

引言

AI 模型能够处理形式不规整的数据,理解含义不明确的信息,提出计划,并操作工具。要将这些能力转化为可靠的应用,就必须控制模型能看到什么、可以做什么,以及它的决策如何影响应用。Lyquid Flow 为此提供了一套组织框架,让构建者能够掌控自己的应用。

设想一个 AI 应用,需要根据陆续到来的订单,分配可用库存和供应商产能。需求与供货信息可能以邮件、文档或不完整的描述呈现。应用必须理解这些输入、比较不同方案、遵守数量限制与交付承诺,并判断何时需要采购人员批准。它既要允许模型灵活探索,也要按照明确规则进行检查和审批,将智能体与预定义工作流各自的优势结合起来。[2]

这类系统中,相当一部分专业经验体现在模型调用之间的组织与决策中。熟练的使用者会打磨提示词、选择上下文,并判断何时继续迭代、何时转换思路,从而用同一个模型获得更好的结果。[3] 他们实际上承担了模型之外的流程控制:收集哪些依据、采纳哪些结果,以及何时需要审批或转交进一步处理。这些选择共同形成了一套方法,可以转化为明确、可重复执行的应用逻辑。

这些专业经验归谁所有、由谁掌控,也随之成为问题。如果一套方法主要保存在平台上的提示词和对话中,创建者需要付费才能使用,却未必容易将它做成能够独立部署、改进和销售的产品。当服务条款与用户设置允许平台将这些交互用于训练时,这些内容还可能帮助服务商改进其掌控的模型资产。[4][14] 问题的核心在于:积累下来的诀窍由谁掌控,又由谁从复用中获益。

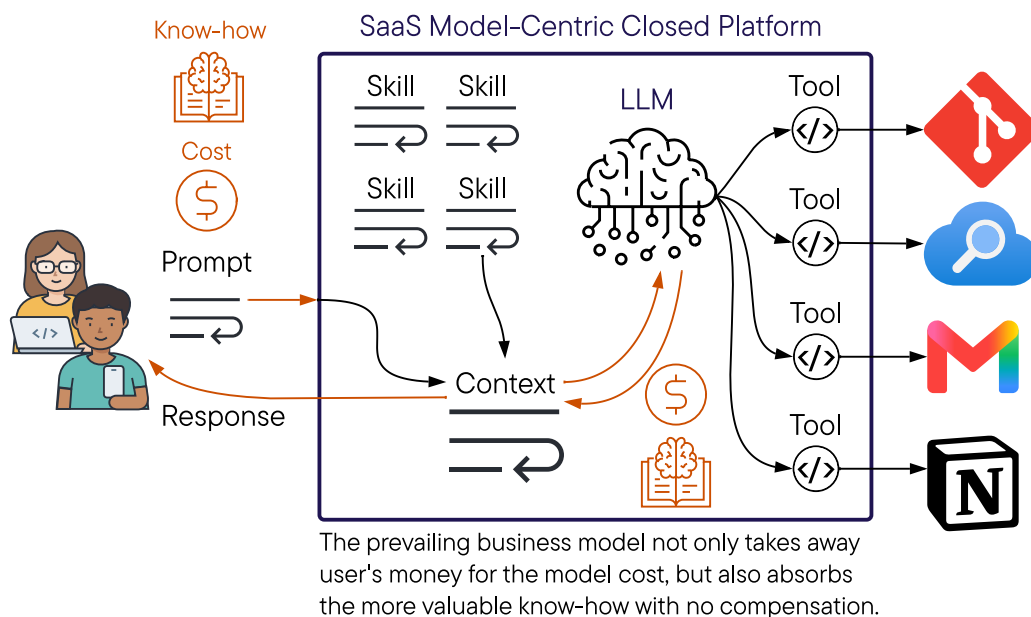


图 1 以模型为中心的 SaaS

提示词与技能将这些专业经验送入模型的上下文,而平台掌握着执行过程及外部工具的访问权。更丰富的上下文表达语言和专门化技能,也可能需要特定的专有模型充当“解释器”,才能发挥作用。在这种情况下,仅仅导出描述它们的文字,并不能将原有能力一并带走;这套方法及其产品价值依然与服务商绑定。

近期围绕纳维-斯托克斯问题成果归属的争议,也凸显了这一关切:通过 AI 工具分享的专业知识会如何被复用,贡献者又能否获得应有的认可。[13]

Lyquid Flow 将组织模型调用的整套方法封装为可复用的软件单元。如何选择上下文、检查输出,以及决定下一步做什么,都成为构建者掌控的应用逻辑。Lyquid Flow 希望让应用价值更少依赖特定模型:模型可以更换,方法仍是构建者自己的资产。一个完整的 Flow 本身也可以成为另一应用中的组件。Flow 可以在多个层级上分支和循环,随着新信息到来,重新审视决策并改进结果。

构建者可以持续改进这些方法、管理其版本、在不同产品中复用,并决定他人以何种方式使用它们。目标是建立一个开放平台,让创建者将专业诀窍转化为自己拥有、能够实际运行的软件资产,使积累的价值留在他们能够掌控并从中获益的应用中。[11]

这一设计有四项实际目标:

- ▶ **可控的适应能力。**模型协助选择下一步,代码负责执行应用规则。
- ▶ **可复用的组件。**完整流程可以通过统一接口嵌入更大的应用。
- ▶ **聚焦的模型上下文。**每个组件选择其模型真正需要的信息。
- ▶ **专业经验归创建者所有。**创建者保有可复用的方法,能够改进、分享,并将其作为产品或服务提供。

Lyquid Flow 是 Lyquor 的一项产品计划,旨在依托其基础设施,将这些理念落实到应用开发中。我们此前的论文 *Lyte Quorum* 介绍了底层服务架构。[12] 本文重点介绍应用的组织与执行方式:如何将代码、模型与人工输入结合起来,形成易于理解、能够复用且归创建者所有的流程。

将领域经验转化为可复用的产品

Flow 让他人能够直接使用专家的工作方法,无需专家亲自指导每一次模型交互。一个负责按客户订单分配供货资源的团队,可以将理解需求、比较供应商的供货方案和处理异常的标准操作流程(SOP)转化为可复用的 Flow。由此形成的应用既可以支持团队自身的运营,也可以作为产品或服务提供给他组织。

更准确地选择判断依据、明确决策标准、加强结果校验,以及减少不必要的模型调用,都可以改善应用效果。构建者可以评估这些改进并纳入版本管理,在模型升级后继续保留和使用这些成果。

构建者可以编写代码、使用可视化编辑器,或借助 AI 起草并由专家审阅来创建 Flow。这三种方式都将使用同一个可复用组件库,让不同技术背景的人能够共同参与同一应用的构建。

创建者可以以开源或闭源代码的形式提供 Flow,也可以将其作为自行运营的服务。客户因此可以选择将代码集成到自己的应用中,或付费使用由创建者维护的服务。

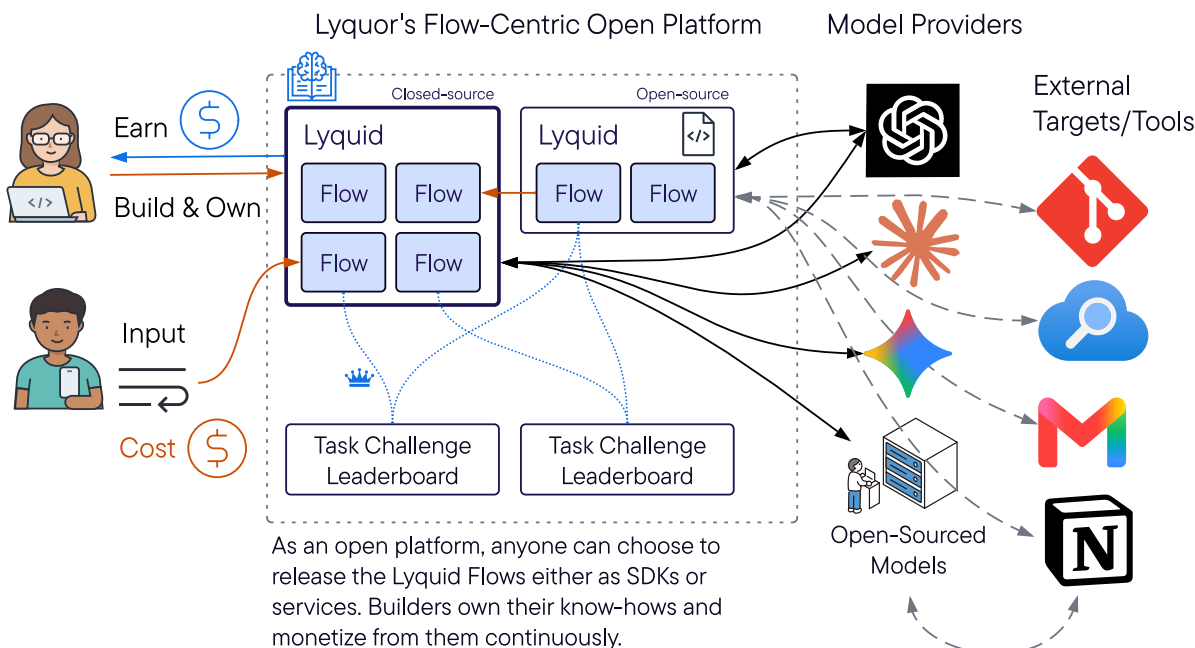


图2 开放的 Flow 应用生态

一个采购应用可以组合某位专家提供的供应商匹配 Flow 与另一位专家提供的资源分配 Flow,使用专有模型或本地部署的模型,并连接外部工具。每位专家提供一项可复用的能力,由应用构建者决定如何将其组合成完整产品。

通过解决许多应用共有的问题,专家能够触达更广泛的市场。将需求与合适的资源匹配,就是可能反复出现在数十乃至数百个应用中的核心任务。针对这一任务的更优 Flow 可以改进许多产品,其创建者也可随着 Flow 被采用和复用而获得收益。

任务排行榜可以把这些需求集中呈现出来,让构建者围绕同一任务比较方案、持续改进。Lyquor、社区成员和商业伙伴可以围绕真实业务需求或上述共性瓶颈发起挑战。每个排行榜都按任务表现对可复用的 Flow 排名,模型和技能的选择则由构建者自行决定。例如,供应商匹配挑战可以评估匹配结果是否满足条件、能否正确识别信息缺口,同时考察模型成本与响应时间。客户可以在自己的应用中评估有潜力的 Flow,再取得代码使用许可或付费使用服务,让任务表现转化为实际采用的依据,并为创建者带来收益。

保护数据与专有诀窍

企业在采用外部模型或专家提供的组件时,既要保护内部数据,也要保护自身的业务方法。Flow 将这套方法组织为 Step(步骤),即比较供货方案、请求审批等独立的工作单元。应用代码控制交给模型的任务、模型接收的信息,以及模型答案的使用方式。

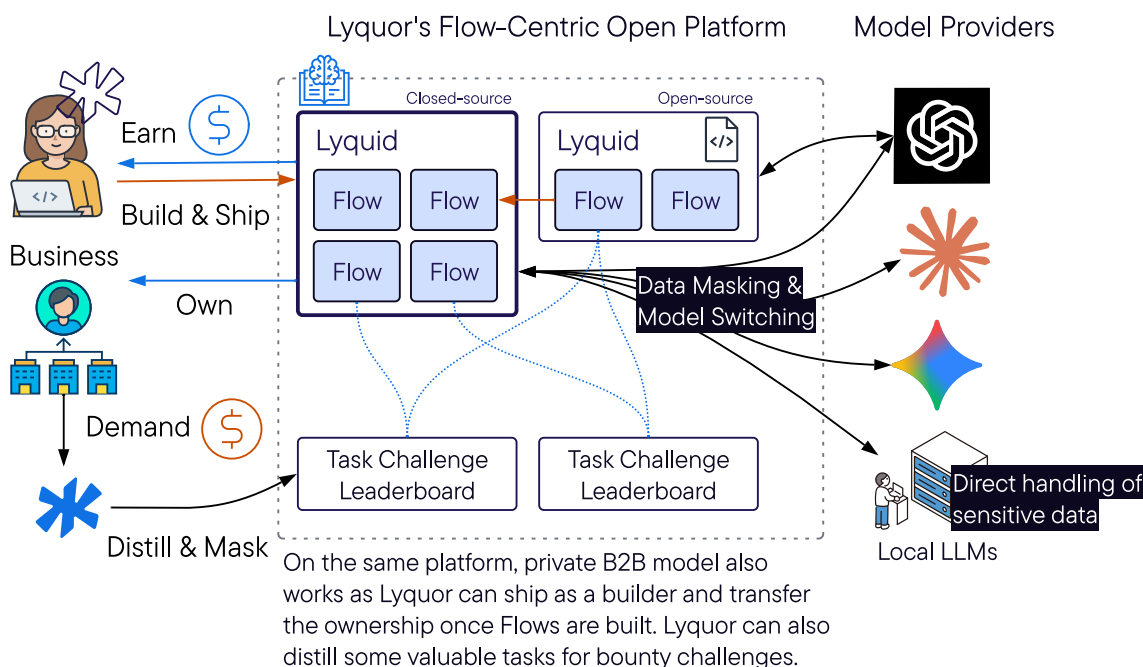


图3 企业私有部署

有些任务难在求解,却不难表述。某项任务可能需要强大的模型,但只需要一个简单的 Step 和一个范围明确的问题,因此不必暴露整套方法的太多细节。反过来,一个 Flow 也可以通过巧妙地拆解问题、准备上下文和组合结果,承载丰富的领域诀窍,使较小的模型同样发挥良好作用。^[2] 以下三项控制手段,让构建者能够根据每项任务的需要,选择合适的模型能力,并控制对外提供多少信息:

动态模型切换与混淆。应用代码可以随着工作推进,依据每项任务的推理需求和披露限制,选择兼容的模型。将调用分散到不同服务商,还可以使任何一家服务商所能看到的执行历史都不完整,增加还原整体方法的难度。每次调用都可以只交给模型一项范围明确的任务,省略或替换内部名称及无关的流程细节。保护效果取决于每次调用暴露了整体方法和执行历史中的多少信息。

由 Step 选择上下文并脱敏。每个使用模型的 Step 都为自己的任务选择上下文模板。应用代码可以在调用服务商托管的模型之前,对组装后的整个上下文进行脱敏,包括任务数据、外层 Flow 的指令及相关历史。发给模型的提示词可以依据获准对外提供的事实信息构建;必要时,可先由本地模型从敏感原始材料中提取相关信息。这有些像讲述一则寓言:改变或去除敏感的现实细节,同时保留解决问题所需的关系与约束。例如,比较供应商方案的 Step 可以发送匿名化的订单要求和供应商能力,同时将身份、议定价格和内部优先级规则留在企业内部。^[11]

用本地模型处理敏感工作。需要接触原始敏感数据的操作,可以交给本地或私有部署的大语言模型(LLM)。它可以 从内部记录中提取事实,再仅将获准共享、经过脱敏的结果交给服务商托管的模型做深入分析。范围明确的任务也能让 较小或开源的模型发挥作用:Flow 提供相关依据、安排执行顺序,并落实应用的校验规则。构建者可以将能力更强、成本 更高的模型留给真正能从中受益的任务。

这些控制手段共同帮助构建者在利用模型能力的同时,限制自身数据与核心诀窍的披露。实际的保护边界取决于 上下文选择、部署方式、日志记录及服务商政策。

这种私有部署方式,也让企业在与专家合作时保有相应的控制权。应用归企业所有,共享哪些信息也由企业决定; 专家则可以提供或维护其中的独立组件。为了引入改进,企业可以在审核并脱敏后,发布围绕某个具体问题的任务挑战, 同时将完整流程和原始记录保留在内部。

供应资源分配示例

在订单履约应用中,供给可能来自不同卖家,而各家的商品目录、库存数据和交付条款对供给的描述方式并不统一。核 心问题是先为每个订单找到符合要求的供给,再在不同订单之间分配有限的库存与产能。有些事实十分明确,例如数量 和价格;另一些则需要理解与判断,例如某种替代品是否可以接受,或某项交付承诺是否满足买方需求。模型帮助理解 这些形式各异的描述,将其转化为可以相互比较的数据,以及优化算法能够处理的约束条件。Flow 将理解信息、澄清疑 问和资源分配串成完整流程,在获得新依据时重新评估判断,并用明确规则限制企业可以作出哪些业务承诺。

准备可比较的输入。代码读取库存数据并统一计量单位;本地模型可以从消息、产品描述和供应商备注中提取要 求与条件。信息缺失或相互冲突的情况会被明确标示。Flow 只向外部模型发送经过筛选和脱敏的信息作为判断依据, 原始记录和专有分配规则则留在企业内部。

理解条件并寻求澄清。可以向模型提出这样的问题:“需求 R 要求最迟周五整批到货。方案 A 承诺周五发货;方 案 B 承诺周五送达,但允许分批交付。要判断任一方案是否符合条件,还需要澄清哪些问题?”Flow 可以向供应商或买 方请求补充信息,再据此重新评估这些方案。

校验并分配。关键不确定性得到澄清后,代码或约束求解器根据数量、产能、允许的交付时间范围,以及企业的成本 和优先级规则,计算各订单的分配方案。在预留库存或下单之前,应用代码会重新检查可用性,并落实审批要求。

当团队将自有 GPU 算力分配给内部 AI 应用与服务时,同样的匹配问题也会出现。需求可能以任务描述、示例和服 务目标的形式提出,其中对结果质量和响应延迟的要求,往往还需要进一步理解和澄清。同一需求可能由不同能力层级 的模型满足,而这些模型对 GPU 类型、显存和算力容量的要求各不相同。Flow 可以澄清用户对质量和延迟的需求,识 别合适的模型,再将其计算需求与可用 GPU 匹配。代码则检查兼容性、可用性、预算和访问权限。需求或容量变化时, Flow 可以重新考虑模型选择和 GPU 分配。

可复用的资产是整套方法:如何收集依据、理解条件、消除不确定性、分配资源,以及审批相应的业务承诺。模型、 代码和人在不同环节发挥作用。统一接口让构建者能够将这些环节组合起来,形成可以整体复用的流程。

每轮执行的统一约定

Flow 负责组织和协调称为 **Step** 的工作单元。Step 可以执行代码、调用模型,也可以调用另一个 Flow。统一的执行约定 规定了组件如何请求输入、如何返回待确认的结果。应用因此可以连接不同作者提供的组件,而不必依赖各组件的内部 结构。

Flow 可以分支、重新进入某个 Step,也可以包含带有自身循环的内层 Flow。例如,内层的供应商匹配 Flow 可以在 澄清与比较之间反复迭代,而外层的分配 Flow 则随着产能或优先级变化,再次执行匹配。每次会话运行都对应一个 **Scope(会话运行上下文)**,在上述可分支、可嵌套的结构中按先后顺序逐轮推进。即使流程包含嵌套循环,也仍然具有清 晰、可检查的执行历史。

轮次协议将一轮执行分为三个阶段:请求尚缺的输入、形成执行提案,再按应用规则处理结果。Step 只需使用其中 必要的阶段:

1. **输入(Input)。**Step 声明自己需要哪些信息,以及这些信息应采用什么形式。人或外部系统都可以提供输入。应用 可以在明确的等待点暂停,请求补充依据或等待审批,而不必让模型自行补全缺口。
2. **模型上下文与执行提案(Model context and proposal)。**代码选择相关依据和指令,模型据此提出结果及下一步行 动建议,形成执行提案。对于可由明确规则处理的操作,代码可以直接生成提案,从而省去不必要的模型调用。

3. **最终处理 (Finalization)**。Step 的代码会检查或修订提案, 按照相应的领域规则完成该 Step 的工作, 并汇总结果。随后, 所属 Flow 校验提案指定的执行目标和各项 State 更新, 再提交本轮结果, 使相应的执行进度与状态变更生效。

要检查执行过程, 既需要知道当前状态, 也需要知道它是如何一步步形成的。在供应资源分配应用中, **State (状态)** 保存数量、候选供应商、交付条件、待确认的分配方案和审批状态。**Trace (执行轨迹)** 记录已完成的轮次, 以及 Flow 接受的 State 变更。结合这两类信息, 就能区分尚待确认的分配方案和已经批准的业务承诺, 并追溯相应的决策过程。

Scope 将 State、Trace、当前 Step、正在等待的输入, 以及嵌套 Flow 的进度保存在一起。完整保存一次运行, 就可以在检查或复制分配过程时, 保留各个子流程已经执行到的位置。

Advance (推进) 提案用于请求进入另一个 Step 或结束执行, 并附带建议的 State 更新。Flow 校验执行目标, 以及更新值是否符合相应字段声明的数据类型。通过校验后, 执行进度和 State 变更一并提交, 并将这一轮写入 Trace。无效提案不会改变当前运行的 Scope。[11]

Step 也可能操作外部系统, 例如预留库存或下单。应用必须在操作前落实审批, 并处理重试, 避免意外重复执行。拒绝后续提案会使当前运行的 Scope 保持不变, 但不会撤销已在其他系统中完成的操作。

用可复用的 Flow 构建应用

一个可复用的供应商匹配 Flow 应能自行管理内部 Step, 并向调用它的应用提供所需的结果。因此, 一个完整的 Flow 可以作为另一个 Flow 中的一个 Step。外层 Flow 是**调用方**: 它接收内层 Flow 的结果, 并决定接下来做什么。

例如, 供应商匹配 Flow 可以理解需求、比较候选方案、寻求澄清, 并最终返回 **matched** (匹配成功)、**needs_information** (需要补充信息) 或 **escalate** (转交进一步处理)。采购应用收到 **escalate** 后, 可以转到一个请采购人员作出决策的 Step; 生产计划应用收到同样的结果, 则可以转到一个请计划人员调整排程的 Step。如果专家改进了 Flow 区分发货日期与到货承诺的方式, 这两个应用都可以采用这一改进, 而不必重写各自的审批或转交处理逻辑。

内层 Flow 在自己的 Scope 中执行各个中间步骤, 并读取调用方提供的上下文。要复用一個组件, 应用必须满足它对输入和 State 字段的要求, 由应用代码提供或适配相应数据。调用方根据返回的结果名称选择自己的下一个 Step, 并在推进流程前校验建议的 State 更新。组件之间通过结果名称衔接, 无需引用对方内部的 Step 名称。因此, 专家可以改进内部实现, 同时保持现有应用使用的接口不变。[11]

企业在复用某套方法时, 也可能需要加入自身的规则。**包装器 (wrapper)** 可以在不改变接口的前提下, 在现有 Step 的执行前后增加处理逻辑。例如, 先将待确认的分配方案送入审批 Step, 之后再由另一个 Step 预留库存。包装器也可以增加用于诊断问题的日志。这样, 调用方就能调整这套方法在自身应用中的使用方式, 而无需重写其内部逻辑。

上下文成为程序的一部分

可复用的 Step 需要明确规定如何获取模型所需的信息。比较供应商方案, 可能需要订单要求和供应商能力; 审查分配方案, 可能需要待确认的计划、剩余容量、交付日期和审批条件。如果每个 Step 都接收同一份不断增长的对话记录, 每次模型调用都要从大量无关历史中筛出所需的判断依据。[3]

决定选用哪些信息, 本身就是专家工作方法的一部分。Flow 将这种选择写进可执行逻辑: 每个使用模型的 Step 都会选择上下文模板, 从 State 与 Trace 中选取相关内容, 并判断这次任务是否值得调用模型。此前轮次中已被采纳的结果可以影响这些选择, 使流程能够在构建者设定的规则下调整。

模型看到的上下文, 是从本次运行的信息中按任务需要选取的内容, 与完整执行记录分开管理。审阅者可以查看某个 Step 采用了哪些字段和先前决策, 再调整选取范围, 而不必改变 Flow 其余部分的连接方式。完整历史仍可保留用于排查问题, 无需每次调用模型都全部发送。

嵌套结构还可以减少对背景信息的重复处理。模型的上下文按嵌套层级组装, 从最外层 Flow 逐层延伸到当前执行的 Step。内层 Flow 可以读取外层 Scope, 却不能修改它们, 因此内层执行时, 外层上下文能够保持稳定。如果连续调用的提示词都以这些不变的外层内容开头, 支持提示词缓存的模型运行环境就可以复用对这前缀内容的计算结果。[5] 成本与延迟是否改善、改善多少, 取决于工作负载, 仍需通过实测确定。[11]

可检查、可分支的运行过程

将执行过程划分为可以逐一识别的 Step, 就能分析这套方法在各个环节的表现。Trace 记录经过 Flow 确认的执行路径和 State 变更。如果同时记录每个 Step 的耗时与模型 token 用量, 构建者就能识别哪些 Step 消耗最多时间或 token、哪些被反复执行, 以及哪些是几乎每次运行都会经过的热点。即使只是对一个高频 Step 的小幅改进, 也能让许多次运行受益。

对于结果不理想或失败的运行, 还可以进行复盘。Trace 展示经过 Flow 确认的执行路径; 额外记录的提示词、模型响应、调用尝试与错误, 则有助于解释某个 Step 内部发生了什么, 或为何未能完成。构建者可以据此识别缺失的上下文、调整 Step 的提示词模板, 或修正决策规则, 再在后续运行中评估改动。

当多种分配方案都可行时, 企业需要比较它们, 同时避免重做已完成的工作。Flow 通过复制 Scope 支持这一点, 复制内容也包括嵌套 Flow 的进度及其正在等待的输入。各个副本随后可以独立继续执行。

一个分支可以评估使用备用库存, 另一个分支则可以评估刚刚澄清条件的供应商方案。两个分支都从已记录的同一组需求及资源可用性假设出发, 随后接收各自的输入并独立推进。这样就能比较不同分配策略, 而无需从对话记录中重新拼出先前的决策过程。

当某种思路无助于推进任务时, 运行也可以回到先前保存的位置。应用决定允许使用哪些快照; **Restore (恢复)** 提案则指定其中一个属于同一 Flow 的快照。提案通过校验后, 当前运行从该快照的副本继续执行。恢复的是快照中记录的执行进度; 已经发生的外部操作, 其恢复需要单独处理, 新的模型调用也可能给出不同答案。被放弃的分支与恢复事件如何呈现在审计历史中, 仍有待设计。^[11]

应用背后的基础设施

Flow 应用可能会调用部署在不同主机上的企业组件、模型服务, 以及专家提供的方法。这些服务协同工作时, 应用需要保留处理中的数据、协调执行进度, 并决定接纳哪些外部结果。Lyquor 的服务网络由独立运营的节点组成, 各节点负责托管和运行服务。其服务环境采用定制的 WebAssembly 运行时, 支持应用状态的持久保存与跨服务的协调执行。以下四项基础设施能力支撑这种部署方式:

- ▶ **按需托管与一致性协调。**运营方可以专注于自身支持的应用和硬件, 所托管的服务仍能与其他服务协同运行。Fate-Constrained Ordering (FCO) 将共享状态更新的排序与实际执行分离。每台主机可以选择托管哪些服务, 无需运行平台上的所有应用; 相互协作的服务对共享更新的先后顺序仍保持一致。
- ▶ **持久化的应用内存。**资源分配服务可以保留需求、供货方案和中间结果, 无需在每次调用时从序列化记录中重新还原数据。直接内存架构 (Direct Memory Architecture, DMA) 让程序中的原生数据结构在多次调用之间保持可用, 由运行时负责内存页的加载与保存。同一编程模型既支持服务共享的状态, 也支持各节点独立保存的本地工作数据。
- ▶ **可编程的分布式执行。**构建者可以选择合适的节点承担专业计算任务或提供模型服务, 也可以让多个节点执行同一任务, 实现冗余。通用过程调用 (Universal Procedure Call, UPC) 允许应用代码选择由哪些节点执行任务, 并规定如何校验或合并返回结果。应用可以采用最先返回且符合要求的答案, 也可以比较多个结果, 或等待足够多的响应, 以满足应用规则。
- ▶ **可验证的服务调用审批。**应用可以制定明确且可验证的规则, 决定是否接纳模型结果、API 数据或分布式计算结果。通过认证调用 (Certified Call), 应用可以要求外部观测或计算结果先经多个节点自动审核并联合背书, 再用于修改共享状态。按配置组成的节点委员会可以借助模型、Flow 或数据源, 依据应用制定的规则评估待执行的调用。达到规定的批准阈值后, 生成的证书会将获准的操作、输入与指定的调用目标绑定。接收服务在执行前会核验委员会对该次调用的背书, 并拒绝重放请求。

此前的 *Lyte Quorum* 论文对 FCO、DMA 和 UPC 作了深入讨论。^[12]

在资源分配应用中, 这些能力可以协同完成以下工作: 收集并比较服务商返回的结果, 保存处理记录, 再将选定的分配方案提交审批, 按既定策略决定是否批准, 并协调多个服务共同执行。Flow 定义如何组织这些工作的可复用方法; 基础设施负责持久保存状态、分布式执行任务, 并支持验证相关更新是否获准。

与现有方法的比较

Lyquid Flow 将专家的工作方法封装成软件组件,使创建者能够拥有并掌控这套方法。创建者决定采用哪些依据、如何调用模型,以及如何根据结果安排后续工作。封装后的方法可以进行版本管理、部署,并供不同应用复用。AI 编程工具可以帮助编写 Step 的代码;这些 Step 在运行时又可以调用专用模型或外部服务。

G5:以自然语言描述作为源代码。G5 将用结构化自然语言描述的程序意图作为源代码,由 AI 据此生成应用代码;无论意图描述还是代码发生变化,都会保持两者同步。^[6]它主张软件归用户所有,也主张用户能自由更换辅助编程的 AI 模型。^[7]G5 关注软件如何编写和维护。Flow 关注的则是应用如何运行:选择什么上下文、怎样使用模型,以及根据结果继续执行哪些工作,并将这套运行方法封装为可复用的组件。这类工具也可帮助编写和维护各个 Step 的代码。

Jev:为自主掌控的流程提供判断服务。Jev 以明确的数据类型返回选择结果、评分和概率,由应用代码将这些单项判断组合起来。^[15]需要模型作出哪些判断、向它提供什么信息,以及如何根据答案安排后续工作,仍取决于构建者的专业经验。TypeSafe 不将数据用于训练的政策^[16]回应了训练使用方面的顾虑;判断引擎本身仍由服务商提供。Flow 将组织这些判断的工作方法封装为构建者掌控的组件,Jev 则可以在其中的 Step 内提供具体判断。

LangGraph 与 workflow 框架:组件间的统一执行约定。LangGraph 提供执行路径控制(路由)、子图和检查点,提示词和整体架构由应用代码定义。^{[8][9][10]}AWS Step Functions 也支持嵌套 workflow。^[1]Flow 则统一规定组件如何提出执行提案、如何校验并提交结果,并用快照保存包含各层嵌套 Flow 在内的完整运行上下文。内层 Flow 返回以名称标识的执行结果,由调用方选择下一步。^[11]组件作者通过统一接口封装领域逻辑,应用仍可决定这些逻辑如何用于自身流程。

这些机制也可以用更底层的流程编排能力实现。Flow 将它们规定为不同作者独立开发组件时共同遵循的执行约定:专家可以改进组件内部的 Step,使用方则继续掌控自身应用的执行规则。复用的产品,是已经转化为可执行软件的专家方法;模型则是其中可按需选用的组件。

参考文献

- [1] Amazon Web Services. [Start workflow executions from a task state in Step Functions](#).
- [2] Anthropic. [Building effective agents](#).
- [3] Anthropic. [Effective context engineering for AI agents](#). September 29, 2025.
- [4] Anthropic. [Is my data used for model training?](#). March 16, 2026.
- [5] Anthropic. [Prompt caching](#).
- [6] G5 Labs. [G5 Labs Raises \\$14M Seed Round to Pioneer the Post-Developer Era](#). September 15, 2026.
- [7] G5 Labs. [Platform and use cases](#).
- [8] LangChain. [Graph API overview](#).
- [9] LangChain. [LangGraph overview](#).
- [10] LangChain. [Persistence](#).
- [11] Lyquor Labs. Lyquid Flow Design. 未公开的技术设计规范;暂不披露内容。
- [12] Hao Hao, Dahlia Malkhi, Maofan Yin, and Lizan Zhou. [Lyte Quorum: Off-Chain Ready Smart Contract Hosted with Choice](#). arXiv:2509.23448, 2025.
- [13] Davide Castelvecchi. [Who gets credit in the AI era? OpenAI maths bombshell sparks debate](#). Nature, September 17, 2026.
- [14] OpenAI. [How your data is used to improve model performance](#). Updated March 13, 2026.
- [15] TypeSafe AI. [Introduction](#).
- [16] TypeSafe AI. [Models and data handling](#).