：如{"type": "<事件类型>", "data": {...}, "source": "<触发来源>", "id": "<事件 ID>"}等。type 用于标识事件类别（方便路由给订阅此类别的 Agent），data 承载具体内容（例如传感器读数、用户请求等），source 指明事件来自哪个 Agent 或系统组件，id 用于跟踪。同样，可以扩展字段如 timestamp 记录时间。中阐述了黑板流程，从初始问题作为事件写入黑板开始，后续 Agent 不断产生新的部分解作为事件更新黑板——我们定义的事件字典将作为此流程中的标准载体。**输出**则采用**黑板记录对象**，它可以与事件格式类似，或为更丰富的类。例如定义一个 BlackboardRecord 类，包含 record_id, producer, content, confidence, status 等属性。Agent 处理完事件后，应将结果封装为 BlackboardRecord，由框架写入黑板共享。例如一个 Agent 将计算结果填入 content，标记状态为 SOLVED 或附加一个置信度。这样所有 Agent 输出在黑板上结构一致，便于评估 Agent 行为和让其他 Agent 消费结果。输入事件与输出记录都可序列化为 JSON 用于日志和持久化。需要注意输入

输出协议必须**前后一致**：Agent 对收到的事件字段要严格解析，对输出记录也只填规定字段，未知字段忽略，以确保系统松耦合和兼容升级。

- **错误处理规范**：为提升系统健壮性，规定 Agent 在发生异常时的处理策略和框架协同方式：
    - **异常捕获与日志**：每个 Agent 在 process 和 on_event 实现中应该使用 try-except 捕获可能出现的异常。对预料中的错误（如输入数据格式不符）可以抛出自定义的 AgentError 异常，并由框架捕获；对未预料的异常，Agent 也应在 except 中记录错误日志（包括异常类型、消息、堆栈），然后可以选择向上抛出让框架处理。框架应集中捕获 Agent 调用中的异常，将关键信息写入系统日志以及黑板的错误区域。**日志**需包含 Agent 名称、事件 ID、异常原因等，便于日后调试分析。通过严格的异常日志记录，可以定位问题根源并评估 Agent 可靠性。
    - **退出码/状态码**：若 Agent 以独立进程形式运行（可选的实现），则定义统一的进程退出码协议。例如约定 0 表示正常结束，1 表示初始化失败，2 表示处理过程中发生未恢复的异常等等。这在多进程部署时有用，框架据此判断 Agent 是否需要重启。提到脚本可返回特定退出码用于表示不同错误，我们可以在 Agent 主循环退出时 return 相应码供调度器读取。不过在单进程实现中，退出码可转换为在返回值或异常中带一个错误代码字段，同样达成标识效果。
    - **重试标记与机制**：对于可临时性错误（例如网络波动导致某次请求失败），Agent 应能将事件标记为可重试而非完全失败。实现方式可以是在抛出的异常中附带一个属性，如 recoverable=True，框架捕获后判断如果是可重试错误，则将该事件重新放回黑板或任务队列，增加一个重试计数。黑板中的事件字典也可设计一

个 retry_count 字段，每次重试加 1，超过某阈值则放弃并记录为最终失败，防止无限循环。Agent 本身也可在 catch 到可恢复异常时，先记录错误然后返回一个明确的失败结果对象，让框架识别后安排稍后重试。重试需配合**退避策略**（例如延迟一段时间再试）以避免资源浪费。

- **统一错误协议**：为了让所有 Agent 的错误处理行为一致，我们制定统一错误类型和代码。例如定义 AgentErrorCode 枚举：INVALID_INPUT=100, RUNTIME_EXCEPTION=200, RETRYABLE_ERROR=300 等。Agent 遇到对应情况就使用相应代码，框架据此采取动作（如 100 类错误直接标记任务失败不重试，300 类错误允许重试）。同时黑板可设计专门的错误记录区，Agent 或框架将错误事件发布到黑板，供监控 Agent 或评估机制使用。这样系统对错误有**可观察性**：评估 Agent 可以订阅错误事件，统计各 Agent 错误率，判断系统可靠性。

按照以上接口定义，每个 Agent 开发都需实现标准的方法并遵守数据/错误协议。这保证了团队 2 的框架能够用通用方式管理 Agent 而无需了解其内部细节。当一个 Agent 完成一次 process 后产出结果，框架将结果记录投放到黑板，触发后续 Agent 工作；若发生异常，框架按协议处理不中断整体流程。整体而言，这种接口设计提供了**一致性和可扩展性**：后续新增 Agent 只要继承基类实现必要方法，即可无缝插入系统运行，且其输入输出自动被系统识别处理。

团队协作机制与代码合并审查流程

为了保障多人协作的代码质量和开发效率，我们制定了一套完善的协作与代码评审流程，包括 Pull Request 模板、代码风格规范、测试覆盖要求和评审制度。

- **Pull Request 模板**：在仓库中创建 pull_request_template.md，提供标准化的 PR 描述格式。模板将引导贡献者填写以下内容：
  - **变更简介**：简要说明本 PR 做了哪些修改 (修复了什么问题或新增了哪些功能)，对应的任务或 issue 编号。要求语言精炼明确，方便 reviewer 快速了解背景。
  - **实现说明**：描述主要的实现方案和关键技术细节。比如“引入了新的 Agent X，实现策略 Y”；若是框架改动则说明修改了哪些模块的交互。这样 reviewer 可以对照设计意图检查代码实现。
  - **测试情况**：列出已运行的测试用例及结果。如“本地已运行全部单元测试通过，新加了针对场景 Z 的集成测试”，“通过了 CI 中的静态检查和覆盖率达到 85%”。若涉及 UI 或日志输出变化，可以附上截图或日志片段。
  - **影响范围**：说明此改动对系统其他部分可能的影响，例如是否需要团队 2/3 的其他模块配合修改、是否影响向后兼容等。并@相关人员关注。
  - **检查列表**：一个复选清单，确保提交者在创建 PR 前完成了基本检查。例如：
    - 代码通过本地单元测试和集成测试
    - 代码符合风格规范（已通过 Lint）
    - 增加或更新了必要的文档
    - 针对新增功能编写了相应测试

      这个 Checklist 有助于培养开发自检意识，也方便评审人快速看到哪些项已完成。

模板的作用是让 PR 描述**充分且一致**，避免重要信息遗漏。当 PR 创建时，模板内容会自动出现在描述框，填完后 reviewer 能一目了然地了解代码变更及验证情况，提高审查效率。

- **代码风格与质量规范**：团队统一遵循约定的编码风格，例如 Python 代码遵循 PEP8 规范，JavaScript 采用 Airbnb 风格指南等，并配置相应的 Lint 工具强制执行。在 CI 中集成这些静态检查，凡是不符合风格的代码一律不能通过。同时，我们建议使用自动格式化工具（如 Black、Prettier）和 pre-commit 钩子，在开发者提交代码前自动格式化和检测常见问题，减少无意义的人为差异。代码风格上还包括：
    - **命名规范**：采用有意义的英文命名，统一大小写风格（如类名 PascalCase、函数变量名 camelCase），避免缩写。尤其黑板事件、Agent 方法等命名需反映其用途。
    - **结构规范**：每个模块文件尽量内聚单一职责，长度适中。对框架层代码要求多添加注释说明关键逻辑，Agent 实现也应在复杂算法处附注释或链接参考资料。
    - **文档与注释**：公共接口和类必须编写 docstring 或注释说明用法。重大设计决策在 docs/中记录，代码中引用该文档章节以防止理解偏差。
    - **最佳实践**：鼓励遵循 SOLID、DRY 等原则，凡是重复代码考虑抽取公共函数。对可能的空值、异常情况要有处理（正如 Cursor 提示的，AI 生成代码时**常忽略错误处理**，需着重补全）。

此外，我们制定了**代码审查指南**，帮助评审人从风格和质量角度检查，比如：“函数是否过长需拆分”、“是否存在隐含的 bug 风险”、“边界条件是否考虑”。总之，代码风格规范的目标是让整个仓库的代码风格一致、易读、易维护，降低多人协作的摩擦。

- **单元测试覆盖率标准**：我们要求新增代码必须附带相应的测试，并将**整体测试覆盖率**维持在一定水平（例如不低于 80%）。80%覆盖率被视为行业常见的底线，可提供基本保障。在 CI 流程中，我们会通过工具统计覆盖率，如果低于阈值则标记检查未通过，促使开发者补充测试。对于无法覆盖到的代码段（如很难模拟的异常场景），需要在 PR 说明中解释。我们也关注**测试质量**而非仅数量，鼓励采用语义清晰的测试用例名称和充分的断言，避免写无效的“假测试”。代码评审时，reviewer 会检查测试是否覆盖关键路径和边界情况。团队还可定期召开测试回顾会，分析哪些 Bug 未被测试捕获，从而改进测试用例。通过保持高测试覆盖率，我们建立对每次改动的不引入回归的信心。
- **代码审查与合并流程**：实行严格的 Pull Request 审查制度：每个 PR 至少需要 1-2 名合适的团队成员批准才能合并主分支。我们在 GitHub 上配置 Code Owners 文件，将 backend/目录下代码 owner 设置为团队 2 技术负责人，agents/目录代码 owner 为团队 3 负责人等。如此，当相关模块有改动时，会自动请求对应团队的人审核。审查人主要检查：
    - 功能实现是否符合需求及设计意图，是否有遗漏的角落情形。
    - 代码是否遵守上述风格规范，有无可优化之处。
    - 测试是否充分，CI 是否绿灯通过（我们要求 **PR 必须通过所有 CI 检查**才能审批）。
    - 对于跨团队交叉的改动，评审人要特别留意对另一团队代码的影响，必要时邀请该团队共同评审。

  评审过程中采用 GitHub Review 的评论机制，审查人可在代码行上直接评论问题或建议，并给出 approve/request changes 状态。团队提倡良性沟通：评语具体客观，指

出问题原因并建议改进方案；开发者对收到的修改请求应积极响应，修正后在 PR 中回复。通过评论和讨论，达成一致后才进行最后的 approve。

- **合并与发布**：一旦 PR 获得必要的批准且所有讨论 resolved，维护人员（或 CI 机器人）将把 PR 合并至主干。我们采用 **Squash Merge** 方式合并，以保持主干提交历史整洁，每个 PR 对应一条合并记录并自动关联 issue 关闭。合并时需要确保更新 CHANGELOG.md 或发布说明，以记录变更。主干上的更新若需要立即部署，通过 CI 流水线触发自动部署，否则按既定发布节奏（例如每周一次）由维护者手动触发部署。对于影响较大的改动，考虑采用 feature flag 策略，在主干合并但不开启，以便进一步测试时按需激活。
- **团队沟通与协作**：除代码层面的机制，我们鼓励团队间高频沟通。每周举行一次团队 2 和团队 3 的同步会议，交流最近的开发进展和遇到的问题，确保双方对接口契约和设计变更有一致理解。使用 Issue 跟踪功能来管理任务，每个新特性或 Bug 在 Issue 中描述，由相应负责人认领并关联 PR，这样所有人都能追踪进度。对于需要跨团队合作的功能，提前共同制定方案，在设计阶段就审阅彼此意见，避免实现阶段返工。在协作平台（如 Slack 或飞书）建一个共享频道，方便随时讨论技术细节。良好的协作机制配合严格的代码审核流程，可以减少误解和集成问题，提升团队整体效率。