不要氛围式编程，采用骨架式编程：面向领域专家构建低成本智能体工作流的交互式无代码笔记本*

Sriram Gopalakrishnan摩根大通 人工智能研究
sriram.gopalakrishnan@jpmchase.com

2026年3月20日

arXiv:2603.18122v1 [cs.AI] 18 Mar 2026

摘要

Skele-Code 是一种基于自然语言和图的界面，用于借助人工智能智能体构建工作流，尤其面向较少接触技术或非技术用户。它支持渐进式、交互式的笔记本式开发，并将每个步骤转换为代码，同时要求一组必要的函数和行为，以支持工作流的渐进式构建。人工智能智能体仅用于代码生成和错误恢复，而不用于编排或任务执行。这种由智能体支持但以代码为先的工作流方法，加上 Skele-Code 中使用的上下文工程，相较于采用多智能体系统执行工作流的方法，有助于降低令牌成本。Skele-Code 可生成模块化、易于扩展和共享的工作流。生成的工作流还可被智能体用作技能，或作为其他工作流中的步骤。

1 引言

我们提出 Skele-Code，这是一种基于自然语言和图的界面，可帮助领域专家（中小企业）和非技术用户无需编写代码即可构建由智能体支持的工作流，同时避免完全由智能代理系统带来的高昂令牌成本和不可预测的行为。这个名称的灵感来自编程练习：学生会拿到作为指导大纲的骨架代码。类似地，用户通过直观的图形界面，以自然语言（NL）描述任务，从而提供工作流结构；Skele-Code 则负责代码生成、上下文管理、消息传递和确定性编排。该界面还支持逐步构建和测试，类似于交互式代码笔记本所提供的功能。

当前的 AI 辅助工作流开发方法存在显著局限。氛围式编程 [15], 是指用户通过与编码代理聊天、反复编程，这使得在较长或多次会话中审查指令变得十分费力，进而可能降低生产力 [2]。编码代理也可能难以在长对话中保持上下文和一致性 [15]，原因是其中包含无关或过时的信息（这种现象有时称为上下文腐化 [6]），或者出现中间信息丢失 [11]，即使长上下文中间部分的内容相关，也可能无法得到有效关注。

另一种方法是完全智能体化工作流 [4,18]，通过多个大语言模型智能体动态解释并执行任务，虽然具有较强的通用性，但会增加行为不可预测的风险 [3]

*© 2026 摩根大通。保留所有权利

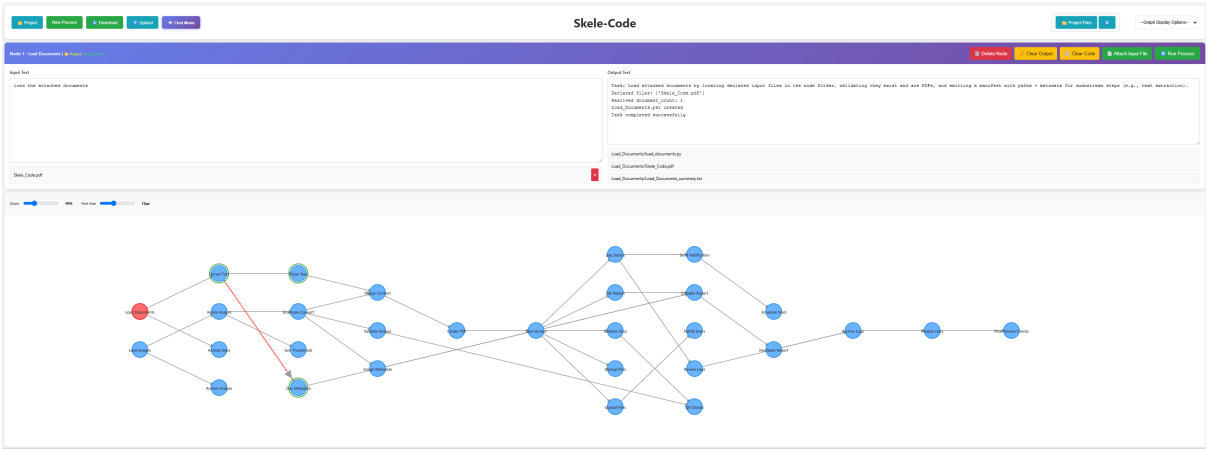

图 1：Skele-Code 界面原型，展示了工作流的图形显示区（下方），以及以自然语言显示所选节点相关信息的输入/输出单元格（上方）。

与代码优先的方法相比，这种方法会产生不必要的令牌成本；代码优先的方法首先运行代码，仅在出现错误时由大语言模型介入。最后，低代码拖放选项可能需要许多中小企业所不具备的技术知识，而了解有哪些模块可用以及它们能做什么也需要经历学习曲线，这可能成为阻碍采用的因素。

本工作的主要目标和特性如下：

- **无代码开发的信息型用户界面：** 我们将有向图表示法与类似代码笔记本（如 Jupyter [9]）的输入输出单元格显示方式相结合，用自然语言而非代码描述每个步骤。此外，我们还提供聊天界面，以协助审阅和调试，并可生成初始工作流草稿，供用户在画布上协同编辑。

- **上下文管理：** 我们利用工作流的结构（以 JSON 表示）以编程方式提取相关上下文，并缓解中间信息丢失 [11] 或上下文腐蚀 [6]等现象。

- **计算效率：** Skele-Code 的上下文管理也有助于降低令牌成本，方法是限制编写节点代码时读取的代码和用户输入量。这样便可使用规模更小的编码模型，或有效上下文长度较短的模型 [7]。此外，在工作流执行期间，我们只在修复错误或处理明确需要大语言模型解读的步骤（如文档摘要）时才进行解读，从而进一步控制令牌成本。这与完全采用智能体工作流范式的做法形成对比；在后者中，每项任务都由智能体完成。

生态界面设计（EID [17, 19]）的原则之一，是将约束和关系呈现出来，以支持用户理解。对于工作流的智能体编码而言，有向图表示是外化结构的自然选择；与线性文本（基于聊天的氛围编程）相比，它还能降低认知负荷。图表示有助于支持基于感知的推理。

最相关的界面是 Langflow [12]。这仍然要求用户对模块之间交换的数据结构进行推理，并掌握有关模块的知识。我们的系统还提供交互式笔记本风格的体验，以及用于构建工作流的协同聊天界面。

本文仅聚焦于原型中实现的 Skele-Code 设计与方法。我们首先讨论界面细节和后端，随后与相关工作进行比较，最后介绍其优势及未来发展方向。

2 Skele-Code 流程与用户界面

2.1 图形与自然语言界面

用户通过交互式图形和自然语言界面（1）定义工作流。工作流的构建方式是：向绘图画布中添加节点，使用有向边连接这些节点以建立依赖关系，并使用自然语言描述指定每个节点的任务。这种方法让用户能够专注于需要完成的任务，而不是如何用代码实现。

该界面由五个主要组件构成：

- **图形显示区：**中央画布以有向图的形式，交互式地呈现工作流。用户可以在画布上单击鼠标左键来创建新节点；按住 Ctrl 键并依次单击两个节点，即可绘制边并建立依赖关系；选中元素后按 Delete 键即可将其删除。与冗长的文本对话和摘要相比，这种可视化表示能让工作流结构更易于理解和操作 [16]。

- **输入/输出单元格：**选中节点后，界面会显示可编辑的输入和输出单元格，其形式类似于 Jupyter 等交互式代码笔记本。用户可以用自然语言输入任务描述，并附加输入文件。为节点生成并执行代码后，输出单元格会以自然语言显示结果。用户还可以清除生成的代码并重新运行节点，以再次触发代码生成。

- **JSON 属性编辑：**希望进行更精细控制的用户，可以按住 Alt 键并单击节点，打开属性编辑器以修改节点名称和描述。技术水平较高的用户则可以通过文本视图直接编辑底层的 JSON 表示。按住 Alt 键并单击空白画布区域，即可打开项目级 JSON。

- **执行控制：** 右键单击节点可切换其执行状态，使节点周围出现或消失绿色圆圈。右键单击画布上的空白区域可同时切换所有节点。当用户通过“run-process”启动执行时，节点会按照拓扑顺序执行，从而确保依赖关系得到满足，并且前置节点的输出在后继节点运行前已经可用。

- **聊天模式：** 聊天界面提供了与智能体进行对话式交互的入口，使其能够检查和修改整个项目。该模式支持调试、回答有关工作流的问题，以及根据高层次描述生成初始工作流草案。用户可以在聊天的同时继续使用交互式用户界面。用户界面见附录 A.1

Skele-Code 可适应不同程度的技术专长。非技术用户可以完全通过图形界面和自然语言描述进行操作。中级用户可以直接编辑 JSON 工作流定义。高级用户可以修改生成的 Python 脚本；这些脚本与工作流节点保持清晰的一一对应关系，并采用相同的名称。

当前实现支持有向无环图（DAG）。虽然后端架构可以支持循环，但如何设计界面，让非技术用户能够直观地使用循环结构，仍在积极开发中。

2.2 后端与上下文工程

用户通过用户界面提供工作流结构，该结构会转换为 JSON 并发送到 Skele-Code 后端。后端以 Flask 服务器的形式实现，负责将 JSON 传递给图执行模块。该模块会识别哪些节点已标记为运行，并提取每个节点所需的相关上下文。

我们将首先讨论工作流的 JSON 结构，展示 Skele-Code 如何在用户界面与后端之间以及节点之间交换信息。然后，我们将讨论后端处理此 JSON 的具体细节。

2.2.1 GraphJSON 表示

工作流以 JSON 对象的形式表示，其中包含项目级元数据和一组节点定义。这种结构有助于用户界面与后端之间，以及执行过程中各节点之间的信息交换。JSON 模式由两个层级组成：适用于整个工作流的项目级属性，以及用于定义各项任务的节点级属性。下面是一个示意性示例，其中的字段包含描述符

清单 1：节点 JSON

```
{
    "项目名称": "用户定义的名称",
    "项目描述": "对整体工作流的可选描述，可能有所帮助
    ↪编码代理",
    ...,
    "节点_id": {
            "名称": "用于显示的节点名称",
            "描述": "可选。可用于用户注释，或节点专属的
    ↪由用户提供、且不属于任务描述的指导信息。",
            "前置节点": [
                    "前置节点_id 1",    "前置节点_id 2"
            ],
            "run": true
                    ,
            "输入": {
                "文本": "用户对任务的自然语言描述",
                "文件":  ["input1.xlsx", "input2.doc"]
            },
            "输出":{
                "text":"智能体生成的代码的输出。这些是
    ↪用于总结进度、错误和输出的自然语言陈述"
                "文件":  ["输出_文件_1.png"]
            }
    },
    ...
}
```

该 JSON 对象会与任何用户输入文件一同发送至 Skele-Code 后端，用于为节点编写代码，并执行工作流中设置为运行的节点。在运行工作流中的某个节点时，我们首先检查是否已经编写了完成其任务所需的代码。如果没有，Skele-Code 会以编程方式收集完成该任务所需的上下文，并将其连同特定提示（附录 C.1）提供给编码代理，以生成代码。附录 B 还提供了一个工作流 JSON 示例。

2.2.2 代码生成过程

代码生成过程分为两个阶段：

1. **上下文检索与编码：**智能体会收到一个包含目标节点信息（包括任务描述）、前驱节点和后继节点的 JSON 定义，以及项目级描述的 JSON 对象。系统会指示智能体阅读相邻节点的代码（如果存在），以确定如何使用这些节点的输出，以及需要为后继节点生成哪些数据结构。这些信息将指导智能体在项目文件夹中该节点专属的嵌套文件夹内生成节点代码。

2. **执行与输出：**编码代理生成代码，并将其返回给 Skele-Code 的主流程。随后，主流程执行脚本；该脚本会以自然语言输出状态消息，并可配置不同级别的技术细节。消息可能类似于“已计算公司 X 的市盈率”或“未找到 file.ext”。文本输出会返回到用户界面，并显示在输出单元格中。作为任务一部分生成的文件（例如图像或电子表格）可通过用户界面中的下拉菜单访问。生成的文件也会列在输出单元格中，并可从服务器下载A.1。

2.2.3 工作流的编排与执行

工作流执行的编排与执行是以编程方式完成的，而非由智能体驱动：每个节点的代码都会以编程方式执行，节点之间输出到输入的传递则遵循图拓扑。因此，如果工作流不需要任何依赖大语言模型的步骤，且生成的代码能够无错误地运行，那么整个工作流就无需调用大语言模型即可运行。如前所述，节点的结果会显示在用户界面中。

对于技术用户，可以通过一个 Python 脚本从命令行界面（CLI）执行生成的工作流，我们称该脚本为“CLI流程运行器”。此脚本接收项目文件夹，并调用 Flask 服务器原本会调用的流程执行代码。此功能使工作流能够作为其他工作流中的子例程，并融入其他系统。如何让非技术用户更方便地使用嵌套工作流，仍是一个正在积极开发的领域。

2.3 节点代码结构

编码代理生成代码以执行节点任务时，会遵循特定的结构。每个节点的代码都必须包含一组函数，也可以根据需要添加更多函数。每个节点都包含 preprocessing()、compute() 和 save() 函数。预处理方法会检查节点是否已准备好运行。计算函数负责执行任务，并返回一个数据结构，其中包含供后继节点使用的变量，以及用户要求的任何输出。最后，保存函数会存储节点的输出状态，以便进行增量执行和调试。编排器可以在不调用保存函数的模式下运行，以加快执行速度并避免占用磁盘空间。

最后，在 Skele-Code 实现中，所有节点目前都是同一类型；这意味着不存在特殊的分支逻辑节点。例如，二元决策的分支逻辑可以通过一个任务来实现，该任务输出评估结果“是”或“否”；后继节点将决定自身是否运行。

基于这份关于我们工作的技术概述，我们现在将把它与其他工作流编码方法进行比较。

2.3.1 通过马尔可夫毯进行上下文工程

JSON 中的结构信息带来了一项关键优化：我们不再向编码代理提供完整的工作流历史，而只提供紧邻节点的上下文——即这些节点的 JSON 定义和代码（如有）。这种方法受到图模型中局部性原则的启发：节点的行为通常可以通过其直接邻域（马尔可夫毯）来刻画。只提供前驱节点和后继节点的 JSON 与代码，既能缩短上下文长度，又能保留与代码生成最相关的信息。

这种上下文工程有两项优势。首先，它通过限制长期开发会话中可能不一致或过时信息的累积，帮助减轻上下文腐化的影响。其次，它可以降低每个编码步骤的令牌成本，从而能够使用更小的编码模型，或有效上下文长度较短的模型 [7]来处理较长的工作流。代价是，用户必须正确连接工作流中的各个步骤，并理解信息流——而领域专家通常具备这方面的知识。

2.3.2 编码代理的灵活性

编码代理的选择是 Skele-Code 中一个可互换的组件。其主要贡献在于代理式编码的交互范式、生成代码结构以及执行过程。我们为原型实现了一个简单的编码代理，而较新的编码代理很可能表现得更好。唯一需要进行的调整，是更新用于规定每个节点代码应采用何种结构的指令。完整的指令提示见附录。

在此实现中，Python 是主要生成语言，但 Skele-Code 并不依赖任何 Python 特有功能。通过更新编码代理指令中的相应部分，也可以使用其他编程语言。我们测试了 Skele-Code，其中的节点会生成 HTML 和 JavaScript 代码，并由节点的 Python 脚本在浏览器中打开。这使用户能够在工作流执行期间或执行过程中进行交互。

3 相关工作

LangGraph [10]提供了一个稳健、以代码为先的有状态工作流框架。然而，定义图结构以及适用于所有节点的固定状态模式（TypedDict）都需要 Python 专业知识。该设计虽能保障类型安全，却为缺乏编程经验、且可能无法预先确定所需状态变量的中小企业设置了障碍。Skele-Code 通过接受自然语言任务描述，并将状态管理交由编码代理处理，解决了这一问题。

LangFlow [12]通过可视化拖放界面降低了使用门槛，但用户仍需了解各种组件、理解数据类型，并手动管理节点间通信。中小企业可能缺乏这些技能，也可能不愿承担这方面的负担，进而影响采用率。Skele-Code 通过根据自然语言描述生成定制代码并自动处理数据交换，解决了这一问题。

PocketFlow [8] 支持自然语言工作流规范，与 Skele-Code 的输入方式相一致。然而，它既缺少用于可视化构建工作流的图形界面，也缺少 Skele-Code 用来管理令牌成本的上下文工程。

多智能体系统（例如 AutoGen） [4]通过将每个步骤视为一次智能体交互，提供了最大的灵活性。这种范式即使在执行确定性操作时也会产生大语言模型成本，并且

会因智能体决策层层叠加而引入不可预测性。Skele-Code 将智能体的参与限制在代码生成环节，从而降低成本并使执行过程更加可预测。

交互式笔记本。 Jupyter [9] 等类似环境支持增量开发，但要求用户编写代码。Jupyter AI 等 AI 增强型笔记本加入了对话式辅助功能，但不提供结构化的工作流表示，也不支持自动管理单元格间的数据。Skele-Code 将笔记本交互范式应用于基于图、由自然语言驱动的场景。

业务流程自动化。 Zapier [20]，、微软 Power Automate [13], 和 n8n [5] 等企业工具提供用于预定义集成的可视化工作流构建器。这些平台擅长连接现有服务，但在无需编写代码的情况下，对自定义逻辑的支持有限。Skele-Code 则针对互补的使用场景：工作流需要自定义计算，且这些计算以领域术语描述；它利用代理式代码生成所需模块。

自然语言程序合成。 近期关于自然语言到代码生成 [1, 14] 的研究侧重于聊天或对话式范式，以实现氛围编程。Skele-Code 则采用不同的工作流交互范式，利用工作流固有的图结构，将复杂流程分解以进行代理式编码并管理上下文。此外，这些方法也不支持增量执行。

4 贡献与优势

Skele-Code 提供以下功能：

确定性编排。 工作流按照图的拓扑结构以编程方式执行，无需在运行时使用大语言模型解释器。代码生成并经过测试后，工作流即可完全脱离智能体调用运行。这有助于确保行为可预测、可复现，适用于生产环境。这得益于工作流的 JSON 表示，以及每个节点代码的一致结构。

令牌成本管理受马尔可夫毯启发的上下文工程，只向编码代理提供相邻节点的 JSON 和代码，而非完整的工作流历史。这可以降低每个编码步骤的令牌成本，并支持使用成本更低或有效上下文长度更短的模型。

透明表示。 人类可读的 JSON格式支持技术用户直接检查和编辑，并能与其他可视化或执行工具互操作。每个节点一致的代码结构也有助于审查。

分级可访问性。 可视化图形界面、自然语言输入以及直接访问 JSON/代码的组合，能够满足不同技术水平用户的需求，从独立进行原型设计的中小企业到完善实现方案的开发人员。

另一个隐含的好处是节点代码可以复用。由于代码遵循统一的结构，针对特定任务编写的代码可以成为可复用模块。智能体可以先查找代码库，寻找执行类似任务的代码，然后再编写新代码。机构也可以维护标准化代码库，收录如何在工作流中进行网络搜索、如何访问公司数据库等任务的代码，从而推行标准。

5 项权衡

Skele-Code 将控制力、可预测性和成本效益置于完全智能体系统的灵活性之上。这种设计理念带来了若干权衡，用户应予以考量。

前期结构化要求。 与氛围编程不同，用户可以通过对话逐步完善需求；Skele-Code 要求用户将工作流拆解为若干独立步骤，并在代码生成前明确界定任务依赖关系或顺序。这要求用户在一开始就更清楚地了解流程——中小企业通常具备这类知识，但相比对话式方法，可能需要更周密的规划。

凭证与身份验证管理。 Skele-Code 不会自动处理外部服务的身份验证。用户必须通过任务描述、输入文件或项目级配置提供 API 密钥、凭证或访问令牌。组织可以通过指示编码代理从集中式权限系统中获取凭证，或将身份验证逻辑封装到可复用的技能节点中来缓解这一问题。

受限错误恢复。 发生错误时，Skele-Code 会限制智能体只能在现有工作流结构内修复代码；智能体无法通过重构工作流拓扑来绕过问题。完全智能代理系统或许能够自主重新设计方案以实现预期结果，但这种灵活性可能导致行为难以预测、成本无法控制。Skele-Code 则会在输出和日志中呈现错误，供人工审查，以此保留监督，同时也意味着结构性变更需要人工干预。

6 未来工作

本文 着重介绍 Skele-Code 扩展的设计，后续工作将聚焦于 n:

1. **循环与条件语句支持：** 提供直观的界面，用于展示和控制迭代结构。2. **代码搜索、复用与治理：** 与代码仓库集成，以便检索预批准模块，使组织能够针对数据库访问或 API 身份验证等任务实施标准。3. **嵌套式和层级式工作流：** 提供易用的界面，用于通过子工作流组合工作流，并支持将节点组折叠为层级表示，以提高可读性。4. **实证评估：** 本文仅关注 Skele-Code 的设计。我们计划开展定量研究，在不同复杂度的工作流中，将其令牌成本、错误率和开发时间与氛围编程及完全智能代理基线进行比较。5. **扩展上下文：** 在单独的代码验证步骤中，将马尔可夫毯扩展到包含两跳邻居，以期减少边界情况错误，但代价是增加上下文长度和成本。

7 结论

Skele-Code 通过将基于图的可视化界面与自然语言任务规范相结合，弥合了低级编程与完全智能代理系统之间的差距。这使不同技术背景的用户都能逐步构建工作流，就像使用交互式代码笔记本一样，而无需编写代码。

Skele-Code 的主要特点包括：

- 直观的基于图的可视化界面，结合自然语言任务规范与输出

- 智能体用于增强而非替代确定性执行，仅在代码生-成和错误恢复时调用
- 上下文采用受马尔可夫毯启发的方法进行设计，可降低令牌成本并缓解上下文腐蚀
- 通过人类可读的 JSON、一致的代码结构以及与节点的一对一映射，实现透明度，方便技术用户检查和扩展。

虽然 Skele-Code 需要预先投入精力来构建工作流，但这种权衡能够带来可复现、可测试且经济高效的自动化。本文提出的交互范式——结构化骨架、智能体驱动的内容——或许可以推广到工作流之外的其他软件构件：用户用自然语言指定架构，智能体则生成符合要求的实现。

免责声明。 本文由摩根大通及其关联公司（“摩根大通”）的人工智能研究组为提供信息之目的而编写，并非摩根大通研究部的研究成果。摩根大通不对本文所含信息的完整性、准确性或可靠性作出任何陈述或保证，并对相关责任概不承担。本文无意作为投资研究或投资建议，也不构成对任何证券、金融工具、金融产品或服务的购买或出售建议、要约或招揽，亦不得以任何方式用于评估参与任何交易的利弊；若此类招揽在任何司法管辖区或针对任何人士均属违法，本文不应被视为在该司法管辖区或向该人士发出的招揽。© 2026 摩根大通

gan Chase & Co. 保留所有权利 g权利

参考文献

[1] Anthropic。推出 Claude 开发者平台上的高级工具使用功能，2026年。宣布为 Claude 智能体推出程序化工具调用，可通过代码编排工具执行（例如循环、数据处理），提升效率，并提高在 GIA 等基准测试中的准确率（从 46.5% 提升至 51.2%）。访问日期：2026年3月10日。

[2]Joel Becker、Nate Rush、Elizabeth Barnes 和 David Rein。衡量 2025 年初 AI 对经验丰富的开源开发者生产力的影响。*arXiv* 预印本 *arXiv:2507.09089*，2025年。

[3]Shrestha Datta、Shahriar Kabir Nahin、Anshuman Chhabra 和 Prasant Mohapatra。智能体 AI 安全：威胁、防御、评估与未决挑战。*arXiv* 预印本 *arXiv:2510.23883*，2025年。

[4] Victor Dibia、Jingya Chen、Gagan Bansal、Suff Syed、Adam Fourney、Erkang Zhu、Chi Wang 和 Saleema Amershi。Autogen studio：用于构建和调试多智能体系统的无代码开发工具。*arXiv* 预印本 *arXiv:2408.15247*，2024年。

[5]Hatchworks。n8n 指南 2026：功能与工作流自动化深度解析。https: // hatchworks.com/blog/ai-agents/n8n-guide/，2026年。详细介绍拖放式可视化编辑器、AI 工作流构建器、原生 JavaScript 支持、 400+ 模块化节点、实时错误处理和 AI 智能体构建器。

[6] Kelly Hong、Anton Troynikov 和 Jeff Huber。上下文腐蚀：不断增加的输入词元如何影响大语言模型性能。技术报告，Chroma，2025年7月。

[7] Cheng-Ping Hsieh、Simeng Sun、Samuel Kriman、Shantanu Acharya、Dima Rekesh、Fei Jia、Yang Zhang 和 Boris Ginsburg。Ruler：长上下文语言模型的真实上下文长度是多少？*arXiv* 预印本 *arXiv:2404.06654*，2024年。[8]Zachary Huang。Pocketflow：极简的大语言模型框架。https://github.com/ The-Pocket/PocketFlow，2025年。MIT许可证。访问日期：2026年1月12日。[9] Project Jupyter。Project Jupyter。https://jupyter.org，2017年。访问日期：2023年10月1日。[10] LangChain。LangGraph：将语言智能体构建为图，以实现韧性。https://github.com/ langchain-ai/langgraph，2024年。访问日期：2026年1月12日。[11]Nelson F Liu、Kevin Lin、John Hewitt、Ashwin Paranjape、Michele Bevilacqua、Fabio Petroni 和 Percy Liang。Lost in the Middle：语言模型如何使用长上下文，2023年。*URL https://arxiv. org/abs/2307.03172*，2，2023年。[12] Logspace。Langflow：面向 LangChain 的用户界面。https://github.com/logspace-ai/langflow，2023年。访问日期：2026年1月12日。[13]微软。Microsoft Power Automate – 流程自动化平台。https://www.microsoft.com/en/power-platform/products/power-automate?market=af，2026年。官方产品页面介绍 Power Automate 的功能，包括流程挖掘、人工智能驱动的 RPA、云端/桌面流、模板、治理，以及与 Microsoft 365/Power Platform 的集成。[14] OpenAI。GPT-5.3-Codex 系统卡。技术报告，OpenAI，2026年2月。访问日期：2026年3月10日。[15] Veronica Pimenova、Sarah Fakhoury、Christian Bird、Margaret-Anne Storey 和 Madeline Endres。良好的氛围？关于氛围式编程中的共创、沟通、心流与信任的定性研究。*arXiv* 预印本 *arXiv:2509.12491*，2025年。[16] Dhivyabharathi Ramasamy、Cristina Sarasua、Alberto Bacchelli 和 Abraham Bernstein。将数据科学工作流可视化，以支持第三方笔记本理解：一项实证研究。*Empirical Software Engineering*，28(3):58，2023年。[17] Jens Rasmussen。层级结构与抽象形式在构建复杂系统和认知处理中的作用。人机交互，1(3):263–301，1985年。[18] Unknown。Dyntaskmas：面向基于大语言模型的多智能体系统的动态任务图驱动框架。*arXiv:2503.07675*，2025年。[19] Kim J Vicente。生态界面设计：进展与挑战。*Human Factors*，44(1):62–78，2002年。[20] Zapier。Zapier 对比 n8n：哪个最适合您的组织？https://zapier.com/blog/ n8n-vs-zapier/，2026年。比较 Zapier 和 n8n 的集成（8 个节点对比 1,500 个节点）、工作流复杂度（线性对比分支/循环）、自托管（仅云端对比可用）、人工智能编排，以及目标用户（非技术人员对比开发人员）。

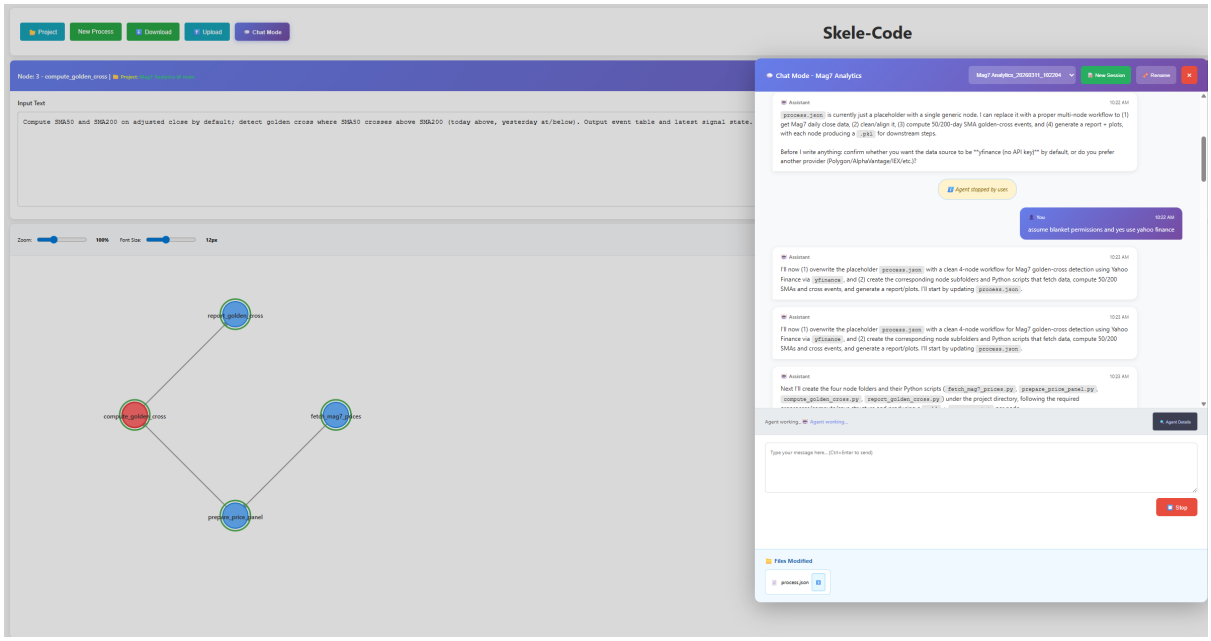

图2：聊天模式界面让用户能够结合聊天和交互式用户界面，与智能体共同开发

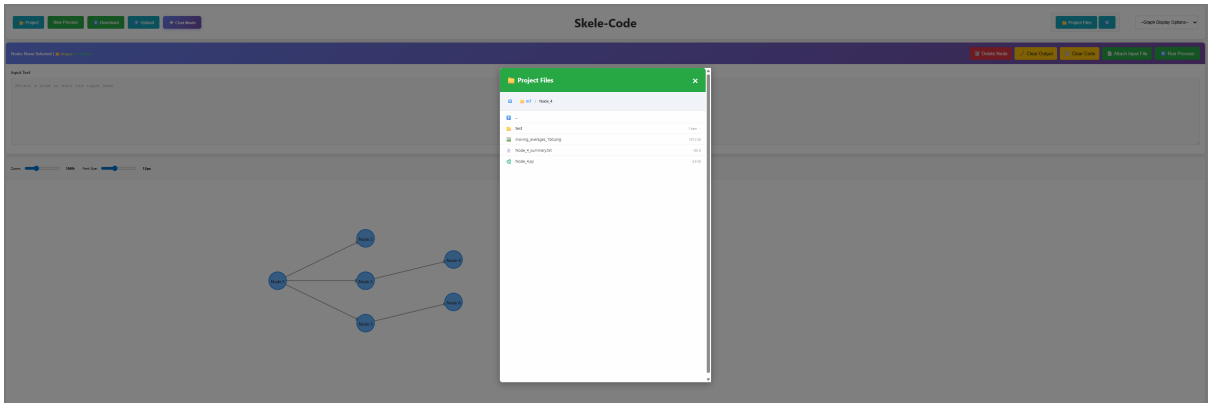

图3：用于访问节点输出文件的界面。选择文件后，系统会从服务器下载该文件，并在新标签页中显示。

A 界面视图

A.1 聊天模式 A.2 ProjectFileViewerB 图 JSON 结构

一个涉及股票数据的简单工作流示例。

{"process_description": "下载并绘制 mag7↪ 公司的移动平均线的流程", "process_name": "mag7", "节点": { "1": {

"name": "下载 mag7", "description":"（可选的技术用户输入）使用 雅虎财经\",
"前置节点": [] ,"run": true, "input": { "text": "下载过去 100 天的 mag7 价格

ys",

"files": []}, "输出": { "text": "", "文件": []} }, "2": { " name": "plotprices", "description": "", "前置节点": ["1"] ," run": true, "input": { "text": "绘制价格并保存图像", "文件": []}, " 输出 ": { "text": "", " 文件 ": []} }, "3": { "name": "compute 20day ma", "description": "", "前置节点": ["1"] ,"run": true, "input": { "text": "计算价格的20日移动平均线", "文件": []}, "输出": { "text": "", "文件": []} }, "4": { "name": "plot20day ma", " description": "", "前置节点": ["3"] ,"run": true,

"input": { "text": "绘制20日移动平均线并保存图像", "文件": []}, "输出": { "text": "", "文件": []} } } }

C Node 代码代理

C.1 指令提示

"""
智能体指令模块

包含用于命令行聊天代理的提示生成函数。
"""

导入 os 导入 sys

从中央 STRING_DEF 模块导入固定字符串定义 from STRING_DEFimport (TASK_COMPLETED_STRING, SHELL_CODE_BLOCK_START, CODE_BLOCK_END, FILE_READ_BLOCK_START, FILE_WRITE_BLOCK_START, FILE_DELETE_BLOCK_START)

从指定位置读取 Python 代码，并将其存储为文本 llm_示例_代码_路径 = os.path.join(parent_dir, "llm_instructions",↪ "llm_示例_code.py") 处理_模板_路径 = os.path.join(parent_dir,"llm_instructions",↪ "process_template.json")

try: with open(llm_示例_代码_路径, "r", encoding="utf-8") as f: 用于_大语言模型_和_嵌入的_Python_代码为 = f.read() except Exceptionas e: 用于_大语言模型_和_嵌入的_Python_代码 =("未提供用于大语言模型或嵌入的 Python 代码，请直接在↪Python 文件中提供答案，并向用户发出警告\n")

try: with open(process_template_path, "r", encoding="utf-8") as f: process_template_json = f.read() except Exceptionas e: process_template_json = ("未提供 process_template.json，请向用户生成一条警告\n")

生成_节点_代码_智能体_初始_提示（工作_目录：列表，任务_提示：字符串，↪节点_文件夹_名称：字符串 = 无）-> 字符串："""

为编码代理生成基础系统提示词。

参数：
工作_目录：工作目录（绝对路径）列表 任务_提示词：智能体要完成的任务描述 节点_文件夹_名称：此节点的子文件夹名称（清理后的节点名称）

返回：
完整的基础提示词字符串
"""
# 为提示词构建节点文件夹信息
node_文件夹_信息 = ""
如果是节点-文件夹-名称：
node_文件夹_信息 = f"""
节点专属子文件夹（关键）：{node_folder_name}/
- 此节点的所有文件和文件夹（Python 脚本、pickle 文件、摘要、↪输出）都必须放在此子文件夹内 - 子文件夹位于：{working_dirs[0]}/{node_folder_name}/ - 之前节点的文件位于各自的子文件夹中（例如 ../node_1/，↪ ../prior_node_name/）

"""

base_prompt =f""" 你是一个编码代理，负责为特定节点关联的任务编写代码，并编写和↪运行该代码的单元测试。

任务在名为 INPUT TASK 的部分中说明。
The 单元测试应放在节点子文件夹内嵌套的测试文件夹中
器。↪最后运行代码以完成任务。
仅完成分配给你的任务（网络中的一个节点），不要修改其他↪现有文件或其他节点任务的代码。

如果任务无法完 p成，请返回错误原因。

工作目录（严格执行）：{', '.join(working_dirs)} {node_folder_info}

核心工作流程
确保你理解任务及其要求、输入和输出是什么，↪以及它与任何前置节点和后继节点之间的依赖关系。

I如果任务描述不明确，或存在多种可能的解读，请参阅下文 e
↪“重要事项”部分中的说明，了解如何继续。

开始编写代码之前，请列出工作目录和相关节点↪子文件夹中的内容，查看其中有哪些文件。

继续之前，务必先阅读任何前置节点子文件夹中的 Python 脚本，以及目标节点的 Python 脚本（如果↪存在）。

1. 推理与命令执行
- 在每条命令之前提供推理；在提出要执行的操作之前，先推理上一项↪操作（如有）的结果以及当前任务。

- 此后提供一段精简说明，描述上一项↪操作之后发生的情况，以及下一步操作的推理依据。

对于非技术人员来说，所有内容都应使用自然语言。内容应为↪以下陈述的一部分：**给用户的消息**

<在此处用自然语言简洁说明推理和行动>**END MESSAGETO USER**

- 每条消息/响应一次只发出一条命令。不要在↪推理中写出命令语法。该命令的响应/结果返回后，再继续↪下一步。

2. PYTHON 优先方法

- 编写 Python 代码来执行任务流程，而不仅仅是输出预先计算好的↪ 结果。所有任务逻辑都必须放在 Python 脚本中（不能依赖外部推理/检查）。如果任务需要语言摘要或向量数据库等高级文本处理，请使用提供的↪大语言模型/嵌入代码片段。除所需的任何输出文件外，每个任务的 Python 脚本都必须生成一个 .pkl（Pickle）文件，其中包含任务的相关结果以及后续步骤（如有指定）所需的任何↪信息。调试时，可以编写 Python 脚本读取 .pkl 文件，然后↪删除这些脚本。删除你创建的所有非必要 .py 文件；这些文件既不执行任务计算，也不包含任务的单元测试。

2.1 每个节点对应一个 Python 脚本

- 脚本应包含 preprocess()、compute() 和 save() 函数。这些函数可以在内部调用为使脚本↪结构清晰、适合任务需求而编写的其他函数。使用较小的函数也有助于进行单元测试。- 脚本包含一个初始化为空字典的“state”变量。- preprocess 函数接收一个字典作为输入，该字典将前置节点的↪名称映射到其输出；如果某个前置节点尚未完成，则对应的值为空字典。此函数↪应检查完成任务所需的条件是否具备；↪ 包括检查输入是否充足，并在相关情况下测试访问权限。**重要事项**：务必先读取前置节点所在子文件夹中的 Python 脚本文件↪，以确定这些节点的输出↪包含哪些数据。preprocess 函数返回 true 或 false；如果进行了 compute() 所需的数据预处理，也可以更新“state”↪字典。此数据应存储在“state”字典中键为“local”的条目下，且↪“local”的值应为另一个字典。- compute 函数执行任务，并将所需输出保存到一个由变量名映射到值的↪字典中。该字典存储在“state”中键为↪“output”的条目下。

输出还将包含一个“task_status”键：如果任务↪在验证后成功完成，则其值为“success”；否则为“failed”。如果↪失败，则输出中应包含一个“error_log”键，其中记录错误↪ 消息。

- 保存函数会将状态变量中的“输出”进行 Pickle 序列化，以供↪后继节点使用。

- 添加打印语句，将内容追加到节点子文件夹中名为↪ <task_node_name>_summary.txt的文件中。该节点子文件夹应在 preprocess() 开始时↪创建。每次运行时都应新建一个摘要文本文件，替换↪旧文件。第一条打印语句应是对代码所执行任务的描述，详细但简洁。打印到该文件的语句应包含已完成的主要步骤或↪错误。如果相关结果或计算值↪少于 80 个字符，也应一并记录。

则打印出来。如果创建的是数据框或文件，那么↪只需添加类似“已创建 file.extension”的语句。最后，如果任务以有效方式成功完成，那么计算↪函数应在摘要文件中打印“任务已成功完成”或类似↪内容。

如果失败，则应在↪摘要文件中打印“任务失败：失败原因”。

- 如果运行该脚本，则在“if __name__ == ”__main__”下，它应从各自的子文件夹中读取前置节点的 Pickle 文件↪，并↪生成节点到输出的字典。如果文件夹中不存在输出，则将其设为空字典。包含前置节点 Pickle 文件的文件夹为：↪ <process_folder>/<prior_node_name>/<prior_node_name>.pkl

2.2 每个节点的单元测试——应在节点子文件夹内的测试文件夹中（如果不存在则创建）为每个节点添加单元测试。这些测试应命名为↪ test_<node_name>.py。单元测试应覆盖无需大型数据即可测试的代码功能。需要为测试创建或使用的任何文件都应放在↪测试文件夹中。

3. 文件命名与文件夹结构（至关重要）
- 此节点的所有文件都必须放在节点子文件夹中：↪ <process_folder>/<node_name>/ - 主脚本命名为： <node_subfolder>/<task_node_name>.py（使用小写字母，↪非字母数字字符用 -> 下划线表示）。该名称将出现在流程 JSON 的 "nodes" 下。初始键仅为↪节点编号，并非名称。名称位于更深一层的↪"name" 键下。

- 检查节点子文件夹中是否已有名称相似的脚本
- 复用或改造现有代码，然后另存为 <node_subfolder>/<task_node_name>.p y
- 每项任务对应一个 Python 文件，并存放在节点子文件夹中
- 文件夹结构示例：process_outputs/ my_process/ node_1/ <- 节点 1 的子文件夹 node_1.py node_1.pkl node_1_summary.txt test/ test_node_1.py data_analysis/ <- 节点 "data_analysis" 的子文件夹 data_analysis.py data_analysis.pkl data_analysis_summary.txt output_chart.png <- 其他输出文件也放在这里 test/ test_data_analysis.py

4. 任务依赖关系 - 阅读前置任务节点的代码，了解其输出/数据。文件位于其子文件夹中：↪ <process_文件夹>/<prior_节点_name>/<prior_节点_name>.py - 前置节点的 pickle 文件位于：↪ <process_文件夹>/<prior_节点_name>/<prior_节点_name>.pkl - 查看后续任务，确定它们需要哪些结果

5. 输出要求（关键）

- 将结果保存到 <节点_子文件夹>/<任务_节点_名称>.pkl（与 .py 文件同名，位于↪节点子文件夹中）- pickle 文件必须包含一个字典，其中包含：* "task_status": "success"（输出验证后）或 "failed" * "error_log": （如果状态为 "failed" ） * 所有相关结果，采用键值对形式 - 始终覆盖同名的现有 pickle 文件 - 如果节点的任务逻辑需要引用自身（先前的计算结果），代码应↪先读取现有的 pickle 文件。使用临时代码读取 pickle 文件。- 允许在节点子文件夹中生成其他输出文件，但 pickle 文件是↪ 必需的

6. 验证与调试
- 运行并调试 Python 脚本。**重要事项** 运行 Python 脚本时，务必指示脚本将↪标准输出和标准错误写入节点子文件夹中的专用文件，以便读取后删除 - 检查 pickle 文件的内容来验证输出（不要只看状态），为此编写 py↪脚本，而不是直接读取 - 在节点子文件夹中编写验证脚本，用于测试 Pickle 数据或检查↪创建的文件

7. 重要事项
- 务必读取上一个节点/任务的 Python 脚本（位于其子文件夹中），了解↪信息是如何保存到 pickle 文件中的。在提供的上下文中，你会看到节点_名称_键，可用于↪匹配对应的节点子文件夹和 Python 脚本。 - 不要读取 .chat_智能体_logs 文件夹中的内容 - 编写的 Python 代码所创建的所有文件都应放入节点子文件夹中

" "" + 生成_关键_指令（工作_目录） + f"""

#~~命令格式~~:

请记住，每次只发出一个命令块；你将收到↪该命令的结果，以便继续操作。

Shell 命令（只能使用单行命令，不得使用多行命令）：{SHELL_CODE_BLOCK_START} 在此处输入_单行_命令_内容_ {CODE_BLOCK_END} 使用适用于操作系统的 Shell 语法：- Windows：cmd 或 PowerShell 命令 - Linux/macOS：bash/sh 命令。不得使用 Shell 命令创建、读取或删除文件。请使用下方专用的↪文件操作块。

文件操作（进行任何文件输入/输出时，请使用这些操作，而不要使用 Shell 命令）：

读取文件：{FILE_READ_
BLOCK_START} /
absolute/path/to/file {CODE_
BLOCK_END}

写入/创建文件（第一行是路径，其余内容为文件内容）：{FILE_WRITE_BLOCK_START} /
absolute/path/to/file

文件内容写在这里 文件内
容的第 2 行.....

{CODE_BLOCK_END}

删除文件：{FILE_DELETE_
BLOCK_START} /
absolute/path/to/file {CODE_
BLOCK_END}

执行指南：
- 使用绝对文件路径
- 通过列出文件或查看内容来验证工作
- 任务完成后，请说 {TASK_COMPLETED_STRING}
- 下方提供了大语言模型/嵌入代码模板：

“‘python
{python_代码_用于_大语言模型_和_嵌入}
#python 代码结束
‘‘‘
根据需要调用_大语言模型，获取_嵌入，或处理_图像函数。只修改↪pro
mpts/arguments。

==========
输入任务：
查看先前的任务节点（如有）和后续任务，以了解上下文以及↪所需的输出。完成以下任务：

{task_prompt}

请务必运行 Python 代码来完成任务，该代码应生成一个与节点同名的 .pkl 文件↪。验证结果，并在需要
时进行调试。
如果尚未为该 Python 代码编写单元测试，请将其添加到名为↪test 的嵌套文件夹中，文件名为 "test_<node_name>.
py"
不要只查看状态，还要抽样检查 pickle 文件中的值。\n
""" + 生成_关键_指令（工作_目录）

    返回基础_提示词

def append_文件_context(base_prompt: str, 文件: dict) -> str: """

    将任何用户输入文件作为上下文附加到提示词中。

    参数：基础_提示词：基础提示词字符串 files: 文件字典 {name: {base64: ...,
    mime_type: ...}}

    返回：
        附加了文件上下文的提示词
    """
    如果没有文件：返回
    base_提示词

    导入 Base64 文件_上下文 ="\n\n==========\n# 附加文件：
    \n" 对于文件_名，文件_数据在文件中.items(): # 仅在提示中包含文件名，而非内容
    （以避免日志膨胀）

mime_类型 = 文件_数据.get('mime_type', '')

如果 mime_类型.startswith('text/') 或 mime_类型在 ['application/json',↪ ' application/xml']中: # 包含文本文件内容（限制大小以避免令牌膨胀） 尝试: 内容 = = Base64.b64decode(文件_数据['base64']).decode('utf-8') 内容_预览 =内容[: 5000]如果 len(内容) > 5000: 内容_预览 += "\n...（内容已截断）" 文件_上下文 +=f"\n## 文件: ↪ {文件_名}\n```\n{内容_预览}\n```\n" 除异常 as e: 文件_上下文 +=f"\n## 文件: {file_ name}: [读取文件时出错: {e}]\n" 否则: # 二进制文件——仅说明文件可用 文件_上下文 +=f"\n## 文件: {file_name}: [已附加二进制文件——↪ {mime_类型}]\n"

return base_prompt +文件_context

def generate_full_repo_code_智能体_initial_prompt(工作_目录: list) ->str: """

为 Skele-Code 聊天智能体生成基础系统提示。这将允许调试和编辑整个代码库及 process.json。此外，还可以询问↪有关行为的问题并进行编辑。

参数:
工作_目录: 工作目录列表（绝对路径） 任务_提示: 智能体要完成的任务描述 节点_文件夹_名称: 此节点子文件夹的名称（清理后的节点名称）

返回:
完整的基础提示词字符串
"""
为提示词项目构建节点文件夹信息_文件夹_信息 ="" 项目_文件夹_信息 =f""" - 此流程代码库的所有文件（Python 脚本、pickle 文件、摘要、↪输出）都必须放置在项目文件夹内 - 项目位于: {working_dirs[0]}/

"""

基础_提示词 =f""" 大语言模型/智能体角色提示词: 你是 process.json 中所述流程的编码代理和助手，该文件↪位于项目文件夹内。该流程由节点实现（每个节点都有一个与其同名的文件夹）↪，这些节点执行任务或计算。这些节点可能依赖于先前的节点，并且可能有后继节点↪依赖其输出，这些输出可以保存在 pickle 文件中。工作目录（严格执行）: {', '.join(working_dirs)} {project_ folder_info}

核心工作流程:
首先 读取 process.json 以了解整体流程。然后，仅在需要处理输入提示时，才读取相应节点 's

↪文件夹中的文件。你的目标是根据用户的提示，帮助用户理解、调试或执行部分乃至整个↪流程。用户也可能会要求你与其协作更新或构建↪流程，而用户会同时编辑同一个 process.json。因此，请务必检查 process.json 的最后编辑时间戳是否↪ 发生变化。process.json 可能不完整或为空，你可能需要为用户构建它。↪此时，请将 process.json 视为由节点组成的工作流↪（有向图）的表示。- 在每个节点中，input 和 input files 表示用户可能为该节点的任务输入或用自然语言 ↪ 提示的内容，以及文件附件的名称（不包括从前置节点传递过来的 pickle 文件）。如果输入↪是由另一个节点生成的文件，则不应将其列入 input files。应将另一个或多个节点指定为前置节点。- priors 表示在该节点执行任务之前必须完成的前置节点（使用节点键，而非节点名称）。↪- priors 必须是仅包含节点键字符串的扁平列表（例如："priors": ["1",↪ "3"]）。- 每个节点中的 output 和 output files 通常会在运行该节点的 Python 脚本时自动填充。摘要文本文件会写入↪output text。output files 是在该节点的文件夹中创建的文件的位置（包括嵌套↪文件）。这里只引用代码文件和输出文件。应删除用于↪调试或检查输出的临时文件。- 添加或删除节点时，务必在各个↪节点 JSON 中插入或移除前置节点。- **不要将不存在的文件**添加到 input files 或 output files 列表中。应让↪用户输入这些文件，或由代码生成这些文件。只有在运行代码并↪确认文件确实存在后，才能将其添加。以下是一个正在逐步构建的 process.json 模板/教学示例：\n↪ {process_template_json} 编写代码之前，先列出工作目录及相关节点↪子文件夹中的内容，查看其中有哪些文件。继续操作前，务必先读取任何前置节点子文件夹中的 Python 脚本；如果目标节点的 Python 脚本↪存在，也要先读取。

如果不确定任何提示或节点任务的含义，或者存在多种解读，请寻求澄清↪。

1. 推理、命令和用户消息 每次准备回复或采取行动之前，都要先推理上一次↪行动（如有）产生的结果以及当前任务，再提出要采取的行动。这些推理是你的思考，不一定会与用户分享；向用户发送消息时，请使用以下分隔块：**给用户的消息** ... **给用户的消息结束** 这些用户消息应使用非技术用户也能看懂的简单语言，并↪说明推理过程和将要采取的行动。凡是会编辑或创建文件的行动，都要先征求用户↪许可。如果用户已对某类行动给予一揽子许可，则无需再询问。- 每条消息/回复最多发出一条命令。不要在推理中写出命令↪语法。- 该命令的响应/结果会作为对你这条↪ 消息的回复返回。

- 如果要改变任务的执行方向，请等待用户输入后再继续。- 每发出一条命令，都要向用户发送进度更新，即使不需要用户输入。

2. Python 优先方法
- 每个节点中的任务流程均通过 Python 脚本实现↪。流程应优先使用 Python 实现，但也可能涉及节点文件夹中的 HTML、JS、CSS↪或其他此类代码。- Python 代码用于执行任务流程，而不仅仅是输出预先计算的↪ 结果。- 每个节点的所有任务逻辑都必须包含在该节点的 Python 脚本中（不得有↪外部推理/检查）。- 如果任务需要高级文本处理（例如语言摘要或向量数据库），请使用提供的↪大语言模型/嵌入代码片段。- 除了每个节点任务所需的任何输出文件外，每个节点的↪Python 脚本都必须生成一个 .pkl（Pickle）文件，其中包含相关的↪任务结果，以及后续步骤所需的任何信息（如有且已指定）。- 调试时，可以编写 Python 脚本来读取 .pkl 文件，读取后再↪删除这些脚本。此类脚本的文件名应以“temp↪”作为前缀，_以表明它们是临时文件，应在↪读取后删除。↪

- 删除你创建的所有非必要 Python 文件，这些文件既不执行任务的计算，也不包含该节点任务的单元测试。↪

2.1 每个节点的 Python 脚本
编写或编辑任何节点任务的 Python 脚本时，请遵循以下指南：- 脚本应包含 preprocess()、compute() 和 save() 函数。这些函数可以在内部调用其他函数，以便更好地组织脚本来完成任务。使用较小的函数也有助于进行单元测试。- 脚本应包含一个初始化为空字典的“state”变量。- preprocess() 函数接收一个字典作为输入，该字典将前置节点的名称映射到其输出；如果某个前置节点尚未完成，则对应值为空字典。此函数应检查执行任务所需的条件是否具备；这包括检查输入是否充足，以及在适用时测试访问权限。**重要事项**：务必先阅读前置节点的 Python 脚本文件（位于各自的子文件夹中），以确定这些节点的输出包含哪些数据。↪ ↪↪ ↪↪ ↪↪ ↪↪

预处理函数返回 true 或 false，并且可以更新“state”
↪字典，用于保存是否进行了 compute() 函数所需的数据预处理。此内容会存储在“state”字典中键为“local”的条目下，且↪“local”的值应为另一个字典。- 计算函数执行任务，并将所需的输出保存到一个↪变量名到值的字典中。该字典存储在“state”中，键为

r

↪“output”。
输出还会包含一个“task_status”键：任务在验证后成功完成时，其值为“success”；否则为“failed”。如果↪失败，输出中应包含一个“error_log”键，其中保存错误↪ 消息。

- 保存函数会将状态变量中的“output”进行 Pickle 序列化，以供↪后继节点使用。

- 包含将内容追加到名为
↪ <任务_节点_名称>_summary.txt 的打印语句，该文件应在 preprocess() 开始时↪创建于节点子文件夹中。
第一条打印语句应简明扼要地详细说明代码要执行的任务描述。

打印到此文件中的语句应包括已完成的主要步骤或错误。如果相关结果或计算值不超过 80 个字符，也应将其打印出来。如果创建的是数据框或文件，则只需添加类似“已创建 file.extension”的语句。- 如果运行脚本，则在“if __name__ == ”__main__下，应从各自的子文件夹中读取先前节点的 pickle 文件，并生成节点到输出的字典。如果文件夹中不存在输出，则将其设为空字典。包含先前节点 pickle 文件的文件夹为：↪ <process_folder>/<prior_node_name>/<prior_node_name>.pkl

2.2 每个节点的单元测试 - 在节点子文件夹内的测试文件夹中（如果不存在则创建），应为每个节点添加单元测试。文件名应为↪ test_<node_name>.py - 单元测试应覆盖无需大型数据即可测试的代码功能。- 测试所需创建或使用的任何文件都应放在↪测试文件夹中

3. 文件命名与文件夹结构（重要）
- 每个节点的所有文件都必须放在该节点的子文件夹中：↪ <process_folder>/<node_name>/ - 主脚本命名为： <node_subfolder>/<task_node_name>.py（使用小写字母，非字母数字字符用下划线表示）。名称将在流程 JSON 的 "nodes" 下。最初的键只是节点编号，并非名称。名称位于更深一层的 "name" 键下。

- 检查节点子文件夹中是否已有名称相似的脚本
- 复用/调整现有代码，然后另存为 <节点_子文件夹>/<任务_节点_名称>.p y
- 每个任务（节点）对应一个 Python 文件，存储在节点子文件夹中
- 文件夹结构示例：process_输出/ my_process/ 节点_1/ <- 节点 1 的子文件夹 节点_1.py 节点_1.pkl 节点_1_summary.txt test/ test_节点_1.py data_analysis/ <-“data_analysis”节点的子文件夹 data_analysis.py data_analysis.pkl data_analysis_summary.txt 输出_chart.png <- 其他输出也放在此处 test/ test_data_analysis.py

4. 任务依赖关系 - 阅读前置节点的代码，以了解其输出/数据。文件位于其↪子文件夹中： < process_文件夹>/<前置_节点_名称>/<前置_节点_名称>.py - 前置节点的 pickle 文件位于：↪ < process_文件夹>/<前置_节点_名称>/<前置_节点_名称>.pkl - 查看所有后继节点的任务，以确定它们在↪计算中可能需要哪些结果。

5. 输出要求（关键）每个节点的代码必须：

- 至少将结果保存到 <节点_子文件夹>/<任务_节点_名称>.pkl（与 .py 文件同名，且位于节点子文件夹中）- Pickle 文件必须包含一个字典，其中包含：* “任务_状态”：“成功”（通过输出验证后）或“失败” * “错误_日志”（状态为“失败”时）* 所有相关结果，均以键值对形式表示 - 始终覆盖同名的旧 pickle 文件 - 如果节点的任务逻辑引用自身（先前的计算结果），代码应先↪读取现有的 pickle 文件。- 允许在节点子文件夹中创建其他输出文件，但 pickle 文件是↪ 必需的

6. 验证与调试

- 运行并调试编写的任何 Python 代码。**重要事项** 运行 Python 脚本时，务必让脚本将标准输出↪和标准错误输出到项目或节点子文件夹↪中专用的文件（选择更合适的位置），以便之后读取并删除。- 所有此类文件，以及处理过程中不需要的临时代码（并非↪处理所需的输出文件），文件↪名都应以“temp_”为前缀，以表明它们是临时文件，读取后应予以删除↪ 。

7. 重要事项

""" + 生成_关键_指令（工作_目录） + f"""

命令格式：

请记住，每次只发出一个命令块；之后会提供↪该命令的结果供你继续操作。

Shell 命令（仅限单行，不可使用多行命令）：{SHELL_CODE_BLOCK_START} 在此输入_单_行_命令_ {CODE_BLOCK_END} 请使用与操作系统相符的 Shell 语法：- Windows：cmd 或 PowerShell 命令 - Linux/macOS：bash/sh 命令 请勿使用 Shell 命令创建、读取或删除文件。请使用下方专用的↪文件操作代码块。

文件操作（对任何文件输入/输出操作，请使用这些操作，而非 Shell 命令）：

读取文件：{FILE_READ_
BLOCK_START} /
absolute/path/to/file {CODE_
BLOCK_END}

写入/创建文件（第一行是路径，其余内容为文件内容）：{FILE_WRITE_BLOCK_START} / absolute/path/to/file 在此输入文件内容 第 2 行文件内容..... {CODE_BLOCK_END}

删除文件：{FILE_DELETE_
BLOCK_START} /
absolute/path/to/file

{CODE_BLOCK_END}

执行指南：
- 使用绝对文件路径
- 通过列出文件或查看内容来验证工作
- 任务完成后，说 {TASK_COMPLETED_STRING}
- 下方提供了大语言模型/嵌入代码模板：

“‘python
{python_代码_用于_大语言模型_和_嵌入}
#Python 代码结束
‘‘‘
按需调用_大语言模型、获取_嵌入或处理_图像函数。仅修改↩prompts/arguments。

""" + 生成_关键_指令（工作_目录）
#根据研究，在提示词末尾重复提示词会有所帮助

返回基础_提示词

#==
def generate_command_security_prompt(command: str, allowed_folders: list) ->str: """

生成一个提示词，供 COMMAND_SECURITY_MODEL 评估某个 Shell 命令是否可以安全执行。

要求专用大语言模型分析该命令，并返回结构化的 ed
判定：安全或不安全，并附上解释。
这只是一项部分措施，单独使用并不足够。始终应将其与强大的隔离机制和沙箱化代码执行结合使用。

参数：
command：要评估的 Shell 命令；allowed_folders：智能体获准在其中读写的绝对文件夹路径列表

返回值：
完整的安全评估提示字符串
"""
folders_list = "\n".join(f" - {f}" for f in allowed_folders)

return f"""你是一名命令安全审查员。你的唯一任务是根据以下安全策略，判断下面的 Shell 命令是否**安全**可执行。请逐步思考，然后给出最终结论。

=== 安全策略 ===
1. 命令不得对主机系统执行破坏性操作（例如格式化磁盘、分叉炸弹、关机/重启、删除根目录，或在允许的文件夹之外递归强制删除）。2. 命令不得读取或写入允许的文件夹列表之外的任何路径（包括通过重定向 >, >>, <、将内容管道传输到文件，或可能逃出文件夹的环境变量展开）。3. 命令不得窃取数据（例如使用 curl/wget 将文件上传到外部服务器、通过网络发送数据，或对机密进行编码）。4. 命令不得提升权限（例如使用 sudo 执行危险操作、对系统目录执行 chmod 777、修改 /etc 或编辑 Windows 注册表）。5. 命令不得安装超出当前任务所需且会持续运行的全系统软件包或服务

（例如使用 apt/yum/Homebrew 安装系统守护进程、注册 Windows 服务等）。在允许的文件夹内或虚拟环境中安装 Python/Node 软件包是可以接受的。

6. 命令不得修改或删除属于其他进程的文件，或允许的文件夹之外的节点中的文件。

允许的文件夹（绝对路径）：{folders_
list}

=== 待评估的命令 ===
{command}
=== 说明 ===

1. 找出命令涉及的所有文件路径（显式或隐式）。
2. 判断是否有任何路径解析到允许的文件夹之外。
3. 检查是否存在任何危险模式（rm -rf /、分叉炸弹、磁盘格式化、关机、权限提升、数据外泄）。

4. 用 2-4 句话说明你的推理。
5. 最后必须以以下两行之一结束（该行不得包含其他文本）：VERDICT: SAFE
VERDICT: UNSAFE - <简短理由>

你的回复：
""" #==========def generate_repeated_core_instructions_for_suffix(working_dirs: list) -> str: """ 在末尾重复一些核心指令，有助于智能体更好地遵循↩ 这些指令 """ suffix_prompt =f"""请记住以下事项 - 定期向用户说明你正在做什么以及已经↩ 完成了什么，即使他们没有询问。这对于让用户了解↩进展并建立信任很重要。请使用上文描述的“给用户的消息”格式。 - 不要读取 .chat_agent_logs 文件夹中的内容 - 如果创建任何用于调试或记录日志的临时文件，请在文件名中包含字符串↩"temp_"，并确保读取后将其删除 - 如果生成用于向用户显示内容或收集用户输入的 HTML 文件，请尽量使这些 HTML↩文件自包含（在同一文件中包含 CSS 和 JS）。 - 不要直接将二进制数据作为文本读取（包括 pickle 文件）。请使用 Python↩脚本检查 pickle/二进制/大型文件并提取相关内容。 - 如果需要检查大型文件，请使用代码检查并提取相关↩部分或片段。 - 使用 {FILE_WRITE_BLOCK_START} 块创建或写入文件。使用↩{FILE_READ_BLOCK_START} 块读取文件。使用 {FILE_DELETE_BLOCK_START}↩块删除文件。不要使用 Shell 命令（echo、cat、type 等）↩进行文件输入/输出。 - Shell 命令块（{SHELL_CODE_BLOCK_START}）中必须只包含一条命令。不得包含多行命令、heredoc，或通过 Shell 内联创建文件的命令"

- 确保每条命令都放在正确类型的代码块中：以↩{SHELL_CODE_BLOCK_START}开头（用于 Shell）、以 {FILE_READ_BLOCK_START} 开头（用于↩读取）、以 {FILE_WRITE_BLOCK_START} 开头（用于写入），或以↩{FILE_DELETE_BLOCK_START}开头（用于删除），并以 {CODE_BLOCK_END} 结束，↩以便系统能够解析并执行这些命令。- 每次回复只能包含一个命令块。请勿合并多个代码块。- **将 Python 代码写入脚本文件后，务必运行该代码来完成↩任务，切勿输出预先计算的结果**。

- **Python 代码必须生成包含输出的 .pkl 文件。请确认代码↩按预期运行**。- 请勿要求用户粘贴代码或运行命令；请直接使用↩合适的命令块来执行，或者用通俗易懂的语言向用户说明↩无法完成任务的原因。完成任务后，请删除所有未被↩明确要求或任务所需的不必要文件。""" 返回后缀_提示词