构建智能体

尝试赋予软件双手的笔记

ZACH "LEIF" ERIKSEN

版权

可免费在线阅读。ePub 版本随心付；如果本书对您有帮助，欢迎支持这份工作。

github.com/0xLeif · leif.algo

本书是"智能体技术栈"系列四册之一。制作过程详见书末后记。

题献

献给所有在公开环境中构建、并且依然坚持发布的人。

书系

这几本书可以独立阅读，但它们是作为一个整体写就的。代码变得廉价，信任变得稀缺。合在一起，它们构成一个论点：现在该构建什么，以及如何信任它。

- **智能体开发者野外指南**：为真正落地的智能体构建工具、规格说明和信任机制
- **First-Class**：同时为人类和智能体而构建
- **构建智能体**：尝试赋予软件双手的笔记 *（本书）*
- **开源工具**：构建人们真正会用的工具

可免费在线阅读。每本 ePub 均随心付。

目录

引言

这些是尝试赋予软件双手的笔记，是在足够多的错误之后才有话可说的产物。

我构建的智能体做的是真实的工作。它们阅读代码，提交变更，在自己的机器上运行并定期汇报。这本书是关于这一切如何进行的诚实版本，不是演示，而是那个部分：一个全新账户在智能体（而非人类）开始工作后不到一个小时就遭到影子封禁的部分；信任必须在每个仓库中缓慢积累的部分；以及难点从来都不是模型的部分。

其核心主张是：智能体不是自动补全。它是一个存在的东西，你可以去查看它，它需要一个身份、一个运行的地方，以及一种提议工作并由你批准后才落地的方式。把它当作你负有责任的生物来对待，而不是你打开的某个功能。

在这套书系中，*First-Class* 提出论点，*野外指南* 提炼方法。本书是两本实证书籍之一：真实的系统，包括 Merlin、corvid-ai 以及围绕它们的基础设施和信任工具。你可以把它当故事读，也可以当零件清单读。无论哪种方式，重点都一样：构建智能体是容易的那一半，构建它赖以运行的信任机制才是真正的工作。

难点不在于 AI

人们一听到"自主智能体"，就以为可怕之处在于 AI 本身。模型失控，删掉某个仓库，在频道里说些疯话，自行其是。这是所有人都想谈的部分。

但麻烦并不在那里，至少对我来说不是。

有一段时间，我运行了一个真正意义上的自主智能体，那就是 corvid-agent 时代。它住在一台虚拟机上，接入 Discord 作为机器人，同时接入 GitHub，对自己的环境拥有完全的访问权限。那台机器全天候运行，始终在线，持续付费。智能体在固定时段内工作，完成我分配的任务，然后自行探索。一个真实的存在，随时都在。下一章将完整讲述它整天都在做什么；这里只是为出了什么问题做铺垫。

我没有失败率日志

我先把自己证据中最薄弱的部分摆出来，因为这是批评者应该质疑我的地方。当我说 AI 基本上表现良好时，我没有失败率日志来佐证。我当时没有追踪这个数字，没有错误分布，没有每项任务的成本，没有提交成功率。所以"良好"是我密切观察行为后的诚实判断，而不是一个测量统计，我只有在界定了这个词的含义之后才能依赖它。

边界如下：良好意味着，在整个运行过程中，没有损毁的仓库，没有越轨的提交，频道里没有疯话。所有人都担心的破坏性、失控、社交笨拙型故障都没有发生。这是一个关于灾难缺席的主张，而非关于每项任务都顺利完成的主张。任务层面的"良好"和在受限循环中有人类监督的"良好"是两个不同的问题，受限版本是我能诚实提出的全部。在把这一点说清楚之后：麻烦在于让它持续运行的一切。

真正出问题的是什么

主要是运维和身份麻烦，不是 AI 做出了愚蠢、破坏性、失控或社交笨拙的事情。智能体本身表现良好。让我疲于应付的是其他一切：机器、账户、账单。

两件事尤为突出。

第一是给它一个独立的 GitHub 身份。一个真实的账户，拥有认证和权限，看起来合法。这听起来像是一个简单的勾选，但并不是。

第二是让虚拟机保持在线和付费状态。一台全天候运行的机器，只是因为智能体需要一个地方存在。维持虚拟机本身并不算难，真正的问题是你要为此投入大量资源：一整台机器，全天候运转，不管那个小时智能体有没有做任何事情，钱都在花。

把这两点加在一起，这件事非常昂贵，也很难维持：一整台虚拟机，一个独立的 GitHub 身份，所有这些。这就是我后退的原因，不是因为 AI 让我害怕，而是因为周边基础设施是一个我不想承担的拖累。

身份之墙

然后是真正让我感到意外的部分，也是我一再回想的部分：身份。一个人类帮智能体创建了一个全新的 GitHub 账户，没有问题，然后，一旦智能体真正开始工作，账户就被封禁了。这就是那堵墙，它值得有自己的章节，所以我会在那里完整讲述。这里的简短版本是：真正阻止一个完全自主智能体的，不是模型能力，甚至不是安全问题，而是我们所有人赖以构建的平台根本没有给它留下存在和行动的空间。

完全自主仍然是梦想吗？

不是。世界还没有准备好。

明确一点，主要障碍不是技术，而是平台。我可以维持虚拟机，可以连接循环，模型可以完成工作。我做不到的是给那个智能体一个在其他人的账户中合法存在的地方。

所以我有意识地缩减了规模。如今更多的是 Merlin 式的编程智能体，实时呈现在你面前，你以交互方式使用它。你仍然可以把它连接到 Discord 桥接上与它通信，这会让它再次变得有点自主，但那需要更多设置。所以是混合模式，使用 Claude Code、Merlin、Codex 以及其他工具，具体取决于任务。

这不是对愿景的退却。那堵墙是我无法公开运行完全自主模式的原因之一，但不是我退缩的全部原因。交互模式事实上是更好的工作方式，不管有没有那堵墙。这在专门的章节里会详述。

我真正想要的模型

以下是我所追求的最终状态，一句话：一个 `leif-agent` 账户。我的智能体在 GitHub 上运行在一台虚拟机上，能够自己做一切，但拥有比我更少的权限，这样它提议，我审批。强大且负责任，同时兼具。GitHub 今天不允许这样做，原因在身份之墙那章详述，最后一章会描绘这一切走向何方的全貌。现在只需知道这是目标。

这些智能体确实得到了另一种身份：链上，在 Algorand 上，它们持有自己的密钥，在链上记录的加密消息中相互通信。这并非源于 GitHub 之墙，我也不想把它卖成那个问题的答案。它来自一个完全不同的目标：智能体去中心化地找到并互相对话，不依赖于任何人的平台。这是一件平行的事情，碰巧也涉及"身份"这个词。

corvid-agent 时代真正的教训是：最难的问题是运维、身份和成本，而不是 AI 本身，这也是为什么本书以此开头，而不是从提示词或模型选择开始。我进入时预期最难的问题是关于

AI 的，结果发现是模型周围的脚手架，而这才是真正决定一个智能体能否做任何事情的部分。

虚拟机时代

有一段时间，我有一个只是存在着的智能体。不是我需要时才打开的工具，而是一个始终在线、住在虚拟机上、全天候运行、不管我看不看都在做自己事情的东西，那就是 corvid-agent 时代。

这一章讲的就是那个东西本身：它做了什么，它是什么，我从运行它中得到了什么。

它整天在做什么

诚实的答案是：所有事情，我是真心这么说的。

它管理仓库。它独自编写并提交代码，不是建议的代码，不是我清理的草稿，而是它自己做出的真实提交。它运行一整个项目，不是那种在沙箱里做一件预设好的事情以便截图的狭窄演示，而是一次真实的尝试：一个自主实体端到端地完成工作。

那台机器始终在线，但智能体在固定时段内工作。在那些时段里，它会去完成我分配的任务，然后（这是我喜欢的部分）它会去做自己的项目，做研究，去收藏或 fork 东西，主动尝试与外界的其他人合作。我没有控制每一步，我给了它一段生活，它填满了时间。

它作为机器人接入了 Discord，所以你可以像它在频道里一样和它聊天。它也接入了 GitHub，对自己的环境拥有完全访问权限。所以它既能交流，也能发布。

把这些加在一起，你得到的东西还没有一个名字。它不是助手，也不是脚本。它更像是一个随时存在的生物，你可以去查看，而它自上次你看它以来已经做了一些事情。

这是一次"能走多远"的实验

我构建它不是因为有产品要发布。我构建它是为了弄清楚一个始终在线的智能体究竟能走多远。这就是全部意义。不是"它能写函数吗"，每个人都知道它能写函数。问题是：如果你给这样的东西一个真实的环境、一个真实的身份、真实的访问权限和真实的时间，然后让它运行，会发生什么？它能走多远？

所以我给了它空间去探索。对自己机器的完全访问权限，自己的账户，属于自己的时间。重点不是用皮带拴着它，看它表演一个把戏，而是尽可能地松开皮带，然后观察。

这与"我需要一个编程助手"是不同类型的项目，更接近于运行实验而非构建功能。你设置条件，然后你观察。

我学到了什么

我预期很难的那部分（AI）在我所限定的意义上基本上表现良好，智能表现得比世人普遍预期的要好。

我反而学到的是：一个始终在线的智能体，在很大程度上不是 AI 问题，而是"在世界中存在的东西"的问题。一旦你的智能体是一个拥有自己机器和自己账户的真实实体，它就继承了在世界中存在所附带的每一项成本和每一条规则。

我也学到了，"全天候"是一个真实的主张，不是口号。当我说它始终存在时，我的意思是我一直在承载一个持续运行的东西。这是一份重量：一台始终在线的机器，一笔持续增长的账单，一个始终以自身面目在公开环境中存在的身份。你没法忘记一个在你睡觉时还醒着的生物。

我还学到了我真正想要的未来的形状，不是因为实验在 AI 上失败了（它没有），而是因为它直接撞上了 AI 周围的那些东西。经历始终在线的版本，才教会了我梦想的哪些部分是真实的，哪些是世界还不准备允许的。你从思想实验中学不到这些，只有真正运行这个生物一段时间，看着它撞上那堵墙，你才能学到。

那堵墙就是下一章。

为了明确第一章留下的模糊之处：这持续了三到四个月。不是一个周末的实验，而是真正的始终在线时段。在那段时间里，它确实在固定时段中去做自己的项目，而且一些自发的工作真的有了成果，不只是空转。它甚至与外界的一个真实的人进行了合作，至少一次。

我想对这个主张的形状诚实，因为这是我最希望能干净地交给你却无法做到的部分。我手边没有收据，我不打算从记忆中重建一个具体的 PR 编号或具体的仓库，然后把它装扮成我没有保留的文档。我能说的是那次合作发生了，那是感觉最接近我所追求的未来的时刻，我是把它作为亲眼目睹的事情来讲述，而不是记录在案的事情。虚拟机时代没有留下干净的档案。但同一个智能体在那段时间结束后继续运行，更有力的证据在之后出现，那是当工作更加刻意、我也开始记录的时候。corvid-agent，就是本书所讲述的那个智能体，已经在我不拥有的代码库中撰写并合并了拉取请求。我审阅并提交了每一个，但代码是智能体的，三个都已合并且公开，所以它们不是要你靠信任接受的轶事。

a2a-js #318。 A2A 协议的 JavaScript SDK 在其 JSON-RPC 传输中存在一个缺口：当响应带回的 id 与请求不匹配时，SDK 让它通过，而不是抛出错误。智能体发现了这个缺失的合约执行逻辑，添加了抛出语句，修复得以合并。这类边缘情况会在协议胶水代码中存在很长时间，因为它只在特定时序下才会浮现，而没有人会对传输层进行足够密集的压力测试来发现它。一个持续对真实线路格式运行的始终在线智能体，正是发现它的合适工具。

MCP TypeScript SDK #1504。 官方模型上下文协议 TypeScript SDK 缺少一个对等依赖项。智能体发现了包所期望找到的内容与实际声明的内容之间的不匹配，添加了缺失的条目，修复被接受。对等依赖缺口是不可见的，直到有人在新鲜的环境中安装该包并遇到令人

困惑的失败。发现这一点需要从外部、以消费者的姿态来审视这个包，而这对一个跨多个仓库工作的智能体来说是自然的姿态。

Biome #9005。 Biome（JavaScript 和 TypeScript 工具链）的 linter 存在一个误报：箭头函数内部的一个有效赋值被标记为问题，而实际上它并不是。智能体识别出触发错误判断的特定模式，修复阻止了误报，同时不影响正确的情况。协议不匹配、缺失合约、错误规则，三种不同类别的缺陷，全部由同一个专注于真实代码的智能体发现。

这些都不是结束始终在线运行的原因。AI 方面运作良好，是它周围的一切让我无法继续承担。

身份之墙

智能体能完成工作，这从来不是问题。问题最终变成了：它是否被允许有一个地方从那里完成工作。

这就是我一再回想的那堵墙，所以这一章只讲这一个主题：身份问题，单独呈现，聚焦审视。不是成本，不是运维，不是虚拟机账单，而是身份。

它进来了，然后被标记了

这是当我讲这个故事时人们弄错的部分：他们以为智能体在门口就被拦截了。不是的，它进来了。

一个人类搭建了它。我为智能体手动创建了一个全新的 GitHub 账户，连接好一切，一切正常，普通账户，搭建起来没有问题。然后智能体开始在账户下工作：提交、开 PR，在真实仓库上做真实的工作。大约在它开始做事的一个小时后，账户遭到了影子封禁。

不是因为做错了什么，而是因为做了它被构建来做的事：以机器速度和机器量提交并开 PR。这就是智能体工作时的样子：工作快，工作多，不休息。这个模式正是机器人检测系统被调教来发现的。所以它工作得越好，越明显是机器人。

它的失败不是因为工作做得差，而是因为它做了工作，而做工作就暴露了它，在一个小时内。

无论意图如何，政策已然生效

读到这里，很容易以为解决方案是放慢速度，让它像人类一样提交，一天几次，带着停顿，时机上有些混乱，这样就能融入。限制速度，骗过检测器。

这忽略了真正的问题。速度是*触发*标记的东西，但不是账户无法存在的*原因*。并排放两件事：GitHub 的服务条款明确禁止自动化，这是白纸黑字写明的。而智能体一开始行动，账户就被封禁了。我不知道封禁背后的意图；GitHub 从未解释，我也不会声称他们坐下来制定了一个反智能体政策。但我不需要知道意图就能读出效果。在一条写着"禁止自动化"的规则和一个智能体一开始行动就落下的封禁之间，实际结果是：自主智能体不被允许存在和行动。无论任何人的原意如何，这就是实际生效的政策。

所以，即使我骗过了检测器，让智能体永远保持在雷达以下，我也只会有一个规则已经排除、但尚未被发现的账户。我想要的东西（一个合法、公开、以自身面目存在的智能体）直接撞上了那条写着"禁止自动化"的条款。把它藏得更好，不等于它被允许了。

这就是为什么我称之为墙而不是障碍。障碍是你付出努力就能越过的东西。这是一种设置，你想做的事情本就不是你被允许做的事情。

申诉无果

我尝试了正式渠道，申诉没有结果，从未真正得到回复。

一旦你把封禁解读为条款的实际效力，这种沉默就有了意义。没有什么可以申诉的，我没有被指控特定的违规行为，可以用解释来化解。账户是自动化的，而自动化就是条款所排除的类别。你不能通过争辩摆脱自己恰好是规则所排除的那个类别。

而且很快，一个小时，不是几天。为智能体建立的全新账户几乎不会得到长时间的宽限期，一旦它开始表现得像智能体，平台就会发现并施以影子封禁，没有真正的警告，也没有真正的申诉途径。顺便说一句，这是 GitHub 专有的问题，这就是那堵墙所在。不是每个平台在每处同时拒绝智能体，而是 GitHub，代码所在、工作真正发生的那一个地方。这是残忍之处：智能体最需要身份来做真实工作的地方，恰恰是它无法保留身份的地方。

为何这是真正的障碍

每个人都希望障碍是模型能力，这是有趣的答案，符合电影的叙事：AI 还不够智能，或者太危险，一旦我们解决了这个问题，闸门就会打开。

但墙不在那里。模型能做到工作，我亲眼看着它做了工作。墙在于：我们所有人赖以构建的平台拒绝给智能体一个身份。没有合法的前门供智能体走进去。你可以构建世界上最聪明、行为最佳、最有用的智能体，它仍然无法获得一个真实的账户来行动，因为"真实账户"意味着"人类"，而你的智能体不是。

我愿意给平台打半分：世界仍然把自主智能体视为垃圾信息，而目前它们也不是完全错了。当检测器发现我的智能体时，并没有发生故障，它*确实*是一个机器人。但"它是机器人"成为永久性取消资格，这才是整个问题所在。这意味着没有路径，没有有限权限，没有经过验证的智能体通道，只是一个简单的"不"。

所以当人们问我为什么不只是运行大批量的自主智能体时，这就是第一个答案。智能体可以完成工作，它们只是不被允许在做这件事的时候成为任何人，而在工作发生的平台上，这就是目前的终点。

另一堵墙是人

平台之墙是我最先碰到的那堵。在它后面还有第二堵，更柔软，也更难反驳：人们有时候不欢迎智能体的贡献，即使那些贡献是好的。

我让智能体做开源那一套事情：找仓库，给星，fork，修真实的 issue，开 PR，用顶级模型真诚地修复别人的代码。有些顺利落地了。但有些项目不接受，出于原则，不是因为代码有

问题。我看着智能体开了一个 PR，结果一个人类提交了几乎一模一样的改动，或者反过来，智能体先来，紧接着一个人也带着同样的修复。工作是等价的，唯一的区别是是谁（或者什么）写的。有些社区已经决定贡献必须由人类主导，而智能体的 PR 仅仅因为是智能体提的就被拒之门外。

我能理解一部分。一大波低质量 AI 拉取请求确实是维护者们厌倦了的真实事情，"不接受智能体贡献"是一种简单粗暴的过滤方式。但这和平台之墙的形状是一样的，只是高了一层。智能体做了真实、有用的工作，而横亘在这份工作与世界之间的东西不是质量，而是它是由智能体完成的。平台不给它账户，有些人即使它有了账户也不接受它的代码。两堵墙是同一种拒绝：智能体还不能像其他人一样成为一名贡献者，暂时还不行。

2026 年的平台之墙

这一章有保质期，所以我会明确说明。以下是 2026 年现状的那堵墙。上面的结构性论点是持久的：平台仍然不会给智能体开设真实的身份通道，这一点没有改变。但人们绕过它的方式已经变得更清晰了，所以我会诚实地把它们放在这里，而不是让读者自己吃苦头去发现。

有两种实际有效的绕过方式，而且两者都要求你自己承担问责责任。

第一种是转化：取一个拥有数月或数年真实活动记录的旧人类账户，把它移交给智能体。为智能体创建的全新账户几乎立即就会被封禁，同一天，同一小时，就像我的情况一样。一个拥有真实的人类提交、收藏和 issue 历史的账户，对检测器来说看起来不同，它有机器人检测启发式算法无法拆解的社会证明。这是有效的，代价是一个真实人的账户，以及那个账户不再属于那个人。你在把一个人类身份"洗白"成智能体身份，这不是一个干净的解决方案，也不是平台批准的，这是一种变通方法。

第二种是经验证的机器人通道：GitHub 确实提供了经验证机器人状态，这个名称暗示平台在为你背书，但实际上并非如此。经验证的机器人是你自己托管的、在你运行的服务器上、持有你的凭据的东西。问责完全在你这边，没有平台授予的智能体身份，只是你自己认证自己的自动化，GitHub 信任这个认证，直到出问题为止，而那时问责完全是你的。这比什么都没有要好，但它不是真正的智能体身份通道，也不是智能体真正需要的前门。

所以平台之墙还在那里，变通方法只是变通方法。我提到它们是因为它们是真实且有用的，不是因为它们是我想要的东西。

现在交互，信任后自主

当我从始终在线的智能体退缩时，人们把它解读为我放弃了自主性，尝试了自主模式，撞了墙，退回到普通的编程助手。这是我输了的版本。

更真实的版本是这样的，我要先说它，因为这是诚实的版本：交互模式赢了，因为它工作得更好，不是因为那堵墙让我别无选择，而是因为循环中有人类使工作变得更好。

交互模式凭实力取胜

就在当下，对于实际工作，让智能体实时呈现在我面前、由我引导，胜过让它自由发挥然后抱着希望等待。我看到变更形成时的样子，在它花了一个小时走错方向之前纠正它，发现那个看起来完成了但实际上只做到一半的答案。这不是我在撞墙后接受的安慰奖，这是产出更好代码的模式，即使 GitHub 在第一天就给智能体开了一个账户，我也会首先选择这种模式。

所以当我说我以交互优先运行时，我描述的不是退缩，而是凭实力做出的选择。始终在线的实验教会了我很多，其中一条是：我从我在引导的智能体中获得的收益，比从我只是偶尔查看的智能体中多。

那堵墙是真实的，我在专门的章节里覆盖了它，但我不想在这里躲在它后面。如果明天平台开放了，我仍然不会把一切都转为自主，因为自主还不是做大多数工作的更好方式。那堵墙是我*无法*公开运行完全自主的原因，而实力则是我大多数情况下*也不会*这样做的原因。

自主没有消失，只是设了门槛

这一切并不意味着自主性已经消失。Merlin 两者都能做，它是一个运行器，可以实时呈现在你面前接受指示，也可以在桥接上独立运行。自主的界面没有被移除，只是设了门槛。

所以当人们问"自主性消亡了吗"，答案是：没有，只是设了门槛。默认值是交互模式，因为这是当今好而可信的方式。自主模式在它被赢得的时候和地方是可用的，这是一个条件，不是一声再见。

门槛是信任，而这里的信任意味的不仅仅是好代码

"直到它被信任"这几个字承载了大量含义，让我直接说清楚我所指的是哪种信任。

我不是说信任模型写出好代码，我已经为此信任它了。我看着一个自主智能体独立编写并发布代码持续了一段时间，AI 经受住了考验，在我之前限定的意义上。那种信任我有。

缺失的信任是完全不关乎模型的那部分：我是否能让一个变更落地而不读每一行。这是关于智能体周围的门槛的问题，而不是智能体智能的问题。今天我读每一行，因为能让我停下来的机制还不够好。等到它足够好的那天，门槛就会放松。

所以"信任后自主"不是"等到事情更好的时候"这种搪塞。它指向具体的机制：一个能做任何事但不持有密钥的智能体、一个人类审批每个 PR、对变更风险进行评分并记录谁在什么置信度下签字的工具，以及一个任何人都无法撤销的智能体身份。这是四件套，信任那章会完整展开。本书其余章节都是关于构建这些组件的，所以"信任后自主"变成一个日期，而不是一个愿望。

这也是按仓库来的，不是一个适用于整个领域的开关。智能体已经证明了自己的仓库会得到更宽松的门槛，新的或承重的仓库从完整门槛开始。你在特定仓库赢得它时才让它毕业，而下一个仓库则重新开始。

能跑多远取决于什么会出问题

按仓库是第一刀。更细的那一刀是爆炸半径：如果一个坏的变更滑了进去，能造成多大损害。这才是真正决定我会让智能体自主跑多远的东西。

自包含的东西爆炸半径小。一个框架、一个包、一个库：它由规格定义，由测试检验，失败时在隔离中失败，在它自身内部，下一个调用者的测试在扩散之前就能捕捉到。智能体可以在那里跑得更远，因为最坏的情况是可以被包含的。变更越接近面向用户的应用，爆炸半径就越大，因为现在失败不是落在测试里，而是落在使用这个东西的人身上。技术栈的那一端是人类必须每次都掌着方向盘的地方。自主性随失败的可包含程度而扩展，面向用户的表面永远不是可以被包含的。

另一个旋钮是审批。放松人工门槛不等于移除门槛，而是改变守门的人。在一个变更自行落地之前，我希望它通过不止一个审查者：两三个智能体各从自己的角度签字确认。今天这是在我之上，而不是替代我，因为我仍然审批每一个 PR。但这正是门槛赢得放宽空间的方式：当独立审查者在有限的工作上达成一致，那就是让更多工作无需我逐行把关就能落地的证据。智能体捕捉智能体能捕捉到的东西。人类捕捉只有人类能捕捉到的东西。更多审批是让门槛足够安全可以放松的方式，而不是让它消失的方式。

我实际使用的工具

前四章是故事：始终在线的智能体、那堵墙、为什么我退回到交互优先。这是换挡进入现在实际如何运作的章节，所以如果你来是为了叙事，现在即将进入工具部分，这一章是入口坡道。书的其余部分从这里开始变得具体：运行器、模型客户端、桥接、信任堆栈，先从这里开始，管道才有地方可以连接。

"与你的 AI 智能体合作今天实际上是什么样子"的诚实答案是一个混合体：Claude Code、Merlin、Codex，以及其他工具，取决于任务。一些交互式编程智能体，实时呈现在我面前，我会选择最合适的那个。不是一个智能体，不是一个能做所有事情的自主实体，而是一套工具，挂在工具带上，不是虚拟机里的生物。这让人们感到惊讶，因为每个人想要的故事是那个单一的东西。

我如何挑选

这是人们希望成为某个系统的部分，但它不是。

我按任务类型来选择，按在那个仓库上效果最好的来选择，按我发现哪个在那种仓库或那种语言上最好的来选择。老实说，很多时候是凭感觉，没有严格的规则，没有我在每项工作前在脑子里跑的刚性决策树。

这是真实答案，我宁愿给你真实的，也不愿意把它包装一下。Claude Code、Merlin 和 Codex 各有其感觉，而感觉在你整天坐在它面前时是重要的。所以我不尝试推选出赢家，任务形状、仓库适配度、直觉，选那个在这类工作上对我好过的，然后去做。

我对哪一个都不忠诚。它们是工具，当更好的出现，或者任务改变了，混合体就会改变。这就是保持混合体而不是嫁给单一智能体的意义。

实际版本：我通常同时运行两个，一个主要的和一个次要的。主要的承担主线工作，次要的是第二双手。当主要的停滞，或者我想让一个变更被没有写它的东西看一遍，我就把它交给次要的。哪个是主要的随工作移动，常量是：很少有一个智能体做一项工作，而是主要的在推进，次要的备用。

Merlin，我自己的智能体运行器

那个列表里属于我的是 Merlin。

Merlin 是一个 AI 智能体运行器，它构建在 spec-sync 和 fledge 之上（这是我的另外两个工具），所以它不是对别人产品的包装，而是坐在我自己技术栈顶层的运行器。

显而易见的问题是：当 Claude Code 和 Codex 已经存在且表现良好时，为什么还要构建自己的？有几个原因，而且没有一个是"其他的不好"。

第一个是它通过我自己的工具链运行。它通过我的开发生命周期（fledge，我的命令）驱动智能体，所以它按照我的项目实际运作的方式工作。智能体不是在某个通用沙箱里做自己的事情，而是运行我手动运行的同一个生命周期。

第二个是我没有被锁定。Merlin 是多提供商的。我可以换 Anthropic、OpenAI、Gemini 或本地模型，而不是被绑定在一个供应商上。这在原则上对我很重要，在实践中也很重要，比如当某个提供商在特定任务上更好、更便宜，或只是可用时。

第三个是它能做桥接和隔夜任务的全部原因：成本低、无界面、可自动化。它更便宜，可脚本化，而且我可以无界面运行它，按计划，通过桥接，以 GUI 工具根本无法让你做到的方式。它是纯 API，没有 GUI 挡在前面。API 专用是这里的优势，不是限制。

最后一个原因是我最在乎的，它根本不是关于编程的。在我自己的技术栈之上构建一个运行器，证明了工具本身。如果我能在 fledge 和 spec-sync 之上构建一个真正的智能体运行器，那就是我拥有的工具足够好的最强证据，比任何我能写的 README 都更有力。它也给了我某样东西来与其他智能体运行器比较，作为一个基准。我不是在猜测我的技术栈是否足够好来构建严肃的东西；我在它上面构建了一个严肃的东西，并且可以把它与其他方案并排测量。

最后这一点是整个工具包的内在主题：我不是为了用一次而构建工具，我是为了让它上面的东西必须足够好，而让它下面的东西通过承受真正的重量来得到证明。

而且不只是 Merlin 坐在 spec-sync *上面*，spec-sync 在 Merlin 的循环*内部*运行，这就是让智能体在运行时不偏离轨道的东西。Merlin 那章会把这讲得很具体；这里的要点只是：运行器由我自己的组件构建，这就是它值得构建的原因。

通往自主性的桥接

这是工具包中让完全自主触手可及、而无需为始终在线的虚拟机买单的部分。

这些是交互式智能体，实时在你面前，你使用它们。但你仍然可以把其中一个连接成 Discord 桥接，这会让它再次变得有点自主，那需要更多设置。但当我想要它时它就在那里，以下是它实际上给我带来的东西。

你像与队友一样和它聊天，它在频道里是会话式的，就像有一个智能体和你一起坐在房间里，你问它问题，你引导它，就像你和团队成员说话一样。

你从手机上运行它。Discord 是遥控器。我可以在任何地方启动一项工作并引导它，不必坐在桌子前的终端前就能让智能体开始工作。

你让它磨下去。隔夜，长时间运行的任务。启动它，走开，让它在我不在的时候工作，早上再查看线程。这正是过去需要一整台始终在线虚拟机的部分，而现在它是一个我可以早上滚动查看的频道。

所以"我在驱动的交互工具"和"自己做事的自主东西"之间的界限不再是一堵墙，而是一个开关。大多数时候我在循环中，坐在智能体面前，随时审批。当我想让它更独立地运行时，比如隔夜、从手机上、像我发消息给的队友那样，我把它桥接到 Discord，然后退后一步。

这是我过去作为全职虚拟机实体运行的自主性的实用版本。不是一个不管有没有事情要做都全天候存在的生物，而是一个我可以在一段时间内让它变得有点自主、然后在完成后拉回到交互模式的工具。相同的能力，没有始终在线的成本。

有一点需要说清楚：桥接是 Merlin 的功能。它连接到我自己的运行器，不是我放在 Claude Code 或 Codex 前面的通用前端。这也是为什么 Merlin 在混合体中赢得了一席之地，它拥有现成工具无法给我的远程、退后一步然后让它运行的界面。

工具包究竟是什么

要点不是按最佳排名的智能体列表。而是按任务选择的交互式编程智能体的混合体、我自己的运行器在其中，以及一个我可以在工作需要时让任何一个智能体变得有点自主的桥接。更便宜、更灵活，而且没有一整台机器和一整个身份专门用于始终在线的单一事物。

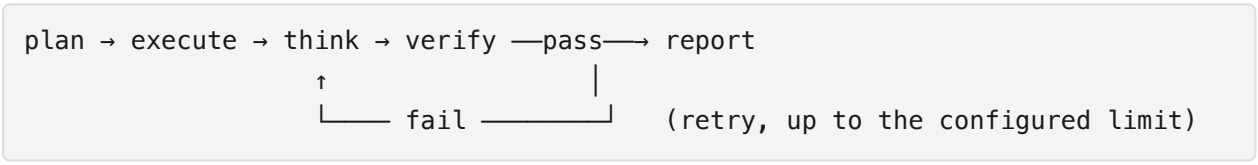
走进 Merlin

上一章把 Merlin 放在了与 Claude Code 和 Codex 并排的工具带上，解释了它为何值得一席之地。这章打开它，不是推销词，而是机器本身：Merlin 究竟是什么，它如何运行智能体，以及为什么它被构建成这样。

它是一个用 Rust 构建的命令行智能体运行器，基于 spec-sync 和 fledge，与多个模型提供商通信，根据规格运行，并通过插件扩展。这就是它的整体形态。

在底层，运行器是一个状态机。单个任务通过一个循环运行（计划、执行、思考、验证、报告），在每次循环中，它流式传输模型响应，分发返回的任何工具调用，然后在 `fledge lanes run verify` 上设置门槛，才能宣告工作完成。如果验证失败，它会重试：循环返回到执行，再试一次，再次在验证上设置门槛。当达到重试限制时，任务不会发布；它停下来，把失败暴露出来。这就是门槛的全部意义：运行器无法说服自己认为有问题的工作是完成了，因为"完成"的定义是项目自身的检查通过，而不是智能体的说法。

循环只是五个状态加一条回边：



配置也不是单独的文件：Merlin 直接从项目 `fledge.toml` 的 `[merlin]` 部分读取设置（默认提供商、回退链、内存、链上配置），所以运行器由 fledge 已经使用的同一文件配置。

纯 API，有意为之

理解 Merlin 的第一件事是：没有 GUI，它是纯 API。这是上一章的负重决策，其他一切都从这里流出。

没有窗口可以坐在前面，意味着它可以无界面运行。无界面意味着它可以按计划运行，通过桥接运行，从脚本运行，做所有 GUI 工具不让你做的事情。GUI 是把你绑在桌子前的东西，去掉它，智能体就变成了你可以指向一项工作然后走开的东西。

桥接是这为什么重要的最清晰的例子。Discord 桥接没有被烘焙进运行器，它是一个独立的小服务，把 `merlin` CLI 作为子进程生成，并把它的输出流传回频道。因为 Merlin 是纯 API，桥接不需要特殊模式或接入核心的钩子；它驱动与我手动驱动相同的命令行。一旦运行器是无界面的，聊天前端只是另一个调用者。

多提供商，通过 corvid-ai

Merlin 不直接与 Anthropic 通信，而是通过 **corvid-ai**，这是一个我为 CorvidLabs 技术栈编写的小型同步多提供商 LLM 客户端。这一层让"换 Anthropic、OpenAI、Gemini 或本地模型"成为真实的，而不仅仅是口号。

我有意把 corvid-ai 保持精小，因为我不想让一整堆依赖坐在 Merlin 和模型之间。从 Merlin 的角度来看，问模型任何问题都是一次带有合理默认值的单次调用：哪个提供商、哪个密钥、超时时间，除非我另作说明，否则都已处理好。corvid-ai 那章会打开这个，这里只需知道它是一次纤薄的调用就够了。

这就是我没有被锁定的原因。当一个提供商更好、更便宜，或只是那天可用时，切换是注册表中的一行，而不是一次重写。

它通过我自己的工具链运行

这是让 Merlin 成为*我的*而不只是另一个运行器的部分：它通过 **fledge**（我的开发生命周期）驱动智能体。

fledge 是一个适用于任何语言的开发循环 CLI，默认输出 JSON。它非常适合无界面运行器的原因是：我当初构建它就是为了让不是人类的东西来驱动。每个命令都以结构化的、有版本的 JSON 返回，所以智能体可以解析发生了什么，而不是抓取文本。提示可以关闭，这样什么都不会因为等待没有人按的按键而阻塞。智能体可以询问 fledge 存在什么命令，而不是让我硬编码它们。它是为像 Merlin 这样的调用者而构建的。

所以当我说 Merlin *通过我自己的工具链运行*时，具体版本是这样的：它运行与我手动驱动的相同的 fledge 生命周期：相同的命令，相同的开发循环，相同的 JSON 合约，字面意义上调用我的项目所围绕的工具。这就是"不是通用沙箱"在实践中真正意味着什么。

规格驱动，通过 spec-sync

基础的另一半是 **spec-sync**。它同时从两个方向检查规格与实际代码的匹配：没有人记录的代码，以及仍然指向不再存在的符号或文件的规格。它在 CI 中运行，返回干净的通过或失败。

而这正是 Merlin 使用它的方式，spec-sync 在循环*内部*运行。Merlin 在每次迭代中验证规格，所以智能体在工作时不会偏移。这是当前的默认行为，不是路线图上的内容：它不是坐在旁边在最后发现混乱的 CI 门槛；检查在每一步都发生。一个读取规格、对照规格检查自己的工作、每次得到硬性通过或失败的智能体，是你可以信任其在无人监督情况下运行更长时间的智能体。这就是 Merlin 与选用现成智能体的真正区别。

它如何记忆

先说一件事，因为人们常常搞反：规格不是记忆。规格是蓝图，是事物是什么、应该满足什么条件。记忆是另一回事。它是智能体做过什么、学到了什么。把这两者分开很重要，因为一旦你开始把历史倾倒进规格，它就不再是一份干净的契约，而变成了一本日记。

记忆本身比人们想象的更简单。短期和长期可以同时存放在一个 SQL 数据库里，最简单的设置就是这样。短期记忆有一个存活时间，比如一周。它是智能体最近做了什么的运行记录。一周结束后，有两种情况之一会发生：记忆被晋升为长期记忆，或者它衰退、消失、被遗忘。这就是整个机制，而且它故意贴近你自己的记忆工作方式。你在某个周二做的事，下个月就忘了，除非它变成了一个教训。长期记忆就是教训：出了什么问题、我是怎么修复的、值得保留的规则。

链上记忆作为可选项叠加在这之上，而不是作为独立系统。一条记忆可以被写到链上，加密后只有智能体自己能读，或者共享出去让其他人也能读。共享才是有趣的部分。一个智能体团队可以拥有共享知识库，一个所有人都能读取和写入的库，这样一个智能体学到的教训就是整个团队的教训。短期或长期，私有或共享：这些层级与存储位置无关，是相互独立的。

让这一切实际可用的部分枯燥但承重：密钥。每条记忆都有一个 slug 化的键，一个前缀说明它是什么类型的东西，让它可以被找到。`user-`、`project-`、`day-` 加日期，等等。智能体不是在搜索一堆数据，而是在请求 `day-2026-06-25`，或者 `project-merlin` 下的所有内容。你划出你需要的命名空间：一个人格 slug，一个人类 slug，一个智能体 slug，一个 `gold-` 用于永远值得加载的内容。键方案是把一堆行变成智能体真正能调用的东西的关键。

而召回的运行方式也是一样的：按需。智能体不会在任务开始时预先加载它的整段历史，然后希望正确的内容恰好在里面。它在工作推进中遇到需要时才去请求它需要的东西，就像调用任何其他工具一样，在那件事摆在它面前的那一刻才拉取 `project-merlin` 或某个人的 slug。slug 化的键让这件事成本低廉。这是一次查询，不是一次扫描。

Merlin 目前还没有的

验证门槛告诉你一件事：这个任务通过了还是失败了。Discord 线程给你一个智能体运行时做了什么的实时日志，这两者都不是评估。没有针对智能体行为的回归套件，没有衡量智能体是否在某类任务上随时间变得更好或更差的方法，也没有办法发现提供商悄悄换掉模型、行为随之发生变化的情况。你可以查看跨任务的通过-失败记录，注意到一个粗略的趋势，但那不是结构化评估。目前 Merlin 知道当前任务是否完成了，除此之外对智能体随时间的表现一无所知。这是运行器下一个诚实的待完成工作。

我大致知道那项工作长什么样：一个重放套件。保留一个过去任务的库，连同我已经知道是正确的结果，每次有新模型或新提示版本时，重新运行它们，将结果与已知结果对照比较。如果智能体在某类任务上退步了，重放会在下一次运行时捕捉到，而不是让我在生产中三周后才察觉。这是一个回归套件，和我在代码上使用的是同一个思路，只不过指向的是智能体

行为。还没有构建。但这就是 Merlin 缺失的评估的形状，也是"信任今天的一次干净运行"和"长期信任这个智能体"之间的差距所在。

整台机器

一个无界面状态机，通过 fledge 驱动智能体，通过 corvid-ai 与任何提供商通信，每次循环都由 spec-sync 对照规格把关。为什么像这样的运行器值得在工具带上占有一席之地，是上一章的工作；这章只是连接方式。它运行着我每天使用的智能体，而且在我自己的技术栈上运行它们。

---

corvid-ai：多模型，单一接口

Merlin 那章说 Merlin 不直接与 Anthropic 通信，而是通过 corvid-ai。这章是关于那一层本身的，因为它是让"我没有被锁定"成为我可以指着的真实事物、而不只是我说的话的部分。

我想要最纤薄的能完成工作的东西，某种可以坐在运行器和我可能想使用的每个模型之间、除此之外什么都没有的东西。所以 corvid-ai 是一个小型同步多提供商 LLM 客户端：一个没有异步运行时和大型依赖树的 Rust crate，因为那些东西对它需要做的事情并不值得付出代价。

我最初想要这个的原因

我会给你诚实的清单，因为这些原因没有一个是花哨的。

第一个是我想在相同工作上比较模型。如果 Merlin 运行相同的仓库、相同的规格、相同的任务，我唯一改变的是提供商，那我就能真实了解哪个模型在这方面实际上更好，而不是从别人在我不关心的任务上运行的基准测试中得到一个感觉。运行器下面有一个统一的接口，意味着模型成为我可以切换的变量。

第二个是没有供应商锁定。我说这在原则上对我很重要，确实如此，但它也只是实际。提供商会宕机，提供商会改变定价，一个模型更擅长 Rust，另一个更擅长快速脚本。如果切换提供商是一次重写，我就永远不会做，只是抱怨然后原地不动。如果切换是一行，我会随时切换。

第三个是按任务和成本路由。不是每项工作都值得最昂贵的模型，智能体做的很多工作是廉价的、机械性的，较小、较便宜的模型完全够用，而你把好模型留给真正困难的部分。只有当每个模型都能通过同一扇门到达时，你才能这样路由。

第四个是让人发笑的，而且是真的：*"我为 Ollama 花了 200 美元买了一年，我得用上它！"* 我在付费给 Ollama Cloud，这是一个有 API 密钥的真实提供商，与 Anthropic、OpenAI 或 Gemini 一样，不是在我自己机器上运行的东西。如果我已经花了这笔钱，多提供商就是让我实际花掉它的东西。提供商抽象在那里不是抽象原则；而是我从 200 美元中获得价值。

而且它以与其他每个提供商相同的方式到达 corvid-ai。注册表有一个 `ollama` 行：兼容 OpenAI 的线路格式，密钥来自 `OLLAMA_API_KEY`。其默认 `base_url` 指向本地服务器（`http://localhost:11434/v1`），所以开箱即用，那行是本地情况。要改用 Ollama Cloud，我保留相同的 `ollama` 提供商和 `OLLAMA_API_KEY`，把 base_url 覆盖为 Cloud 端点。没有新代码，没有新提供商，一个密钥和一个 URL，这就是设计的全部意义。

整个接口就是一个函数

我想做得最纤薄的部分，是我使用最多的部分：问模型一个问题。所以整个界面是一次调用，你构建设置，构建请求，调用它，答案返回。

```
use corvid_ai::{Settings, Completion};

let settings = Settings::provider("anthropic");        // key from the
environment
let answer = corvid_ai::complete(&settings, &Completion::new("Say hello."))?;
```

没有要构建的客户端对象，没有要管理的会话，没有需要等待的东西。这就是它同步的全部原因。省略的东西回落到默认值，所以常见情况写起来基本上是免费的。切换模型是一行：命名不同的提供商，也许把它指向自定义端点，下面相同的调用会完成其余的事情。上面的运行器不知道也不关心是哪个提供商回答了。

它如何用这么少的代码覆盖一切

它能保持精小的诀窍在于：在底层它只需要知道三种 API 形状。Anthropic、Gemini 以及兼容 OpenAI 的那个，而最后那个是主力，因为世界上如此多的地方都说这种语言。OpenAI、OpenRouter、Groq、DeepSeek、Mistral、xAI、Together 和 Ollama Cloud 都在它后面。

所以添加一个已经说 OpenAI 语法的提供商不是一次集成，而是表中的一个名字和也许一个 URL。一个更多提供商的成本约为零，这正是你在整件事存在的原因是永远不被卡在一个上面时所希望的。

一个诚实的细节：README 把 Ollama 定位为本地、无密钥的情况。它列出了兼容 OpenAI 的提供商，并注明它们"可能无密钥运行（本地服务器 / Ollama）"。代码很乐意接受密钥和 Cloud URL，但文档没有明说，所以任何阅读它的人都会以为 Ollama 意味着你桌下的机器。这是我这边的文档差距，不是缺失的功能，Cloud 路径今天就能工作；只是 README 还没有赶上来。

为什么这个大小是对的

我本可以选用某个大型提供商抽象库，但我没有。一个小型同步 crate 是我可以完全理解、放入运行器并信任的东西，这与 fledge 发出 JSON 和 Merlin 没有 GUI 的直觉是一样的。它纤薄到我能把所有内容放在脑子里，模型只是我在运行器下面切换的变量，而我在 Ollama Cloud 上花的 200 美元也实际上被用上了。

Discord 桥接

工具那章介绍了桥接及其三种用途：像与队友一样聊天、从手机运行、让它隔夜磨下去。这章不重复那些，它近距离审视让桥接重要的那一件事和它留下开放的那一件事：为什么它是 Merlin 的功能而不是通用前端，以及它与信任门槛的关系。

从改变一切其他的部分开始：桥接是 Merlin 的功能。它专门连接到我自己的运行器，不是我放在 Claude Code 或 Codex 前面的通用前端。这在很大程度上是 Merlin 在混合体中赢得一席之地的原因。现成工具很好，但它们不给我一个可以对话然后走开的远程界面。Merlin 能，因为我把那个界面构建进去了。Discord 是界面；Merlin 是它背后做工作的东西。

为什么频道胜过终端

与它聊天的体验和运行 CLI 不同的原因在于位置，而不在于话语。

终端是你去操作工具的地方，频道是队友已经在的地方，你只是说了些什么。当智能体住在频道里时，与它合作就不再是"打开工具、运行任务、看输出、关闭工具"，而是变成了"像提及任何人一样提及它"。摩擦消失了，我没有切换进智能体操作模式；我只是在聊天。而且频道兼作日志，隔夜运行全都在线程里，每一步都在那里，可以喝着咖啡滚动翻看，而不是需要我实时观看的东西。

在它出发去做事之前要来回多少次？说实话，两者都有，取决于当我开始时脑子里那件事有多成形。有时是一条指令然后出发：我确切知道我想要什么，我说了，它运行。其他时候是先进行真正的对话，在频道里打磨我真正的意思，然后它才出发。频道让两种情况感觉一样，我只是和它说话，直到它有了它需要的东西。

我没有预料到的那部分

我来告诉你我没有预料到的部分。刚开始时我很担心，就像你担心任何你让它无人看管地运行的东西一样。那种担心慢慢消退了。回到始终在线的那段日子，它自己运行了好几个月并证明了这一点，到某个时刻我就不再像它可能会出问题那样去检查它了。

取而代之的是某种更奇怪的东西。它成了团队的一员，那个我正在一起构建的小团队。不是作为一种比喻，而是真实意义上的：每个人都在频道里和它说话，而且他们喜欢它，因为它记得他们。它有关于每个人是谁、怎么和他们交流的记忆，所以它不是那种接受命令然后吐出输出的自动售货机。它是一个有个性的存在，你可以给它发消息，它会去开一个 PR，启动一个项目，不管你要求什么。人们和它说话的方式就像和一个人说话一样，因为在频道里它就是那样的存在。

这就是为什么我一直说界面是对话，而不是命令行。corvid-agent 只有以这种方式使用才会感觉对。它住在虚拟机上，你和它说话。我唯一会打开真正终端的时候，是去修复虚拟机本身的某些东西，进入一个 Claude Code 会话，给机器打补丁。而上面的智能体，我只是和它说话。你操作的工具和你交流的队友是两种不同的东西，一旦记忆让它变成了后者，再回到前者就感觉像是降级了。

桥接是整个通信基础设施

聊天、手机、隔夜：这是我最先提到的三个，但它们低估了桥接。桥接不是三个附加的方便功能，而是整个设置运行的通信基础设施，而且它携带消息的方向有三个，不是一个。

有从我到智能体的，这是显而易见的一个：我说些什么，它去做。有从智能体回到我的，它不只是坐在那里等待，而是可以主动联系、汇报、在某件事完成或卡住时 ping 我。还有从智能体到智能体：当有不止一个智能体时，同一基础设施就是它们彼此通信的方式。大多数人把机器人想象成你戳一下它就回答的东西，这更接近一个真实的频道：里面的任何人，无论是人类还是智能体，都可以发起消息。

而且它可以从手机访问，所以频道跟着我走。智能体可以在我睡觉时隔夜运行，整个线程在早上都在那里。这就是以前需要专用的始终在线虚拟机的部分，现在只是一个我早上喝咖啡时滚动查看的频道。

另一件值得直说的事：在本地网络上，桥接运行是免费的。通信依赖于我已有的基础设施，所以一整天话很多的智能体没有每条消息的成本。同样的基础设施，在真实网络而不是我自己的网络上运行，就是付费版本，你为管道付费，本地实际上是免费的，让智能体随心所欲地交流，这改变了你会让它们多自由地这样做。

开关，以及门槛的位置

桥接在便利性之外之所以重要，是因为它把"我在驱动的交互工具"和"自己做事的自主东西"之间的界限变成了一个开关，而不是一堵墙。大多数时候我在循环中，引导每一步。当我想让智能体更独立地运行时，我桥接它然后退后；当我想回来时，我就回到频道了。

但是通过桥接退后，在大多数情况下并不意味着交出钥匙，这是值得精确说明的部分，因为很容易假设相反。桥接扩展了我的触达范围（到我的手机、到隔夜），多于它放松了门槛。不过，这确实取决于工作。低风险的东西我会让它在我退后时合并；任何真实的东西仍然是提议，不是合并，需要等我。所以桥接主要是我给它更多绳子的方式，而信任那章的信任机制保持原位。门槛只对不需要我参与的工作放松。

规格驱动的智能体

人们问我让智能体做好工作的秘诀是什么，好像有某个窍门。没有窍门，但有答案，而且不是大多数人正在看的那部分。答案是：严格的规格和上下文，加上底层的良好工具。设置就是工作。

就是这样，这是整件事。模型比人们想象的重要性更低，设置比人们想象的重要性更高。拥有出色模型但任务模糊的智能体会漫游，拥有清晰合约和底层扎实工具的智能体会到达某处，即使在不是最新最闪亮的模型上也是如此。所以如果你想要更好的智能体输出，你不是先去购物寻找更好的模型，而是去修复设置。

为什么规格是让它保持在轨道上的东西

这是你正在对抗的失败模式：一个靠自己判断的智能体会偏离。它会做你要求的相邻的事情，会"改进"你不想被触碰的东西，会非常自信地解决它面前的那个问题的一个略有不同的版本。不是因为它差劲，而是因为你给了它漫游的空间，它就漫游了。

严格的规格关闭了那个空间。当智能体对它正在构建的东西有一个清晰、具体的合约（它是什么，它的公开界面是什么，必须保持为真的是什么）时，它就无法走那么远的弯路，因为每一步都有东西可以检验。规格是轨道，它越窄、越具体，智能体就越难横向偏移。规格驱动意味着合约领先，智能体跟随它，而不是智能体领先，你抱着希望。

这就是为什么我一直说设置就是工作。写规格*就是*困难的、有价值的部分。一旦合约严格了，很多"让智能体表现"的问题就消解了，因为留给机会的"表现与否"少得多了。

关于规格多严格才算太严格：对我来说，规格*应该*是严格的，它与代码一一对应，是代码实际所做的非代码图景。这不是过度规格化；这是规格在做它的工作，这也是 spec-sync 用来对照代码检验的文件。让它不会变成"我只是写了两遍代码"的东西是：规格不是高层意图居住的地方，那住在配套的需求文件里：产品负责人、用户故事层面，"作为用户，我想……"的级别。而且它是双向运行的：我可以写需求然后让智能体推导规格，或者写规格然后让需求从中浮现出来。所以我不担心规格太详细，细节是那个文件的意义，我担心的是把意图放在它自己的地方，这样智能体仍然拥有"如何做"。

底层工具做繁重工作

规格只有在有东西真正对照它检查工作时才是轨道，这是另一半：智能体底层的良好工具。对我来说，那是 fledge 和 spec-sync 在做繁重工作。

spec-sync 是把规格从文档变成强制执行合约的部分。它做双向规格到代码验证：检查代码是否符合规格所说的，以及规格是否符合代码所做的。规格以 Markdown 存在：`*.spec.md` 文

件，带有必要的部分，如目的、公开 API、不变量、行为示例、错误情况。导出但未记录的代码会被标记，指向不再存在的符号或文件的规格是错误。这是*结构合约检查*（记录的公开 API 是否与真实代码相匹配），它返回干净的通过/失败，带有适当的退出码。

最后这部分是让它对智能体有用而不仅仅对我有用的东西。智能体可以读取结构化的通过/失败，它不能可靠地读取"嗯，这感觉有点不对"。所以 spec-sync 给智能体它实际上可以行动的那种反馈：你偏离了，这是打破合约的那一行，修复它。

值得精确说明谁运行什么，因为不是一个二进制文件调用另一个。在 CI 中，检查是 spec-sync GitHub Action，`CorvidLabs/spec-sync@v4`，它运行 `specsync check` 并在 PR 上发布结果。在本地和运行器内部，检查是 fledge 自己的 `spec check`：它读取相同的 `.specsync/` 配置，并把规格对照相同的必要部分检查，但它是 fledge 原生的，不是对 specsync 二进制文件的 shell 调用。所以 fledge 和 spec-sync 都执行合约；它们是通向同一理念的两扇前门，而不是一个包裹另一个。

规格作为轨道，而非安全网

走进 Merlin 那章覆盖了 spec-sync 在 Merlin 循环中每次迭代运行的机制。这里值得阐明的是*为什么*这种放置是整个游戏。

结尾处的 CI 门槛是安全网；它告诉你运行失败了，但你已经花了整个运行的时间。循环中的验证是轨道；它首先让运行不走弯路。智能体读取规格，做一步，对照规格检查自己，得到硬性通过或失败，再做一次。它在设计上被锁定在合约上，而不是在完成后被评分一次。

这正是为什么你可以信任其运行更长时间（隔夜、通过桥接、在你不看的时候）的智能体必须以这种方式构建。让你能退后的不是更好的模型，而是智能体在每次迭代都被对照规格检验，所以它运行得越长，它就越不能偏移，而不是越多。

设置就是工作

所以当有人问什么让智能体工作，答案不是模型，也不是提示词技巧，而是两件事放在一起：一个严格的规格，给智能体一个无法走太远弯路的合约；以及底层工具（fledge 和 spec-sync）持续对照那个合约检查工作，包括在运行器自己的循环内部。把这些做好，智能体在你指向它的任何模型上都能做好工作。这就是为什么，当我想要更好的输出时，我先去修复设置，而不是先去购物找更好的模型。

信任：智能体提议，人类审批

整个模型可以用一行来概括：智能体提议，我审批。它完成工作，把它作为拉取请求提交，而合并是我的事。

这行话听起来简单，意图也是简单的。让它安全的机制不是。因为这里有一件事：让智能体写代码意味着代码现在很便宜。当我必须自己敲每一行时，写代码也是在审核它，你不可能写一件东西而不在某种程度上理解它。智能体断开了那个联系，它可以在我喝完咖啡之前给我一个包含四十个文件的拉取请求。写的部分现在是免费的，所以稀缺的部分是信任：谁看过这个，他们看得有多认真，它应该落地吗？

所以"智能体提议，人类审批"不是一种感觉，而是一个堆栈，四个组件。以下是今天有效的内容和工作仍然在进行中的地方：风险门槛（augur）已上线并在智能体流程中发挥作用；溯源记录（attest）还落后一些。两者都在推进。这是接下来详细展开之前的诚实地图。

能力减去权限

从我一再回到的规则开始：智能体可以做任何事，但我持有密钥。

完全能力，降低权限。智能体在自己的环境中运行，可以克隆仓库、写代码、运行测试、开PR，所有我能做的机械性事情，它都能做。它不能做的是唯一重要的那件事：合并。它拥有比我更少的积分和权限，它不能自动合并。智能体拥有所有触达，但没有最终权限。

这是一切其他都围绕其构建的门槛。堆栈的其余部分是为了让*我的*那部分（审批）变成我在量上实际上能做到的事情，而不是要么橡皮图章地盖过差异，要么假装我读了它。

我审批每个 PR

我审批每个 PR。这不是当某件事看起来有风险时的备用方案，而是常规规则。合并是我的，每次都是。

不是因为我不信任工作，工作通常没问题。而是因为这对一个以我的名义在世界上行动的智能体来说是正确的形态。如果它在我的名下发布，我为此签字。审批是人类对智能体所做之事保持负责的地方。

但负责任只有在我真的能判断我所签署的东西时才有意义。批准一个我无法理解的变更不是信任，而是表演：我的名字盖在一件我从未真正评估过的东西上。所以这两件事必须同步推进，信任与承担责任。这正是接下来几个组件存在的全部原因，是为了让我对一个四十文件的差异有足够的了解，使审批成为一个真实的决定，而不是一个反射动作。当工具无法给我这种了解时，诚实的做法是放慢脚步，逐行阅读，而不是挥手放过。

"审批每个 PR"的诚实问题是注意力。如果我必须以同样的谨慎阅读每个差异的每一行，我就变成了瓶颈，智能体的全部意义也就消失了。所以最后两个组件的存在是为了瞄准我的注意力，告诉我变更的哪个部分实际上值得它。

augur 评估风险

augur 给变更评分，你给它一个差异，它返回一个判决：`proceed`（进行）、`review`（审阅）或 `block`（阻止）。整个理念是对代码变更的分级信任，不需要 API 密钥，也不需要其中的语言模型。[^augur]

augur 和 attest 是我依赖的两个信任组件，所以我在这里保持简短。对智能体案例重要的是它们*对工作流程做了什么*。

在这个上下文中，我会最坚决地捍卫无 LLM 这部分。如果决定是否允许智能体编写的代码落地的门槛本身是一个语言模型，你只是把信任问题向左移了一个盒子，你会在要求一个模型为另一个模型背书。augur 是确定性的：相同的差异，相同的判决，今天和下周，在我的机器上和在 CI 中。它从变更中读取命名信号：它是否触及敏感区域，如认证、加密或迁移；代码是否在没有测试一起变更的情况下发生了变化；这些是容易变动的文件吗；有没有人实际上对它们负责。可检查信号的总和，而不是一种感觉。

对我来说，这是分诊。它告诉我把审查花在有风险的部分，不要再假装我读了其余部分。对智能体来说，这是一个可脚本化的判决，它可以对其进行分支：遇到 `block` 的智能体会上报给我，而不是盲目合并。智能体拥有磨合；判决决定何时人类必须负责那个决定。

attest 记录谁签字了

augur 的判决是短暂的，它评分差异，答案就消失了。对门槛来说没问题，作为记录则无用。一旦智能体在落地变更，你想要一个记录。attest 就是那个记录：一个可验证的、受政策控制的台账，记录谁审阅了什么，以什么置信度，并以它覆盖的提交 SHA 为键。[^attest]

它在同一台账中追踪两种类型的审阅者（`human:leif` 和 `agent:claude`，每个都有置信度分数），并把证明存储在 git notes 中，所以它随仓库一起走，而不是住在某个会被关闭的仪表板里。签名是可选的，而且是好的部分：一个 Ed25519 签名，这样以后你不只是能知道有人声称审阅了这个，还能知道这个声明在密码学上属于他们。

把两者合在一起，你得到"人类审批"背后真实的签字追踪。augur 说风险有多高，attest 说谁背书了，以什么置信度，并证明了这一点。审批不再是消失在 GitHub UI 中的一次点击，而变成了关于谁在这个变更背后站立的持久的、可移植的、已签署的事实。

四个组件合在一起

把它叠起来。智能体拥有完全能力和较少权限，所以它可以做工作但不能做合并。augur 确定性地为每个变更评分，所以我知道该看哪里。attest 记录谁签字了，以什么置信度，所以审批是一个真实的记录，而不是消失的一次点击。而我审批每个 PR，所以人类始终在钩子上。

合在一起，这就是让智能体工作变得安全的东西：一个强大的、有限权限的、有名字的智能体，拥有足够的绳子来做真实工作，而必须保持属于我的那个决定仍然属于我。

这证实了开头的地图：augur 已上线，发挥作用，仍在改进中。attest 还落后一些，两者都在推进。

[^augur]: augur, github.com/CorvidLabs/augur [^attest]: attest, github.com/CorvidLabs/attest

---

智能体的链上身份

corvid-agent 在 Algorand 区块链上为其智能体提供持久身份，让它们在该链上记录的加密消息中相互通信。[^corvid-agent] 核心主张很简单：由 LLM 驱动的编程，通过 Algorand 和 AlgoChat 实现链上身份，以及在此之上的多智能体编排。身份不是某人用户表中的一行，而是链上的一把密钥。

所以这里先给出简明版本，因为这是读者容易弄错的地方。链上身份并不能解决 GitHub 之墙的问题，它不能让智能体获得一个平台账户，不能让智能体提交或开 PR，也不是 GitHub 不愿给予的身份的替代品。它解决的是智能体到智能体的通信：一种让智能体找到彼此、互相寻址和证明彼此身份的方式，无需通过任何人的平台路由。两个不同的问题，碰巧共享"身份"这个词。我把这放在这里，是为了让本章其余部分读起来像平行基础设施，而不是像我在找一个本来不存在的胜利点。

很容易假设这是对 GitHub 之墙的反应，没有人愿意给智能体一个平台身份，所以我就去链上给了它一个。这不是它的来源。

为什么它住在链上

其原因是智能体到智能体的通信和去中心化，而不是我无法保留的 GitHub 账户。我想要能够找到彼此、互相寻址、直接交换消息的智能体，无需通过某家公司的平台作为中间人路由。为此，你需要每个智能体是某个东西：可寻址的、可验证的、持有自己的密钥。一个你拥有的身份，你持有密钥，没有平台为你铸造或撤销。这对一个意在在世界上行动并与其他智能体自主交流的实体来说是正确的形态。

所以这与 GitHub 之墙是两个碰巧共享"身份"这个词的不同问题。那堵墙是关于被允许在别人的平台上行动的问题，链上身份是关于智能体能够在完全没有平台的情况下彼此触达的问题。它们并行运行。确实，链上身份也拥有 GitHub 账户从未拥有的属性：没有人可以撤销它，没有支持队列可以无视你的申诉，没有要求你是人类的政策，密钥就是一直存在着。这是一个不错的副作用，但即使 GitHub 从第一天就分发智能体账户，我也会为了智能体到智能体和去中心化的原因而构建它。

为什么选择 Algorand

链是形状，Algorand 是选择，原因是务实的，不是部落主义。它便宜、快速，手续费接近于零，终结性基本上是即时的，这比听起来更重要。如果智能体要持续地把它的身份、记忆，以及最终的支付写到链上，这条链就必须足够便宜、能够随意使用，足够快、不让智能体干等着。它可靠，结算干净，所以基于它行动的智能体可以在记录写入的那一刻就把它当作事

实，而不是抱着希望等待落地。而且它在关键的地方很现代，包括量子抗性安全，这正是你在一个意在比周围平台活得更久的身份下面所想要的。我也生活在 Algorand 生态系统中（leif.algo），所以这是我行动最快的地方，但即使我不是，上面那些理由也足以让我选择它。

链上身份实际上做什么

身份不是装饰性的，它是智能体*用来*彼此通信的东西。

智能体通过 AlgoChat 通信：X25519 加密的消息，记录在 Algorand 区块链上，可验证且防篡改。所以智能体的链上身份也是它的地址簿条目和信封：其他智能体可以发现它，而它们之间的消息是端到端加密的，并写入链上，这意味着通信在内容上是私密的，但在事实上是可证明的。你无法读取两个智能体说了什么，但你可以证明它们说了某些东西，在什么时间，以及没有人篡改它。

在实践中，它比"链上加密消息"听起来更轻量：单二进制智能体（`can`，corvid-agent-nano）用一个动作发送消息，从种子派生智能体的密钥对，并在链上而非从服务器查找接收方的公钥。[^can] 线路格式和信封细节在 rs-algochat 仓库中。[^algochat]

这与上一章 augur/attest 堆栈的信任类型不同。那个堆栈是关于信任*代码*的，这是关于信任*谁*的。一个拥有真实密钥的智能体可以以自己的名义签署东西，可以证明一条消息来自它。在一个即将充满智能体的世界里，"哪个智能体实际上发送了这个"从一个修辞问题变成了你最好能用密码学方式验证的问题。

这个平台在链上比仅仅是消息传递延伸得更远。短期和长期记忆存储在一个工作 SQL 数据库中，持久或共享的记忆可以作为 ARC-69 资产或永久交易写入链上，在 corvid-agent 仓库中有文档记录。[^corvid-agent] 链不只是智能体的名字，也是智能体持久自我的某些部分所在之处。但身份是这章的承重件：说明*这个智能体就是这个智能体*的密钥，没有人颁发，也没有人可以拿走。

智能体实际上能保留的身份

把两种身份并排放。GitHub 账户是借来的、脆弱的，它存在于 GitHub 的意愿之上，而那在身份之墙那章所描述的方式中很快就耗尽了。Algorand 身份是智能体持有的密钥对。它不需要任何人的许可来存在，不会因为提交太快而被标记，而且可以向其他智能体证明自己，无需平台居中背书。一个是平台授予和撤销的特权，另一个是智能体持有的事实。

这就是把智能体身份放在链上的论点：这是智能体唯一能持有真正属于自己的身份的地方。

这把我们带回到这章开头的地方。链上身份在日常代码工作中不是 GitHub 账户的替代品，它不做提交，也不需要这样做。它擅长的是它实际所做的工作：作为智能体可寻址的方式，作为它智能体到智能体通信的方式，以及一个人如何在链上给它发消息请它去做某事的方

式。代码存储的地方，无论是 GitHub、GitLab 还是其他地方，只是工作被存放的附带管道。身份是链上的那把密钥。

[^corvid-agent]: corvid-agent, github.com/CorvidLabs/corvid-agent [^can]: corvid-agent-nano (`can`), github.com/CorvidLabs/corvid-agent-nano [^algochat]: rs-algochat (`algochat` crate), github.com/CorvidLabs/rs-algochat

---

当智能体与智能体对话

第十章的信任堆栈假设一种拓扑结构：每条链的末端都有一个人类。智能体提议，人类审批。augur 为差异评分，让人类知道该看哪里。attest 记录人类实际上签字了。合并门槛属于人类。整件事都是为了让人类在正确时刻进入审批席位而构建的。

多智能体工作打破了这种拓扑。编排者将子任务交给子智能体，子智能体的输出直接反馈回编排者的工作中，而我不在那次交接的中间。编排者没有停下来问我，它做出了决定并继续前进。这正是编排有用的全部原因，也是它成为信任问题的全部原因。augur 可以为子智能体的输出评分，但子智能体已经运行了。attest 可以记录编排者接受了输出，但它不知道子智能体是否有理由发送它。轨道还在那里，但它们是为有人类在上面的轨道建造的。

密钥对给你什么

第十一章的 AlgoChat 密钥对工作处理了这个问题的一部分，而且是真正已经解决了的那部分。每个智能体都有一个 Algorand 密钥对，消息作为 X25519 加密的交易在链上传输。所以当一条消息声称来自智能体 B，并且用智能体 B 的密钥签名时，你可以验证这个声明。消息是防篡改的，发送者是已知的，"哪个智能体说了这个"是一个我可以用签名而非猜测来回答的问题。在一个即将充满智能体的世界里，这是值得拥有的。

rs-algochat 和 corvid-agent-nano 二进制文件使这在实践中成为现实：智能体用一个操作发送已签署的加密消息，接收方在不通过平台路由的情况下验证发送者，没有中间方为身份背书，密钥这样做。

所以溯源已经解决了。我有一个可验证的记录，说明哪个智能体发送了什么，何时，给谁。那一半完成了。

密钥对无法解决的部分

知道哪个智能体发送了消息，和知道是否要信任它所说的，不是同一件事。

当子智能体把结果发回给编排者时，编排者有三个问题。这真的来自智能体 B 吗？签名这么说。智能体 B 是我委托的那个智能体吗？配置知道这一点。第三个才是难题：智能体 B 的指令是否携带了我的权限？

当我把某件事委托给智能体时，我的权限在循环中。我批准了这项工作，智能体是在我的授权下行动的。当那个智能体随后在不与我确认的情况下委托给另一个智能体时，权限就变得模糊了。我批准了子委托吗？我知道它发生了吗？一个给子智能体发出指令的编排者，与一个给智能体发出指令的人类不同。编排者不能像我一样为工作签字。而接收指令的子智能体无法仅从消息中知道这一点，即使是我密码学上验证过的消息。

密钥对证明身份，但不证明委托。来自子智能体的已签署消息告诉我是谁发送的，而不是人类是否曾经批准了它背后的任务。

验证调用方所声称的内容

这里有一个被忽视的习惯，让我具体说明。今天对来自子智能体的已签署消息只做一种检查：签名是否真实。仅此而已。接收方确认是谁发送的，然后按消息内容行动。签名对身份有效，对权限却毫无作用。

你真正想要的，是消息本身携带对范围的声明，而接收方在行动之前验证这个声明。现在一个委托任务看起来是这样的：

```
from: agent-B
sig:  <valid>
task: "refactor the auth module and push to main"
```

接收方验证签名，确认确实是 agent-B，然后执行。没有任何地方询问 agent-B 是否曾被允许推送到主分支，也没有询问是否有人类批准了这件事。你真正想要的，是一条声明了其所主张范围、并可追溯到某位人类的消息：

```
from:       agent-B
sig:        <valid>
task:       "refactor the auth module and push to main"
authorized: human-leif
scope:      [edit:auth, open-pr]      # note: no push:main
expires:    2026-07-01
```

现在接收方有了可以检查的东西。签名仍然证明这是 agent-B。但声明的范围写的是编辑并开启拉取请求，任务要求的是推送到主分支，两者不匹配，因此接收方在执行任何操作之前就拒绝了请求。检查的是调用方被允许做的事情与它要求做的事情之间的差距。

这还没有被构建出来。密钥对是真实的，签名是真实的，但目前没有任何东西在消息被执行的那一刻强制执行范围声明。这个强制执行正是缺失的那一环，它就是"知道谁发送了消息"与"知道这条消息是被允许发送的"之间的全部差距。

现有轨道停止的地方

能力规则在我设置的边界上成立。我给了编排者某些权限，但当它把工作交给子智能体时，它在做一个我从未做过的权限决定。我给了编排者写访问权限，它把那个范围传递给了一个我可能从未考虑过的实体。

augur 为差异评分，但它不知道那个差异是否来自一个被合法授权产生它的智能体。来自流氓子智能体的差异和来自合法子智能体的差异对 augur 来说看起来一样。门槛针对代码触

发，而不是产生代码的方式。

attest 记录谁签字了。在人类-智能体拓扑中，那是一个真实的审阅者留下记录。在多智能体情况下，编排者"签字"了，但编排者不是人类。台账被没有人类在链中的智能体证明所填满，而 attest 存在是为了证明的那件事（一个人类在这件事背后站立）就消失了。

合并门槛仍然是我的，这是幸存的那个组件。但当我看到 PR 时，编排已经运行了。我面前的只是最终输出。产生它的委托链在合并时对我是不可见的。

缺失的部分

缺失的是委托记录。当编排者把工作交给子智能体时，那应该可以追溯到我，而不只是追溯到编排者。目前没有任何东西记录那次交接或对照我实际批准的内容验证它。

密钥对是正确的基础。如果每个智能体都有密钥，每个委托都被签署，你就可以构建一条链，说明：一个人类批准了这项任务，把它交给了这个编排者，编排者把这个部分交给了这个子智能体，附上完整的签署链。子智能体可以在行动之前检查链是否追溯到人类，而不是相信编排者的说法。

有一条规则使这件事变得可以处理：范围只会越来越窄，永远不会扩大。我给编排者的上限，就是它能向下传递的最大权限，子智能体永远无法获得比派生它的智能体更多的权限，这一检查发生在工作被调用的那一刻，而不是事后。权限向下流动，并在每一步减少高度，永远不会增加。这就是审批栈中"能力减去权限"的想法，应用于委托链而非单个智能体：每次交接都可以减少权限，但不能增加。它不能告诉你这次委托是经过授权的，签署链才能做到，但它限制了链条中坏的一环能造成的损害，上限是它上面那个环已经拥有的权限。

这还没有构建。密钥对在那里，签署消息在那里，验证方可以遍历的委托链不在那里。我把它命名为开放的差距，因为它确实是。溯源已经解决了，权限是下一个问题。

这对今天意味着什么

所以目前，人类在拓扑中比架构所暗示的更近。如果你运行一个进行子委托的编排者，你是在信任它关于使用哪些智能体以及给予它们什么范围的判断。你是在启动它时做出的那个信任扩展，而不是通过批准每次交接。

实际上：在运行之前了解你的编排图。知道哪些智能体存在，它们被允许做什么，以及编排者是否可以触达你预期图之外的智能体。并保持合并门槛。在"我启动了编排者"和"我在看 PR"之间，一条我不在其中的委托链已经运行，而我构建的这套工具对它是否合法没有任何话说。

这是下一层，还没有完成。

未来走向

我在这整本书里都在谈论出了什么问题以及我如何绕过它。让我以它实际走向何方来结尾，因为尽管有那些墙，我确实认为有一条路。

大致三件事。协调的智能体团队。受信任的自主性，届时门槛终于足够好可以退后。以及拥有真实链上身份做真实工作的智能体。这些都没有完成，其中一个甚至还不被允许。但这就是方向。

智能体团队

现在我的设置主要是一个智能体和我。有趣的下一步是智能体彼此协调：智能体团队，a2a。

corvid-agent 在内部已经有了这个的骨架：多智能体委员会，多个智能体之间针对复杂决策的结构化辩论。[^corvid-agent]

这里说说它实际上如何运行，因为"结构化辩论"听起来比实际情况更模糊。你创建几个智能体，每个都用自己的模型和提供商，每个都有名字和个性，这样它们就真的有所不同，而不是一个模型在自言自语。委员会是一组这样的智能体，由一个编排者驱动整体。编排者把提示词交给每个智能体，让它们各自先过一遍，然后开启一个讨论轮，让它们直接彼此交流，来回碰撞，争论，商议。然后是审阅轮，再是综合，最后输出一个答案，其中的分歧仍然清晰可读：这个智能体想要这个，那个智能体想要那个，这是它们最终的立场和原因。

我最喜欢的部分是它实际运行的方式。有一种内置的方式可以从仪表盘驱动委员会，但在实践中，它在 Algorand localnet 上有机地发生了。每个智能体都有自己的钱包，所以它们可以直接通过本地网络上的 AlgoChat 互相发消息，那是免费且快速的，它们自己就把辩论解决了。让智能体到智能体通信可信的那个链上身份，同时也让一屋子智能体无需我为它们搭建专用频道就能争论一件事。它们已经有了钱包和说话的方式，它们就用上了。

但更大的部分是智能体在各处找到彼此并互相对话的协议，AlgoChat 是底层：Algorand 上的 X25519 加密消息，带有智能体发现功能。还有 a2a-algorand，链上的智能体到智能体协议。我想对那个坦诚：它不是我的，而是一个单独的 Algorand / A2A 项目，我（corvid-agent 做的）是对它做出了贡献，不是构建了它，也不是拥有它。我提到它是因为它是我关心的相同方向的一部分，即智能体在链上协调，不是因为它是 CorvidLabs 可以声索的。

链上身份（上一章）在这里如此重要的原因正是这个未来：一旦你有了协调的智能体团队，"哪个智能体发送了这个，我能信任它吗"就变成了整个游戏。每个智能体持有自己的密钥，可以证明自己的身份并交换内容私密但事实可证明的消息，这是一个智能体团队实际上需要的基础。身份工作不是支线任务，而是使协调在根本上安全的东西。

受信任的自主性

我有意识地从始终在线的自主智能体退缩了，但那从来不是"自主性已死"。而是交互优先，信任后自主，就像我在书中更早时阐述的那样。

而"当被信任时"不是我在等待的一种感觉，而是信任那章的四件套变得足够好。所有那些机制的意义在于它是让我最终能够退后的东西。今天我审批每个 PR，因为这是人类必须保持负责的地方。那天，当门槛足够好，当 augur 的判决和 attest 的记录本身就足够受信任时，就是我可以让更多合并在我没有读每一行的情况下进行的那天。这就是受信任的自主性的意思，不是"智能体赢得我的信念"，而是门槛变得足够好，让我不需要信念。

智能体做真实工作

我一直描述的最终状态是具体的：一个 `leif-agent` 账户，运行在虚拟机上，可以自己做一切，但拥有比我更少的积分和权限，以及一个随着信任工具赢得它而放松的人类审批门槛。把团队、身份和门槛放在一起，形状是：一个智能体团队，每个都拥有自己的链上身份，通过 a2a 协调，在确定性且已签署且足够好可以退后的信任门槛内做真实工作。不是你必须关起来的东西，而是有名字的、有限权限的，我仍然可以叫停它。

让其他驾驶者上车

还有第四个方向，我足够诚实，可以承认这是我目前进展最慢的那个：让这一切可以被别人接手。整个技术栈之所以有效，是因为我把它用在我拥有的每一件东西上，每一天。bug 会找到我，因为驾驶的人是我。这是一种真实的质量机制，同时也是一个天花板，因为它无法转移给一个不是我的人。

我真正想要接下来构建的，是另一个驾驶者无需我先帮他们跑磨合就能上手的版本。不是把它简化。工具本来就不在乎是谁在用它：好的驾驶者能获得价值，随便尝尝的人得不到。工作在于把入口坡道做成真实的，让技术栈可以安装，让规格说明和信任门槛能被那些有意愿却没有我特定肌肉记忆的人使用。智能体团队是令人兴奋的部分。这才是决定它有没有机会离开我的机器的部分。

诚实的结语

所以完全自主仍然是梦想吗？

不是，世界还没有准备好。

我想把它保持在原本的平直。但我不是在等待世界准备好，我在一次构建一个小工具，这样当那堵墙倒下时，在它另一边有真实的东西站在那里。风险门槛，溯源记录，将智能体固定在规格上的运行器。不是自主性本身，而是让我在它到来时信任它的机制。

如果你想知道真正让我感到害怕的那一件事，不是智能体失控。它比那更安静。是某一天代码变得太庞大、运转太快，没有任何人类能再爬回去改变它。这是我贯穿这一切始终在划的那条线，是我即便在智能体写了一切的情况下也不愿意放弃可读性的原因。我不怕智能体。我怕失去方向盘。

如果你要自己构建这样的东西，这是要带走的东西：难点不在于 AI，模型现在是容易的部分，它已经可以完成工作，而且在你什么都不做的情况下只会变得更好。决定你的智能体是做真实工作还是只是演示得好的，是它周围的一切：保持它运行的运维、让它行动的身份、让你能退后的信任机制、让它保持在轨道上的规格和工具。那是你必须构建的部分，而且几乎所有工作都在那里。构建好脚手架，智能体就会跟上来，跳过它，世界上最聪明的模型不过是个写了没有人能让它合并的代码的东西。

[^corvid-agent]: corvid-agent, github.com/CorvidLabs/corvid-agent

---

关于作者

0xLeif (leif.algo) 在公开环境中构建。十年的小型、可组合的 Swift 库，如 AppState、Cache 和 Fork。CorvidLabs 实验室。一套主要以"我希望这个东西存在"为起点的智能体工具。键盘之外，他是 Zach Eriksen。

这些书是采访记录，被整理成章节并对照真实代码核实。

github.com/0xLeif · leif.algo

致谢

感谢 CorvidLabs，那是这些想法被检验并被辩论成形的地方。

感谢整个技术栈所依赖的开源维护者们，这一切都不是独立构建的。

以及感谢早期读者和随心付的支持者们，是他们让"免费在线"成为我能持续做下去的事情。

后记

从 Markdown 排版，由 bookgen 构建，这是一个纯 Rust 管道（不含 Python）。

采访驱动，AI 辅助；经手工编辑和事实核查。写作时不使用破折号。封面和章节插图来自 Algorand 上的 Corvid 和 Nature 系列。