智能体软件工程

与随机性队友一起
以前所未有的规模
构建可信赖的软件

翻译：李豪

“你如何向看不见的人解释颜色呢？
他们需要亲身体验，才能明白。”

丹尼尔 · M · 杰尔曼 (Daniel M. German) 教授
论与 *AI* 队友的协作

关于作者

艾哈迈德 · E · 哈桑 (Ahmed E. Hassan) 教授是穆斯塔法奖 (Mustafa Prize) 得主，这一奖项常被比作诺贝尔级别的荣誉。他同时也是 ACM Fellow、IEEE Fellow、AAIA Fellow，以及 NSERC 斯泰西纪念研究员 (NSERC Steacie Fellow)，这是加拿大为科学与工程领域的中青年领袖设立的最高荣誉之一。

目前，哈桑教授担任加拿大研究主席（Canada Research Chair）和加拿大女王大学 NSERC/BlackBerry 软件工程工业研究主席。他是全球引用次数最多的软件工程研究人员之一，也是唯一一位同时获得 ACM SIGSOFT 有影响力教育家奖（2019 年）和 IEEE TCSE 杰出教育家奖（2020 年）的学者。这两个奖项分别由全球两大权威学会颁发，是软件工程教育领域的最高荣誉。他已培养出 35 位高校教授，这些学者在全球各地领导着各自的研究团队。

作为人工智能增强软件工程（AI-Augmented SE）、软件仓库挖掘（MSR）和 AIware 社区的创始人之一，以及加拿大皇家学会（Royal Society of Canada）会员，他的职业生涯跨越三十余载，在工业界（如 IBM Almaden 研究中心、BlackBerry）和学术界都担任过领导角色。

译者序

当我第一次读到艾哈迈德 · E · 哈桑（Ahmed E. Hassan）教授的这本《智能体软件工程》时，立刻被它的视角所吸引。这是一部从软件工程学科本身出发，严肃思考人与 AI 如何协作构建可信赖软件的著作。

这本书讲什么

当 AI 能以前所未有的速度生成代码时，软件工程该怎么做？

本书提出了智能体软件工程这一框架。它的核心论点是：AI 不是工具，而是队友，一个能力强大但会犯错的队友。因此，可靠性不能依赖于 AI 本身的完美，而必须通过工程系统来保障。全书围绕四个部分展开：第一部分建立概念基础；第二部分解决 AI 队友的可靠性问题；第三部分将协作扩展到团队规模；第四部分为不同角色提供行动指南。作者在前言和正文中有详尽的介绍，此处不再赘述。

这本书适合谁，该怎么读

如果你是技术负责人或工程管理者，从第一部分顺序读起，重点关注第三、四部分。如果你是一线开发者，第一、二部分是你的核心章节。如果你时间有限，直接读第四部分，它是全书的浓缩与行动路线图。作者在前言中有更详细的说明，建议一并参阅。

感谢哈桑教授的信任与支持，使中文版得以面世。感谢每一位选择阅读本书的读者。

李豪
2026 年二月

版权声明

第一版：2026 年 v0.4 增加了后记。
v0.5 更新了信任工程章节。
v0.5a 更新了信任工程、能力工程和协调工程章节。

我为何写这本书，以及我是谁

我们正处在软件工程史上一个怪异、美妙、略带失控的时期。几十年来，我们的领域是万物背后安静的机器。如今它却成了餐桌上的谈资。那些连 Git 和吉他（Guitar）都分不清的人，一边啜饮着意式浓缩，一边自信地宣称“AI 将终结软件工程”。得了吧！

我花了超过 25 年，致力于构建我称之为“智能软件工程”的东西。我一直固执地相信：AI 会重塑软件的构建方式，而且会很快。我很幸运，而且坦白说，很感激有这么多人加入了这一愿景。今天，智能体软件工程对研究和实践的影响已经无法否认。

在同一时期，我曾与一些世界最大公司的领导者和团队合作，尝试将 AI 整合到大规模的软件工程组织里：那些系统混乱不堪，充斥着遗留代码、时间线、审计、人类以及偶尔的“火灾”。一个模式反复出现：人们低估了软件工程是什么。他们把它当成一个工具、一个流程、一份清单。而实际上，它是一个鲜活的复杂系统，需要积极的管理、持续的校准和尊重！

接着，生成式 AI 来了，音量旋钮直接拧断了。曾经的极客话题变成了头条新闻。“软件工程的终结”成了一句你在世界各地的咖啡馆都能听到的随口断言，通常紧接着就有人建议你可以“氛围编程”，然后直接上线交付生产。

我曾经会坐下来与领导者们沟通，向他们讲述真实的情况。但这不再具有可扩展性。我没有时间了。所以我做了那件我曾发誓永远不会做的事：我写了一本书。

目前围绕软件工程的混乱是我前所未见的。说实话，如果这种情况持续下去，我的下一个项目可能会是一部动作片。或者恐怖片。情节相同，只是灯光不同。

这本书的目标很简单：软件工程不会消失。它正变得比以往任何时候都更重要。在超智能体编码（hyper agentic-coding）时代，软件工程是横亘在人类与自信产出的无稽之谈之间的学科。如果你从这本书中只带走两样东西，那就是：1）软件工程从来不只是写代码；2）一个拿着智能体编码工具的傻瓜，仍然是个傻瓜。

在我们开始之前，先谦逊地说明一点：这个世界正以荒谬的速度前进。我会有一些地方弄错。一些想法会过时。这没关系。重点不在于完美。重点在于把它写下来，使其可被讨论，并将这场对话从随意的闲聊提升为一门成熟的工程学科和科学学科。

最后，我并非独自一人走到这里。我感谢那些我有幸与之合作和领导的人类，那些构建基础的研究人员和工程师，那些在混乱约束下交付现实的实践者，以及那些推动编码大语言模型和智能体系统前沿的人们。一路走来，我还遇到了一群轮换的、来自邻近星球的紧张、聪明、偶尔过于自信的外星人，他们各有各的个性和奇怪的小癖好。你会认出他们是 ChatGPT、Claude、DeepSeek、Gemini、Grok 和 Qwen。只能说他们以……一些在餐桌上难以解释的方式提供了帮助。

请享用。

艾哈迈德 · E · 哈桑（Ahmed E. Hassan），人类，地球

本书适合谁阅读

这本书写给那些对最终成果负责，而不仅仅是完成产出的技术领导者。这包括工程总监、副总裁、首席技术官、高级和首席工程师、技术负责人，以及任何在真实组织中负责交付、质量、风险和开发者体验的人。如果你是开发者，这本书同样适合你，只是视角不同。它会为你提供一些心智模型，帮助你理解周围正在发生的变化，以及如何在环境变化时保持高效。

本书不是一本关于智能体编码的指南。如果你想要提示词技巧、工具工作流，或者想掌握最新智能体编码工具链的战术细节，你有无穷无尽的选择。YouTube、博客和供应商文档里充满了巧妙的操作，而且它们会越来越好。本书假设你已经相信智能体编码能够奏效，或者你预期它很快就能很好地工作。如果你好奇我的看法——我认为它今天已经奏效，将来会变得更好！

以下是这为何重要。生成代码的能力从来都不是软件工程的核心瓶颈。弗雷德·布鲁克斯（Fred Brooks，《人月神话》作者）几十年前就论证过，没有“银弹”，因为困难的部分不是打字。困难的部分在于复杂性、沟通、概念完整性、协调，以及随时间推移的变化。智能体编码可以放大产出，但它不会自动解决这些问题。事实上，如果你让速度超越了协调一致，情况可能会变得更糟。傻瓜拿着工具，仍然是傻瓜。

智能体编码真正改变的，是制造软件的经济学。当一个人可以探索更多替代方案、运行更多实验、并行交付更多变更时，限制性资源变成了注意力，而不是击键次数。关于那些罕见的“10 倍效率开发者”的旧故事开始翻转。智能体工作流可以将许多人从 1 倍提升到 10 倍，在某些情况下甚至

更高。这意味着劳动力分布发生了变化，而不仅仅是峰值。这也意味着，昨日衡量卓越的信号，将不再清晰地对应到明日。智能体软件工程时代的赢家，不会是打字最快或编码最猛的黑客。他们将是最能清晰设定意图、管理风险边界并要求证据的个人和团队。

因此，本书是关于智能体软件工程，而不是智能体编码。它探讨的是，当你能生成 100 倍产出、迭代成本骤降时，如何运行软件工程。它关于如何利用 AI 队友，同时控制其弱点；关于如何帮助人类和 AI 都达到 10 倍甚至 100 倍的生产力，同时仍然产出值得信赖的软件。你无法仅凭氛围编程实现这一点，但氛围编程在获取能力与参与方面，仍然是一个强大的均衡器。

如果你仅仅将像 Claude Code 这样的智能体编码工具链视为强大的工具，那么你只看到了图景的一小部分。AI 队友的影响触及需求、设计、测试、评审、发布、事件响应和团队协调。我们看到的不是软件工程的消亡。我们看到的是编码作为瓶颈的终结，以及可信赖的智能体软件工程作为一门“让速度安全”的学科的崛起。

如果你领导团队，本书是你应对新现实的行动手册。如果你构建软件，本书将帮助你看清你所处的系统、塑造它的力量，以及那些最重要的战略行动。

关于 AI 使用的说明

本书中的见解、模式和结论，源于我多年的实践经验，以及与数千名开发者和学者的直接共事。它们完全代表我个人的观点。

在撰写本书时，我践行了我所宣扬的：一种瑞士式的 AI+人类协同思考愿景，即让 AI 成为认知伙伴，而不仅仅是工具。我的 AI 队友扮演了严格的陪练伙伴角色：挑战假设，压力测试论点，并帮助我更精确地阐述复杂概念。这种合作关系提升了论述的清晰度和结构，但并未生成核心思想，也未曾取代人类的判断。

书中所有图片均使用 Gemini 或 ChatGPT 生成。

书籍前言

感谢你选了这条难走的路：不在 AI 炒作中寻求舒适，而是在一个充满随机性的世界里，坚守工程的严谨性。

这是我对你的承诺：这本书不会让你相信 AI 魔法。它会教你，怎样从随机性贡献者那里，工程化出可信赖的结果。

我们脚下的地在天天变。模型在进化，工具在转型，今天的突破是明天的基准。我选择写原则，而不是快照。工具链会变，模型会改进。但对工程化的可靠性、严谨的证据、审慎的团队协议的追求，不会变。

别再吃那份牛排了：软件工程的工程时刻

这篇前言分四幕展开，像任何有力的论证那样，从舒适到现实，再到规模和行动手册。按顺序读下来，你会感受到逻辑随着推进慢慢收紧。

第一幕：牛排的舒适

当我观察软件行业对智能体软件工程的反应时，《黑客帝国》里有个场景总在我脑子里打转。塞弗盯着牛排说：“我知道这块牛排并不存在。我知道我把它放进嘴里的时候，母体会告诉我的大脑它又多汁又美味。九年过去了，你知道我意识到什么吗？无知是福。”那一刻，塞弗选了一个令人舒适的幻觉，因为真相太乱，他接不住。

我们在正干着同样的事。我们想相信真正的工程意味着完全确定性的控制。我们坚持要求每一步都可审计，每个组件都可预测，每个结果都可解释。然后我们遇到那些会随机出错、偶尔过度自信的 AI 队友，于是断定它们还没准备

好。我们选了确定性带来的熟悉舒适感，而不是未来的概率性现实。这种反应可以理解，但也短视。

把话说直白点。智能体软件工程是这样一门学科：通过让整个软件工程系统（人员、流程、工具、工件）准备就绪，从随机性贡献者（包括 AI 和人类）那里生产出高质量、可靠、可信赖的软件。它不是要找完美的智能体，而是在那些可能会以一定概率失败的组件之上，用对的约束和证据，工程化出可信赖的可靠性。

请将本书视为智能体编码工具（包括 Claude Code）所缺失的配套手册。那些工具给了你一个队友。这本书则给你一个软件工程系统，让这个队友能在人员、流程、工具、工件这些维度上规模化地工作。那些工具加速了编码，这本书展示的是，当速度爆炸时，怎么调整你的工程纪律，让你仍然能和随机性贡献者一起交付可靠、可信赖的软件。这就是《智能体软件工程》的承诺，也是它的问题：以前所未有的规模，构建你能信赖的软件。

第二幕：工程从来都不是确定性的

工程从来不是组装完美零件的艺术。土木工程师不会从完美的钢材、完美的混凝土、完美的工人开始。材料有公差，焊缝可能失效，团队会犯错。让桥梁安全的不是神奇的钢材，而是系统：冗余、安全边际、检查制度、标准化的实践，以及对失效模式死磕到底的关注。

不管我们承不承认，软件一直活在这同一个世界里。硬件有非零的失效率，网络会丢包，磁盘会损坏，人会误解规格。但我们仍然构建出了能跑飞机、医院、金融市场的系统。可靠性从来不是零件的属性，它是端到端软件工程管道的属性。

所以说，执着于确定性队友，是在解决错的问题。问题不是“AI 能不能像我们一样理解?”，而是“我们能不能构建出

一些流程，让随机性贡献者也能产出可信赖的结果?”这个问题的答案我们已经有了，因为软件工程一直以来就在干这个。

能力先于解释，是进步的常态。一个反复出现的焦虑是，AI 可以在没法解释的情况下产生结果，或者它可能没“理解”就达成了成果。但人类在形式化力学之前很久，就会投矛、骑马、建大教堂了。蒸汽机跑起来的时候，我们还没有完整的热力学理论。数学里，直觉常常先行，形式化随后。我们行动，然后解释。系统之所以有效，是因为我们测试它、约束它、让它健壮、让它可信赖。

所以，如果 AI 能完成工作却无法解释自己，这没什么史无前例的。这很正常。

信任的双重标准，才是问题所在。听听这些反对意见：智能体有幻觉，会误读需求、误解意图，有时候会给出脆弱的解决方案。把“智能体”换成“人”，其描述的正是协作开发的日常现实。人类开发者也有状态差的日子，会误解需求，会交出没人能维护的“聪明”代码。可我们没禁掉人类编码。我们构建了技术系统来管理人的易错性：版本控制、代码审查、测试金字塔、事件响应、变更管理、值班轮换、事后分析，还有那些奖励清晰、惩罚个人英雄主义的文化。

当我们要求智能体完美，却容忍人类的易错性时，这种反对意见就不是技术问题了。它是情绪化的。它是《黑客帝国》那份牛排背后，虚假的舒适感。它是紧抓着一幅从来就不真实的工程图景。

第三幕：当规模翻转瓶颈时

智能体软件工程是个规模化事件，而规模会改变关键所在。一个 AI 队友能生成的代码，比任何人类团队合理监督的速度更快、并行线程更多。靠代码审查是逃不出那个未来的。审查和测试依然至关重要，但重心变了。可靠性的主要单

元，变成了端到端的软件工程系统：工作流、不变量、权限、门禁、可追溯性、冗余、快速恢复。

这个转变，正是区分软件工程 1.0、2.0 和 3.0 的关键。软件工程 1.0 是代码优先：人驱动整个流程，用经典工具支持需求、设计、实现、测试这些标准活动，通常靠基于程序分析的工具。软件工程 2.0 保持了同样的代码优先姿态，但加入了 AI 模型来支持那些经典软件工程活动，典型形态是副驾驶。人仍然驱动循环，所以认知负荷还是压在开发者身上。**软件工程 3.0 翻转了瓶颈，整个工程系统必须跟着变**。当输出爆炸时，稀缺资源不再是打字，而是注意力。人必须拥有意图和风险边界。AI 队友必须在边界内执行，并且通过端到端的软件工程系统，持续产生证据，证明边界被遵守了。

规模化的信任，建立在确定性证据上，而不是确定性步骤上。一旦我们定义了一个目标，我们不应该要求 AI“用人的方式”构建它。我们应该要求 AI 在约束条件下，按规格，带证据地构建它。我们别再混淆“我们想象解决方案的方式”和“解决方案必须满足的条件”了。确定性证据是工程杠杆：作为门禁的测试，运行时检查，从意图到差异的变更追踪，独立验证，以及在压力下站得住脚的审计轨迹。

第四幕：团队运动与行动手册

软件工程不只是工具，智能体也不是“又一个工具”。如果你觉得把 Claude Code 部署下去就万事大吉，那你会得到一个惊喜。软件工程是人、流程、工具、工件一起运作的事。智能体软件工程逼着我们把这一切当成一个工程系统来看，而不是依赖提示词。我们需要更高级的构建模块：明确的目标，结构化的约束，权限边界，标准升级路径，证据要求。

智能体软件是团队运动，所以 AI 必须能提供反馈。最具误导性的心智模型，是一个孤独的人告诉一个孤独的 AI 该做什么。真正的软件是团队构建的，智能体软件将由更大、更

多样化的团队构建：人和 AI 队友一起工作，并且对彼此有更高的期待。

这意味着双向的交流。人设定意图、价值、约束。AI 队友提出选项，对有矛盾的地方提出异议，在决策要跨过边界时升级处理。

一个具体的例子。你让 AI 队友实现一个功能，它发现这功能需要改数据库模式。在一个正规的组织里，这不是随随便便就能改的。它可能需要迁移计划、数据保留检查、回填策略、性能分析，还要数据库架构师批准。正确的智能体行为，不是悄悄改模式，也不是扔出一个模糊的问题。正确的行为是发起一个结构化的请求，附上证据，提一个安全的迁移方案，然后申请需要的权限。AI 得学会怎么把人当成治理和专业知识的端点来用，就像它用编译器和 CI 一样。可信赖的智能体软件工程不是单向的提示管道，它是一个精心设计的团队协议。

分工不是弱点，它是文明实现规模化的方式。贸易和专业化不会让某一方吃亏。它们让组合起来的系统更丰富，因为专业化减少了浪费，放大了优势。经典的葡萄酒和布料例子（西班牙和英国）告诉我们，就算一方在两方面都更强，当各自生产自己牺牲最小的产品时，双方也都能获益。更宽的教训没变：协调加上专业化创造盈余，这个道理完全适用于人和 AI 的协作。

人擅长的是人独有的部分：定义价值、设定目标、选择权衡、判断什么可以接受。这些是合法性的来源。AI 可以优化你给的东西，但它没法先告诉你什么值得优化。定义目标函数，是我们的工作。

目标一旦清晰且可衡量，AI 就擅长做广泛搜索、快速迭代、生成替代方案、跑实验、探索人类没时间遍历的设计空间。在一个项目内部，最实用的形态是一个队友在界面中，背后是一群专家，每个专家都有工具、约束和成功标准。这不是智能体泛滥，这是工程化了的分工。

现在就构建端到端的软件工程系统，滑向冰球要去的地方。很容易按智能体现有的能力快照去评判它们。那是滑向冰球曾经在的地方。真正的优势在于预见：顺着轨迹，而不是盯着位置。智能体软件工程的那个版本很简单：在能力曲线让你的现行软件工程系统过时之前，就把合适的软件工程系统建好。等完美了再建系统，是等能力到来时被压垮的最可靠方式。

行动手册在其他工程领域已经写好了：规范，约束，验证，恢复。在智能体软件工程里，这意味着明确的目标，影响范围限制，权限化的工具，作为门禁的必需测试和检查，从意图到变更的可追溯性，通过独立验证实现的冗余，还有把故障当成设计里预期要及早捕获的事件。

所以，对，别吃那份牛排了。别再等那个永不产生幻觉的确定性智能体。我们从来就没有过确定性的队友，以后也不会有。但我们能拥有一个更强大的东西：建立在随机性智能之上的工程化可靠性，规模大到前人无法企及。

智能体软件工程不是工程的终结。它是软件工程这门学科，变得比以往任何时候都更加重要的时刻。

放下牛排吧。别再等那个永远不会来的确定性了。智能体时代的赢家，不会是生成代码最多的团队。他们会是那些日复一日、变更复变更，能把随机性的输出，转化成确定性信心的团队。

那就是软件工程的工作。那就是摆在我们面前的工作。让我们开始构建吧，构建那些能在这个时代要求的规模上，赢得信任的软件工程系统。

如何阅读本书

本书分为四个部分。其中两个部分适合所有读者。

从第一部分开始。它为智能体软件工程奠定基础，阐述了与 AI 队友协作意味着什么、它们带来的优势以及你必须应对的矛盾。

如果你是软件构建者，第二部分至关重要。它专注于如何让带有随机性的 AI 队友变得可信赖，让你能在不牺牲严谨性的前提下与它们协作。

如果你是工程团队领导者，第三部分至关重要。它涉及为规模化部署 AI 队友而进行的平台工程，以及你在提供信任与协调基础方面的角色。这些基础能让你的团队在安全的前提下快速前进。

如果你只读一部分，请读第四部分。如果你是繁忙的业务领导者，这是唯一必读的部分。它将全书内容整合起来，并为开发者、技术领导者、业务高管、教育者和研究者提供了针对具体角色的行动指南。

关于附录：附录 A（参考表格）和附录 B（术语表）最好留待你准备实施这些想法时使用。第一遍阅读时可以先跳过，等到需要模板和精确术语定义时再回来查阅。

*开始之前，最后一个请求：*与 AI 队友协作的智能体软件工程正在改变一切。请放下你的戒备，思考一些超越当今软件工程作战手册里所谓“合理”范畴的东西。从我这边，我承诺这将是一本力求平衡的书，公平地对待炒作与风险。

目录

I 智能体软件工程与 AI 队友 1

理解智能体软件工程与 AI 队友 2

1 智能体软件工程：从直觉狂欢到可信工程 3
- 1.1 氛围编程自有其位，但绝非施工现场 . . . 4
- 1.2 一门学科，两种模式，各配优化工作台 . . . 6
  - 1.2.1 人类工作台 . . . 7
  - 1.2.2 智能体工作台 . . . 9
- 1.3 工件即接口 . . . 10
- 1.4 智能体软件工程协调工件及其用途 . . . 11
- 1.5 结构化意图：任务简报 . . . 14
- 1.6 结构化证据：合并就绪包与决议记录 . . . 15
- 1.7 结构化升级：咨询请求包与决议记录 . . . 16
- 1.8 结构化指导：指导包 . . . 18
- 1.9 结构化执行与编排：工作流程运行手册 . . . 20
- 1.10 信任需要强制执行 . . . 21
- 1.11 工件是新的工程层，而非定制化宏 . . . 22
- 1.12 可信性即代码：确定性、强制执行与流程即代码 . . . 23
- 1.13 本章主线与未来之路 . . . 24
- 1.14 边栏：软件工程谱系：氛围编程 → 氛围工程 → 智能体软件工程 . . . 25

2 发挥 AI 队友的力量 28
- 2.1 自动化阶梯：工具变身队友 . . . 28
- 2.2 杠杆效应：智能体软件工程改写劳动力经济学 . . . 30
- 2.3 本章结构：以优势与交互模式为核心 . . . 33
  - 2.3.1 本章使用的成本模型 . . . 34
  - 2.3.2 本章使用的控制点 . . . 34
  - 2.3.3 每个交互模式的结构 . . . 35
  - 2.3.4 交互模式如何关联智能体软件工程工件 . . . 35
- 2.4 集群 A：不知疲倦，不带评判 . . . 37
  - 2.4.1 它是什么 . . . 37
  - 2.4.2 它解锁了什么 . . . 37
  - 2.4.3 四元律速览 . . . 39

2.4.4 此集群中的交互模式 40
2.4.4.1 无限迭代，有限循环 40
2.4.4.1.1 它是什么 40
2.4.4.1.2 它解锁了什么 40
2.4.4.1.3 如何识别它（以及它替代了什么） 41
2.4.4.1.4 风险及应对方法 41
2.4.4.1.5 示例交互流程 42
2.4.4.1.6 起作用的原理 43
2.4.4.2 超越完成 44
2.4.4.2.1 它是什么 44
2.4.4.2.2 它解锁了什么 45
2.4.4.2.3 如何识别它（以及它替代了什么） 45
2.4.4.2.4 风险及应对方法 46
2.4.4.2.5 示例交互流程 47
2.4.4.2.6 起作用的原理 48
2.5 集群 B：强沟通者 49
2.5.1 它是什么 49
2.5.2 它解锁了什么 50
2.5.3 四元律速览 51
2.5.4 此集群中的交互模式 51
2.5.4.1 草率输入，清晰输出 51
2.5.4.1.1 是什么 51
2.5.4.1.2 它解锁了什么 53
2.5.4.1.3 如何识别它（以及它替代了什么） 53
2.5.4.1.4 风险及应对方法 54
2.5.4.1.5 示例交互流程 55
2.5.4.1.6 起作用的原理 56
2.5.4.2 多画图，少废话 57
2.5.4.2.1 是什么 57
2.5.4.2.2 它能解锁什么 58
2.5.4.2.3 如何识别它（以及它替代了什么） 58
2.5.4.2.4 风险及应对方法 59
2.5.4.2.5 示例交互流程 60
2.5.4.2.6 为什么这招管用 61
2.5.4.3 可缩放综合 62
2.5.4.3.1 是什么 62

2.5.4.3.2 它解锁了什么 63
2.5.4.3.3 如何识别它（以及它替代了什么）. 63
2.5.4.3.4 风险及应对方法 64
2.5.4.3.5 示例交互流程 65
2.5.4.3.6 起作用的原理 65
2.6 集群 C：广阔的世界知识 67
2.6.1 是什么 67
2.6.2 它解锁了什么 67
2.6.3 四元律速览 68
2.6.4 此集群中的交互模式 68
2.6.4.1 角色扮演 69
2.6.4.1.1 是什么 69
2.6.4.1.2 它能解锁什么 70
2.6.4.1.3 如何识别它（以及它替代了什么）. 70
2.6.4.1.4 风险及应对方法 72
2.6.4.1.5 示例交互流程 73
2.6.4.1.6 起作用的原理 73
2.6.4.2 魔鬼代言人 75
2.6.4.2.1 是什么 75
2.6.4.2.2 解锁了什么 75
2.6.4.2.3 如何识别它（以及它替代了什么）. 76
2.6.4.2.4 风险及应对方法 77
2.6.4.2.5 示例交互流程 78
2.6.4.2.6 起作用的原理 78
2.7 集群 D：复制成本近乎为零 80
2.7.1 是什么 80
2.7.2 解锁了什么 81
2.7.3 四元律速览 82
2.7.4 此集群中的交互模式 82
2.7.4.1 并行分解 82
2.7.4.1.1 是什么 82
2.7.4.1.2 它能解锁什么 82
2.7.4.1.3 如何识别它（以及它替代了什么）. 83
2.7.4.1.4 风险及应对方法 84
2.7.4.1.5 示例交互流程 85
2.7.4.1.6 起作用的原理 86

2.7.4.2 可弃式赌注，证据决定 87
2.7.4.2.1 它是什么 87
2.7.4.2.2 它解锁了什么 88
2.7.4.2.3 如何识别它（以及它替代了什么） 88
2.7.4.2.4 风险及应对方法 89
2.7.4.2.5 示例交互流程 90
2.7.4.2.6 起作用的原理 91
2.8 本章小结与后续内容 92

3 AI 队友的悖论 95
3.1 永不成长的初级开发者 95
3.2 本章为何令人乐观 96
3.3 布鲁克斯的视角：为何我们对 AI 队友如此苛刻 98
3.4 如何理解这四个悖论 98
3.5 悖论一：热切悖论 99
3.6 悖论 2：上下文悖论 104
3.7 悖论 3：隧道视野悖论 109
3.8 悖论 4：学习悖论 114
3.9 总结：悖论四重奏 119

II 让随机性 AI 队友值得信赖 121

面向 AI 队友的保证工程 122

4 任务工程：让意图清晰可验 125
4.1 根本矛盾 125
4.2 核心概念：任务简报即自主权契约 125
4.3 任务的版本管理 126
4.4 任务工程的关键实践 127
4.4.1 实践 1：意图对齐 128
4.4.2 实践 2：属性控验收 128
4.4.3 实践 3：概念计划对齐 128
4.4.4 实践 4：自主权边界与指令清晰度 129
4.4.5 实践 5：带汇总的迭代精化 129
4.4.6 实践 6：基于证据的收尾与合并就绪 129
4.5 任务工程模式 130
4.5.1 模式：先问再建 130
4.5.2 模式：共同思考检查点 130

4.5.3 模式：预置参数空间 130
4.5.4 模式：角色流动性 131
4.5.5 模式：优雅重启 131
4.5.6 模式：不变量，非轶事 131
4.5.7 模式：声明“不” 132
4.5.8 模式：建设性质疑 132
4.5.9 模式：升级轨道 132
4.5.10 模式：行内反馈，而非大段论述 133
4.5.11 模式：简报即法律 133
4.5.12 模式：合并就绪包优先 133
4.5.13 模式：迁腐问责制 133
4.5.14 模式：审计路径 134
4.6 任务工程反模式 134
4.6.1 反模式：跳过意图对齐 134
4.6.2 反模式：工单彩票 135
4.6.3 反模式：基于氛围的验收 135
4.6.4 反模式：金发姑娘范围失控 135
4.6.5 反模式：步步为营式计划 135
4.6.6 反模式：代码执念 136
4.6.7 反模式：完美的错误 136
4.6.8 反模式：简报腐化 136
4.7 如何衡量任务工程 137
4.7.1 指标：简报新鲜度（腐化率） 137
4.7.2 指标：属性覆盖率指数 137
4.7.3 指标：自主运行长度 137
4.7.4 指标：工具调用自主指数 138
4.7.5 指标：升级负载与质量 138
4.7.6 指标：评审就绪度评分 138
4.7.7 指标：未能合并率与根因编码 139
4.7.8 指标：验证差异 139
4.8 总结 139

5 上下文工程：驾驭随机性贡献者的知识 141
5.1 根源矛盾 141
5.2 核心理念：上下文是接口，不是垃圾场 141
5.3 上下文负载量表 142
5.4 上下文工程的关键实践 142
5.4.1 实践 1：播种最小工作集 143
5.4.2 实践 2：执行中主动管理负载 143
5.4.3 实践 3：隔离探索 143

5.4.4 实践 4：压缩而不失治理 144
5.4.5 实践 5：跨会话传输“干净”的连续性 144
5.4.6 实践 6：有意识地重置 144
5.5 上下文工程模式 144
5.5.1 模式：最小可行上下文 144
5.5.2 模式：隔离兔子洞 145
5.5.3 模式：压缩，而非删除 145
5.5.4 模式：授之以渔，而非授之以上下文 145
5.6 上下文工程反模式 146
5.6.1 反模式：盲目自动加载 146
5.6.2 反模式：上下文囤积症 146
5.6.3 反模式：静默压缩 146
5.6.4 反模式：静态维基倾泻与固定的 RAG 消防水带 147
5.7 主要构建块 147
5.8 上下文工作的实用文件结构 148
5.9 轻量级“上下文卡片”格式 148
5.10 衡量上下文工程 149
5.10.1 指标：压缩频率与可见性 149
5.10.2 指标：上下文冲突率 150
5.10.3 指标：固定不变量丢弃率 150
5.10.4 指标：上下文检索效率 150
5.10.5 指标：上下文传递保真度 150
5.10.6 指标：分叉到合并的规范性 151
5.11 基于证据的监督：跨领域控制 151
5.12 从一个队友到完整的软件工程系统 152

III 面向 AI 队友舰队的平台工程 154

团队规模腾飞：面向 AI 队友舰队的平台工程 155

6 协调工程：防撞与自主流水线 159
6.1 根本矛盾 159
6.2 核心概念：基于决策就绪包的异步协调与计划性集成 . . 161
6.3 协调工程的子组件 162
6.4 协调工程中的关键实践 163
6.4.1 实践一：设计清晰接缝，减少不必要的并行 . . 163
6.4.2 实践二：执行前向冲突管理器提交计划 163
6.4.3 实践三：在受控的隔离工作空间中执行 163

6.4.4 实践四：用分层就绪状态把关，而非单一的合并状态 164
6.4.5 实践五：利用 AI 队友原生策略解决集成冲突 . . 164
6.4.6 实践六：在组织各层级协调“协调基础设施” . . 166
6.4.7 实践七：为自主执行而设计流水线 166
6.4.8 实践 8：为智能体交接订立契约 166
6.4.9 实践 9：在决策点，而非执行点设置人工审批 . 167
6.5 协调中的集成工程 167
6.6 流水线工程：设计自主的多智能体工作流 168
6.6.1 人类是工作流架构师，而非实时调度员 168
6.6.2 剖析编排流水线 169
6.6.3 常见的流水线结构 169
6.6.4 智能体间的交接 171
6.6.5 流水线工程的经济账 171
6.6.6 何时用流水线，何时用手动调度? 172
6.7 协调工程模式 173
6.7.1 模式：使用“决策就绪”咨询包的异步协调 . . 173
6.7.2 模式：计划先行的集成调度 173
6.7.3 模式：用于提升决策质量与创新的 N 版本探索 . 173
6.7.4 模式：区分“工作中”与“工作后”的协调 . . 174
6.7.5 模式：像维护操作系统那样协调团队演进 174
6.7.6 模式：顺序验证流水线 174
6.7.7 模式：并行探索，结构化合并 175
6.7.8 模式：分层委托，集中集成 175
6.7.9 模式：审查循环，限制迭代 175
6.7.10 模式：分阶段审批点 176
6.8 协调工程反模式 176
6.8.1 反模式：合并就绪工作的集成积压 176
6.8.2 反模式：无计划并行，共享表面遭殃 176
6.8.3 反模式：非结构化的同步协调 177
6.8.4 反模式：人在每个循环中 177
6.8.5 反模式：隐式交接 177
6.8.6 反模式：单体工作流 178
6.8.7 反模式：缺乏可观测性的编排 178
6.9 主要构建模块 178
6.10 衡量协调工程 179
6.10.1 指标：集成冲突率 179
6.10.2 指标：解决集成冲突耗时 179
6.10.3 指标：误预测冲突率及根因分类 180
6.10.4 指标：流水线自动化率 180

6.10.5 指标：交接成功率 180
6.10.6 指标：流水线周期时间 180
6.11 总结 181

7 工作台工程：两种模式，两套环境 182
7.1 根本矛盾 182
7.2 核心理念：将人类工作台与 AI 队友工作台分离 184
7.3 工作台工程的关键实践 186
7.3.1 实践一：定义“铺好的路”，让数据包成为一等公民 186
7.3.2 实践二：为“一人对多机”构建人类指挥平面 . 187
7.3.3 实践三：通过差异优先审查、计划台账和精准反馈来压缩信任决策 187
7.3.4 实践四：为快速、自给自足的工作构建 AI 队友执行平面 188
7.3.5 实践五：自动化证据捕获以降低信任成本 188
7.3.6 实践六：从设计上就让 AI 队友执行工作台安全第一 189
7.3.7 实践七：像运营生产平台一样运营工作台 189
7.3.8 实践八：为企业级舰队可观测性和资源控制增设企业指挥中心 189
7.4 工作台工程模式 190
7.4.1 模式：工作台分离 190
7.4.2 模式：收件箱，而非中断 190
7.4.3 模式：将 N 版本比较作为一等公民 190
7.4.4 模式：对所有工件进行差异优先审查 190
7.4.5 模式：计划台账审查与偏离映射 191
7.4.6 模式：可寻址的内联评论与关联的 AI 队友响应 191
7.4.7 模式：可追溯意图的下拉式干预 191
7.4.8 模式：以无干预执行为设计目标 192
7.5 工作台工程反模式 192
7.5.1 反模式：聊天即 IDE 192
7.5.2 反模式：叙事式审查与模糊反馈 192
7.5.3 反模式：工具匮乏与人类复制粘贴循环 192
7.5.4 反模式：无成本和健康控制的无限并行 193
7.5.5 反模式：默认不安全的执行环境和工具链 193
7.6 主要构建模块 193
7.7 衡量工作台工程 194
7.7.1 指标：人类干预率 194
7.7.2 指标：咨询包决策时间 194

7.7.3 指标：会议负载趋势 194
7.7.4 指标：批准的再现率 194
7.8 总结 195

8 能力工程：角色、资质与持续改进 196
8.1 根本张力 196
8.2 核心概念：能动性、能力与记忆基底 197
8.3 作为平台工程与人员运营的能力工程 198
8.4 能力工程中的关键实践 199
8.4.1 实践 1：能力校准与角色分配 199
8.4.2 实践 2：作为代码的指导，具有结构和层次 . . . 199
8.4.2.1 指导机制谱系 200
8.4.2.2 使机制与意图匹配 201
8.4.2.3 为什么不应编码认知策略 202
8.4.2.4 合规性差距 204
8.4.3 实践 3：操作边界与升级矩阵 204
8.4.4 实践 4：资格考试与持续认证 205
8.4.5 实践 5：晋升与恰当的拒绝 206
8.4.6 实践 6：反馈驱动的改进循环与自我改进训练场 206
8.5 能力工程模式 207
8.5.1 模式：升级是一种功能，而非失败 207
8.5.2 模式：专业化队友，然后认证专业化 207
8.5.3 模式：指导方针和操作边界是代码，值得进行变更管理 207
8.6 能力工程反模式 208
8.6.1 反模式：无质量控制的定制 208
8.6.2 反模式：诗意或详尽的指导方针 208
8.6.3 反模式：将能力工程视为可选项或产品团队的副业 208
8.6.4 反模式：用概率性指导去约束刚性流程 209
8.6.5 反模式：把认知策略写进指导 209
8.7 主要构建模块 209
8.8 如何衡量能力工程 210
8.8.1 指标：生产性连续工作时长 210
8.8.2 指标：升级处理质量与路由准确率 211
8.8.3 指标：变更后的认证回归率 211
8.8.4 指标：指导方针膨胀率与冲突率 211
8.8.5 指标：拒绝与升级的健康度 211
8.8.6 指标：机制与意图的匹配率 212
8.8.7 指标：指导遵从率 212

8.8.8 指标：指导工件的生命周期健康度 212
8.9 总结 212

9 信任工程：以机器速度实现治理 214
9.1 根本矛盾 214
9.2 核心概念：信任工程的四大学科 215
9.3 核心概念：可逆世界与委派校准 217
9.4 核心概念：分层验证，或麦当劳给我们的关于随机性行动者的启示 218
9.5 核心概念：可解释自主性的三份清单 220
9.6 核心概念：面向随机性行动者的策略即代码，与默认可审计性 221
9.7 信任工程中的关键实践 222
9.7.1 实践 1：风险分级与自主权门控 222
9.7.2 实践 2：强制执行最小权限和工具访问边界 . . . 222
9.7.3 实践 3：身份感知的信任边界 223
9.7.4 实践 4：让审计追踪默认自动生成并冻结 223
9.7.5 实践 5：高风险工作的安全论证 224
9.7.6 实践 6：分层合规性验证 224
9.7.7 实践 7：事件学习循环与治理更新 225
9.7.8 实践 8：将重新认证纳入常规操作 225
9.7.9 实践 9：由证据驱动的渐进式委托 225
9.8 信任工程模式 226
9.8.1 模式：定义手术室，而非审批每一次下刀 226
9.8.2 模式：安全论证思维 227
9.8.3 模式：无责，但不和稀泥 227
9.8.4 模式：风险分级治理 227
9.8.5 模式：将来源与物料清单提升为一级信任工件 . 227
9.8.6 模式：验证验证者本身 228
9.8.7 模式：自主权是旋钮，不是开关 228
9.8.8 模式：行为可追溯性是信任的硬要求 228
9.9 信任工程反模式 229
9.9.1 反模式：策略剧场 229
9.9.2 反模式：无根之木 229
9.9.3 反模式：权限蔓延 229
9.9.4 反模式：YOLO 模式 230
9.9.5 反模式：将审批当作主要安全机制 230
9.9.6 反模式：声明式合规 230
9.9.7 反模式：错把指导当强制 231
9.9.8 反模式：溜溜球式委托 232

9.10 主要构建模块 232
9.11 衡量信任工程 233
9.11.1 指标：委托边界覆盖率 233
9.11.2 指标：策略违规与未遂事件率 233
9.11.3 指标：平均检测时间与平均控制时间 233
9.11.4 指标：高风险工作的审计与来源完整性 233
9.11.5 指标：合规性验证差异率 234
9.11.6 指标：强制性程序的确定性执行覆盖率 234
9.12 以机器速度进行事件学习 234
9.13 让系统学会自我防卫 235
9.14 用透明度撬动治理 236
9.15 结语：领骑在前，无人轮换 237

10 语言工程：人、AI 队友与机器共通的媒介 239
10.1 编写一旦廉价，阅读即成瓶颈 240
10.2 语言选择：跨越双重模块性的沟通工程 241
10.3 程序理解研究数十年来一直在敲什么警钟 242
10.4 将代码阅读视作核心活动：非新见解，乃新主流 243
10.5 一条可行之路：可扩展的可审计性，而非清教徒式约束 . 243
10.6 为何反应式质量保证在智能体洪流下难以为继 244
10.7 构造安全正成为基线，而非小众偏好 245
10.8 治理救不了错误的基础 245
10.9 关键桥梁：从英语意图到可检验的意义 247
10.9.1 受限自然语言：一座实用桥梁 247
10.9.2 EARS 简明指南 248
10.9.3 为何 EARS 在智能体工作流中变得至关重要 . . . 249
10.9.4 EARS 不是万能药：何时需要其他工具 250
10.9.5 近期论点：携手攀登形式化阶梯 251
10.10 语义评审成熟前，语法仍关键，毕竟人类看的还是它 . . 252
10.11 重复代码变天了：当 AI 能自动传播变更时 253
10.12 面向智能体时代的语言组合拳 253
10.12.1 语言比较：安全性、可评性与工具生态 255
10.12.2 各语言在四个维度上的解读 258
10.13 前路何方：代码成为新的二进制，意义上移一层 265

IV 前进之路 267

你的软件工程 3.0 转型 268

11 你的软件工程 3.0 革命，现在发车：开法拉利的感觉 272
11.1 代码从来不是目标 272
11.2 我们共同构建了什么 273
11.3 愚人的天堂 274
11.4 工程学的本质就是管理不确定性 276
11.5 致开发者：你的学员与队友正翘首以盼 278
11.6 致技术领导者：搭建平台，而非祈祷奇迹 280
11.7 致业务领导者：一场结构性的跃迁 281
11.8 致软件工程教育者与研究者：你们肩负关键使命 282
11.9 最终抉择 284

A 参考表格 292

B 术语表 313

Part I

智能体软件工程与 AI 队友

理解智能体软件工程与 AI 队友

本部分旨在为你打下基础，理解什么是智能体软件工程（软件工程 3.0），以及它如何从根本上改变了传统的软件开发模式。我们会探讨，将 AI 视为并肩作战的“队友”而非单纯工具，究竟意味着什么。我们会看到这些 AI 队友带来的独特优势，也会直面它们引入的那些看似矛盾的挑战。

第 1 章将描绘愿景，区分随性的“氛围编程”与结构化的工程实践，并介绍让智能体协作变得可信赖的核心工件。第 2 章通过具体的交互模式来探讨 AI 队友的优势，这些模式旨在充分利用它们的能力。第 3 章则要直面四个普遍存在的悖论，正是这些悖论，让这些强大的协作者既极具价值，又充满挑战。

这几章共同为你做好准备，让你理解为什么智能体软件工程需要对软件工程体系进行彻底的重新思考，而不仅仅是采用新工具。

1 智能体软件工程：从直觉狂欢到可信工程

开门见山说个残酷现实：傻瓜拿着工具，仍然是傻瓜。

90 年代，我们曾以为给程序员一个 C++ 编译器，就能让习惯过程式编程的人瞬间领悟面向对象的精髓。结果呢？没这回事。21 世纪初，我们又以为买张 Jira 许可证就等于”在做敏捷”。结果呢？照样不是。如今，我们对 AI 正重蹈覆辙。看着 Claude Code、Gemini CLI 这些最新的智能体工具链，就仿佛看到了魔法棒。我们误以为“智能体软件工程”就是运行这些工具链，然后坐等收工。

这哪是工程？纯粹是一厢情愿。

把智能体软件工程简化成提示工程或提示破解，绝非小错。这就像混淆了飞行模拟器和整个航空体系。前者只是个工具，后者却囊括了培训、流程、检查清单、认证、事件响应和审计。只换模拟器，不升级体系，结果绝非更安全的飞行，而是更快的坠毁。

后续所有讨论都基于一个核心论点：智能体软件工程是这样一门学科——它通过让整个软件工程体系在其四大支柱（行动者、流程、工具、工件）上做好准备，从而能从随机性贡献者（无论是 *AI* 还是人）那里，持续产出高质量、可靠、可信的软件。它的目标不是寻找完美的智能体，而是在承认支柱可能以一定概率失效的前提下，借助恰当的约束与证据来构建可靠性。

土木工程每次造桥都秉持同一理念：材料有公差，焊缝会失效，团队会犯错。可靠性并非来自某种神奇钢材，而是源于既有的工程体系——安全边际、冗余设计、检查制度、标准

化实践，以及对失效模式的不懈关注。软件历来由会犯错的人构建。智能体软件工程只不过让这种随机性变得显式：它引入了 AI 队友这类随机性贡献者——它们才华横溢却偶有疏漏，速度飞快但死板教条，效率高超却时而粗心或偷懒。而人呢，一如既往：富有创造力，也会疲惫、犯错，压力之下表现飘忽。明智的应对之道，绝非祈求获得永不犯错的队友，而是打造一个让错误难以发生、更无法隐藏的工程体系。

顶尖的 1% 开发者从来不需要软件工程。他们能把整个项目装进脑子。软件工程是为我们剩下的人准备的。它旨在通过系统化卓越，来遏制个人英雄主义。智能体软件工程不是要把每个开发者都变成孤独的天才，而是通过升级他们周围的系统，将中等水平的开发者提升到能产出 10 倍乃至 100 倍成果的境地。

1.1 氛围编程自有其位，但绝非施工现场

我们必须直面房间里的大象：氛围编程。

眼下正兴起一股风潮，推崇与 AI 进行“氛围”互动：凭直觉引导模型，轻轻推敲，反复生成迭代，直到输出“感觉对了”。这本身没错。对业余编码者，氛围编程打开了创造之门；对专业人士，它相当于数字版的餐巾纸草图，是探索想法、塑造原型、勾勒解决方案轮廓的方式。

但每周都有两个信号越来越强：一方面，工具几乎让任何人都能“氛围编程”；另一方面，智能体开始在一些过去能可靠衡量工程成熟度的任务上显得游刃有余。两者结合，结论不可避免：编码本身，已不再是软件创造的主要挑战。

这正是需要警惕之处。任何发布过真实软件的人都知道，编码在实际工作中占比低得惊人。持久的挑战在于围绕代码的工程实践：那些确保系统在贡献者可能犯错的情况下，依

然能数十年如一日保持可信赖所需的尽职调查、可重复实践与可追溯决策。

不妨把氛围编程想象成微软画图软件。它简单易用，非常适合快速、随意的自由创作。你能迅速画出有用的东西，不拘形式地迭代，几分钟内就能达到“对我够用”的程度。

但你无法用微软画图造桥，也无法在纯粹的“氛围”之上构建真正的软件。

这不是道德主张，而是工程论断。一次性的个人脚本可以随手丢弃，企业级薪酬系统则绝不行。现实世界的软件是团队运动，是漫长马拉松，是一个在首次提交多年后仍须清晰可读的漫长故事。代码一旦写出，其最长且最昂贵的阶段便拉开序幕：*维护*。改变系统最困难的部分，很少是写新代码，而是理解旧代码背后的故事与原理，并知道改动什么才不会让下游的一切崩盘。

这正是过程比结果更重要的原因。最终的代码远不如我们如何抵达那里来得重要：是否恪尽职守？是否有可重复、可信赖的流程？我们是否有解释“为何如此”而不仅是“这是什么”的工件？我们信任麦当劳的汉堡，不仅因为食材，更因为它背后一丝不苟的制作流程。真正的软件复杂得多，它至少应获得对流程同等的尊重。

本书关乎氛围之外的一切。它关乎从艺术工作室到施工现场的转变，关乎将氛围编程的直观速度与软件工程的严谨纪律相结合。我们不仅仅想要“看起来对”的代码，我们想要被“证明可信”的代码，并且这种证明方式能在产出爆炸式增长时随之扩展。

最后这句话至关重要。产出即将爆炸式增长。一个 AI 队友生成代码的速度和并行度，将远超任何人类团队能合理监督的极限。如果你的计划只是“我们多审查几遍”，那么败局已定。审查和测试依然关键，但重心已经转移。大规模的可信度建立在确凿的证据之上，而非确定的步骤。你无法

强求工作完全按你想象的方式进行，你只能要求它在既定约束下，提供证据，满足规范。

这正是大多数团队将折戟之处——不是因为智能体“还没准备好”，而是因为团队从未升级围绕智能体的软件工程体系。

1.2 一门学科，两种模式，各配优化工作台

智能体软件工程这门学科，迫使一种二元性公开化：人本软件工程（以人为本的软件工程）与智本软件工程（以智能体为本的软件工程）。

- **人本软件工程**是人类意图、判断、治理与指导的世界。
- **智本软件工程**则是 AI 队友以机器速度和规模执行任务的世界，并且必须通过工程设计，使其工作可观察、可重现且安全。

这种二元性无关哲学，纯粹出于实用。软件工程的四大支柱（行动者、流程、工具、工件）在两种模式下均存在，但表现形式迥异：行动者不同，流程不同，工件不同，甚至连工作台（工具）也必须不同。

别再把今天的智能体工具链仅仅视为工具升级。应把它们看作新的行动者。这种重新定位，正是“运行工具链”与围绕新型随机性队友构建工程系统的根本区别。

人类的角色也由此变得更加清晰：人类是最终产品的所有者，是设定目标、指引方向的教练。目前，人类还是协调者，通过结构化的循环与交接来调度 AI 队友。假以时日，协调工作本身也将被委派，尤其是在低层级——因为没有任何人类能手动管理爆炸式增长的并行贡献者数量。而工

件，正是随着规模扩大，仍能确保所有权、指导、协调和治理得以实现的机制。

正因如此，那种将单一 IDE 或单一智能体工具链视为软件工程工作中心的观念正在过时。工作台远不止单一工具，它是一个“环境+协议+结构”的整体。它负责将一个混乱、随机、对话式的过程，转变为有纪律的协作。

这就引出了工作台问题——它远不止是个用户界面。

传统 IDE 是为单个开发者编写代码而优化的。它以代码为中心，假设主要任务是实现。在智能体软件工程中，工程师的创造性产出发生了转变：它变成了对意图、约束和指导的阐述，加上对证据的审计，以及对并行工作的 AI 队友的协调。这种演变需要一个指挥中心，而非仅仅一个编辑器。

1.2.1 人类工作台

人类需要一个能减轻认知负荷、增强判断力的工作台。他们需要影响摘要、风险视图和证据包，还需要一个结构化的信箱，而非杂乱无章的聊天流。

一个严肃的人类工作台，应当像一个控制室。它管理着由智能体生成的事件收件箱，包括咨询请求包和合并就绪包。它支持团队级协作，而不仅限于单个人类与单个智能体之间的互动。它让一个人类团队能够与一群共享的 AI 队友进行集体协作。

人类信箱是一个存放需要人类判断和所有权的工件的队列：必须批准的任务简报、在重置和交接过程中保持可恢复状态的连续性数据包、待决策的咨询请求包、待审计和合并判断的合并就绪包、将成为项目记录的决议记录，以及必须在转化为可重用实践前接受审查的指导包更新。

随着吞吐量增长，这些工件越来越像可路由的数据包，而非松散文档：它们拥有稳定的标识符、清晰的所有权、决策

的目标角色、风险层级、生命周期状态和证据链接。工作台正是让这种结构对人类变得可用的地方，它将审查转变为对证明与差异的审计，而非考古挖掘。

人类工作台也是人类有意识路由工作的地方，而不仅仅是审查工作。人类所有者可以将咨询请求包路由给合适的人或 AI 队友（例如安全、数据、基础设施、产品专家），或者将子任务路由给更适合该项工作的另一个 AI 队友（例如专业审查智能体、测试智能体或优化智能体）。路由绝非次要细节——当吞吐量爆炸且无人能处理整个队列时，它是团队规模协调保持高效的唯一途径。

它还支持有纪律的冗余。当风险很高时，应允许多个 AI 队友以最小化的形式手续，独立生成解决方案。它们结果的一致会成为信心信号，而它们的分歧则成为发布前的早期风险警报。

它帮助人类快速把握架构影响。当变更规模很大，或者变更数量很多时（即使每个变更很小），仅看原始差异是远远不够的。如今，许多团队感觉审查是瓶颈。但从长远看，集成才是真正的痛点：一旦许多 AI 队友并行产生变更，限制因素就变成了集成的协调、冲突解决和证据纪律。目前，许多组织尚能部分掌控局面，只因大规模实践智能体软件工程的开发者数量有限，所以即使产出很大，其概念性影响范围和随之而来的实现影响范围也常常有限。工作台需要提供理解视图，显示哪些模块被更改、哪些 API 受影响、哪些数据流被波及，以及风险集中在哪里。它还必须将撰写任务简报、指导包和工作流程运行手册作为一等公民功能，具备完成、版本控制、归档和分析能力——因为这些工件指导着 AI 队友的行为，必须像真正的工程资产一样对待，而我们今天熟知的源代码，则正在变成新的“二进制文件”。

它还需要智能体群组管理功能，因为“一个智能体”绝非未来。协调就是资源管理。表现不佳的 AI 队友会被替换、重

新训练、加以约束或路由到低风险工作。这无关戏剧性，这就是工程。

至关重要的是，人类工作台必须允许用户直接干预。有时编写一个复杂公式比描述它更高效；有时进行精确编辑比反复指导来得更快。工程师必须能够切换到传统 IDE 视图进行那些罕见但必要的精确代码修改，然后返回工作台视图，且不破坏已发生工作的记录。

今天的智能体工具链，包括 Claude Code 和 Gemini CLI，都是朝着这个方向的早期尝试。它们主要存在于人类工作台中，是生态系统演进方向的原型，而非终点。它们离最终目标还差得很远。但方向是明确的：更多的结构、更丰富的信箱、更清晰的协议，以及为协作模式设计、而非简单附加到单个智能体上的工作台。

1.2.2 智能体工作台

AI 队友得用另一套工作台。它们擅长确定性检查、稳定接口、机器可读反馈和快速内部循环。它们要的是结构，不是长篇大论。还得有个信箱，专门接收清晰、可审计的输入和输出。

智能体信箱里装着执行单元、任务和子任务，这些都源自任务简报，并受工作流程运行手册管理。同时，它还包括约束和引导执行的各类工件：作为行动规范的任务简报；作为必需流程、可随时拉取并在任务简报中直接引用的工作流程运行手册；同样可按需拉取、直接引用的可复用指导包；作为权威结果的决议记录；用于在重置和交接时保存可恢复状态的连续性数据包；以及当运行手册触发升级或提交点时所需的产出，比如咨询请求包和合并就绪包。

别逼智能体用人用的工具。人类需要降低认知负荷的界面，智能体可不需要。它们在原始、低开销、为计算效率优化的工具上如鱼得水，还得配上结构化的、机器可读的反馈。这

意味着，得为智能体量身打造执行环境，而不是照搬人类那套。换个角度看，AI 队友能同时跟踪 1000 个断点，而人类呢，顶多两三个！

智能体原生工具长得就不一样。它们有语义搜索工具，返回结构化结果，而不是空话连篇；有结构化编辑器，把代码当语法树操作，而非纯文本；有调试工具，能探索巨大状态空间而不崩溃；还有监控基础设施，能揪出漏洞、标记意外高成本、修复损坏环境，并自动替换失败的沙箱。目标很明确：只有需要人类战略干预的大问题，才会上报到人类工作台。其他一切，都该在智能体工作台内部消化。

正因如此，语言和工具链的护栏必须收紧。富有表现力的反馈，人类听着舒服，对 AI 队友来说，更是立即可用的训练数据和指导素材。与随机性贡献者合作时，严格的安全保证和丰富的反馈能及早揪出幻觉。编程语言基础则成了安全系统的又一道防线，确保软件可信。

1.3 工件即接口

智能体软件工程的核心，是从“随意聊聊”转向“按章办事”。同一个理念反复出现：意图要结构化、执行要结构化、证据要结构化、连续性要结构化、指导也要结构化。本章接下来就给承载这些结构的具体工件起名，让抽象理念落地。

这些可不是文书工作，而是控制面板。它们决定了意图如何转化为实际工作，工作又如何变成可信的变更。

本章贯穿始终的实用框架是：我们通过工件与智能体交互，方式有两种。其一，非正式交互，就像和人类队友头脑风暴：探索选项、提出问题、勾勒约束、发现未知。其二，正式交互，通过可治理的工件：一些主要用于提出请求（委托和交付），另一些则用于治理（控制行为、强制执行流程、让可信度可审计）。智能体软件工程的关键不是禁止非正式

协作，而是确保风险升高时，非正式协作能结晶为明确的、版本化的结构。

1.4 智能体软件工程协调工件及其用途

在经典软件工程中，协调主要靠**代码形态**的东西：差异、测试、工单、架构决策记录和运行手册，辅以人与人之间的沟通。而在**智能体软件工程**中，我们得引入一套主要是**自然语言形态**的工件。

这些工件充当了人与智能体之间、以及智能体彼此之间交接任务、在不同程度人类监督下协作的接口。它们绝非官僚主义，而是我们实现以下目标的利器：

- **明确意图**，确保整个系统协调一致。
- **路由不确定性**到合适的人或机器解决者。
- **让可信度能以机器速度审计**。

本章有个贯穿始终的重要区分：人类编写协调系统，而智能体生成任务工件。模式不会“改变”智能体已经生成的工件，它改变的是人类编写的规则和关卡——这些规则规定了智能体接下来必须生产什么，以及什么才算合格。当模式后文提到“将此编码在 X 中”，意思是：把它编进那些控制和约束智能体产出、由人类掌控的工件里，并要求智能体在其生成的工件中包含特定部分。

然而，随着 AI 队友能力进化，我们将从这种人类强控制的模式，转向由 AI 队友受托编写原始工件的状态。在这种成熟状态下，AI 队友将接管人类委托给它们的监督角色。这种委托是层次化的，当前的 AI 队友会把子任务再委托给其他智能体。为了本章清晰起见，我们简化为“人类”与“智能体”两种角色，但这里“人类”更准确的描述是任何处于监督位置的实体，无论其本身是人还是 AI 队友。

不妨把以下内容当作心智地图：

- **任务简报（人类编写，针对特定任务）：**规范的工作定义——目标、非目标、约束、概念性计划（策略和检查点，而不是脆弱的步骤）、精选的上下文指针、基于属性的验收标准、自主权边界，以及证据义务（最终必须提供什么证明）。它防止智能体自行脑补填空，并赋予其明确的、有边界的意图。
- **连续性数据包（有界形式的连续性）：**连续性数据包是任务连续性的有界形式，团队可以根据管理构建块扩散的偏好，用两种方式实现。一些团队将其保持为单独的构建块，在自然重置点（如交接、上下文压缩、智能体更换、每日结束时）更新，避免任务简报膨胀；另一些团队则将其折叠进任务简报，作为一个“当前状态”部分，并积极压缩。无论哪种，“会话”指的是拥有稳定工作集的执行片段，以上下文重置、智能体交换或显式交接点为界，而非神秘的时间概念。功能都一样：保留对恢复关键的内容，并明确列出死胡同，以免系统重蹈覆辙。
- **指导包（人类编写，长期存在）：**机构规则手册——本地约定、边界、停止规则、术语表、上下文管理规则，以及这里“做得好”的标准。它把一个教训变成持久的行为准则，减少重复指导。
- **工作流程运行手册（人类编写，长期存在或任务特定）：**执行协议——一套带有关卡、检查点和必备产出的标准操作程序。它能编码分解和所有权协议、触发咨询请求包的升级条件、证据生产步骤，以及分层就绪要求（例如，团队级的“合并就绪”与“集成就绪”）。它把可信度变成一条流水线，而不是听天由命。

其余工件通常由智能体生成，但它们是在任务简报、指导包和工作流程运行手册所施加的要求下生成的：

- **咨询请求包（智能体生成，在升级时）：**结构化的“需要决策”数据包——决策点、选项、权衡、风险、证据、建议，外加路由元数据（目标角色、风险等级、紧急性、生

命周期状态）。它防止智能体在边界上瞎猜，强制将升级变为可审查、可路由的形式。

- **合并就绪包（智能体生成，在交付时）：**交付数据包——变更内容、原因、影响范围、已运行的测试、证据、计划台账（对比概念性计划与执行计划，附差异和理由链接）、探索档案（渐进式披露）、证据/探索构建块的机器可读清单、影响面、回滚计划以及未决风险。它让“完成”无需考古就能审查。
- **决议记录（持久，风险分级作者身份）：**对特定升级或合并审查的明确、版本化决议——决定了什么、为什么，以及由此产生的新约束。它链接回触发数据包，并将教训纳入持久治理：当决议建立新规范时，就更新指导包；当它揭示缺失的关卡或所需证据时，就更新工作流程运行手册；当它澄清重复出现的约束时，就更新未来的任务简报模板。

本章重点介绍上述核心工件集。后续章节将深入探讨其背后的控制规程，以及 AI 智能体舰队规模的扩展（如更丰富的数据包路由、作为标准审查面的计划台账，以及作为协调机制的分层就绪）。

- **核心层（先读此章节）：**心智模型——智能体软件工程的双重性（人本软件工程 vs 智本软件工程）、四大支柱（行动者、流程、工具、工件），以及**大规模信任是靠证据和执行设计出来的**这一理念。本章关键术语总结于**附录 B 智能体软件工程术语表**。
- **参考层（实施时使用）：**表格中的详细工件模式。它们有意写得详尽，但**不是核心信息**。它们是入门模板和思考提示，并非金科玉律。如果您想要完整的逐字段版本，请参阅**附录 A 参考表格**。

如果您是来了解“为什么”的，请顺着叙述主线走。如果您是来学“怎么做”的，等您准备好实践智能体软件工程时，再翻到附录不迟。

1.5 结构化意图：任务简报

在智能体软件工程中，委托的基本单位是任务简报。

任务简报可不是“一个更好的提示”。它是一份行动规范：有版本控制、可测试，专为 AI 队友消费设计。它让意图明确、护栏可见、验证无从逃避。同时，它也防止上下文过载。把整个厨房水槽都倒进 AI 队友的基础模型大脑，只会增加成本、降低性能。任务简报精选上下文，让其唾手可得。

任务简报也是两种交互模式的绝佳例子。团队通常从非正式协作开始——头脑风暴约束、风险和方法——然后将这种探索转化为精确、有边界的工作指令，好让智能体能在不自行发明规则的前提下执行。

任务简报还明确了两项团队常常拖到为时已晚才想起的要求：

- 自主权边界：AI 队友能决定什么、必须升级什么（以及向谁升级），以及绝对不能碰什么。
- 证据义务：最终必须拿出什么证明，而不仅仅是“代码看起来没问题”。

任务简报通常包括：

- 目标和意图
- 概念性计划（策略 + 检查点，而非脆弱的步骤清单）
- 成功标准（通常表述为属性/不变量）
- 精选的上下文指针
- 实施指南（偏好和“请勿触碰”的边界）
- 验证计划和证据义务
- 升级和权限（自主权边界）
- 版本管理和汇总信息

有关完整模式，包括“每个部分为何存在”和“缺了会怎样”，请参阅**表 1**。

任务简报不是僵化的一次性合同。它会演进。可以从轻量级开始，随着约束澄清和证据期望明确而逐渐丰富。但它始终应保持可读、可检查。重点不在篇幅，而在精确。

连续性数据包通常包括：

- 当前状态快照
- 关键决策和约束
- 未决问题和后续步骤
- 已探索的死胡同和遭拒的方法
- 证据指针
- 交接元数据

关于其各个部分的详细解释，请参阅**表 1A**。

1.6 结构化证据：合并就绪包与决议记录

如果强迫人类以机器速度审查未经整理的拉取请求，倦怠是必然的，垃圾代码照样发布。人类的认知负荷必须聚焦在审计结构化的证据包上。

直呼其名吧：合并就绪包。

合并就绪包不是“一个描述漂亮的拉取请求”。它是一个证据捆绑包，旨在证明 AI 队友的工作真的具备了合并条件。

随着团队规模扩大，“就绪”的定义变得分层：代码写完不等于合并就绪，合并就绪也不总等于集成就绪或“合适的发布时机”。第 1 章不深入探讨集成调度，但合并就绪包的形态应能支持后续演进：它必须以可路由、可审计、可排序的方式承载证据和决策轨迹。

合并就绪包通常包括：

- 范围到证明的映射（任务简报成功标准 → 检查项 → 证据）

- 验证包（测试、日志、新增测试、边缘案例）
- 工程卫生证据（代码检查/静态分析、变更一致性）
- 理由和权衡（为何选此方案；其他选项为何不行）
- 计划台账（概念性计划 vs 执行计划；差异及理由链接）
- 探索档案（渐进式披露）
- 证据/探索构建块的机器可读清单
- 带有渐进式披露的审计跟踪（链接到治理工件及工具输出）
- 风险和发布计划（影响范围 + 回滚方案）
- 合规/访问/保留说明
- 集成就绪/时机说明（如适用）

表 5 展示了一个详细的合并就绪包示例。

结构化审查需要一个结构化的结果：决议记录。决议记录将决策链接回证明其合理性的工件，记录被采纳的内容，并成为持久项目记录的一部分。没有它，审查就是一次性事件；有了它，审查就变成了机构记忆。

1.7 结构化升级：咨询请求包与决议记录

在正经的软件项目里，实现往往演变为治理。当 AI 队友碰到决策边界——比如模式变更、安全影响、向后兼容性、性能风险或操作约束时，正确的应对可不是让它闷声不响地快速操作。正确的应对是：升级。

而且，这种升级必须有章可循。

一份咨询请求包要用一句话点明决策关键，概括相关背景，列出选项与权衡，附上证据，说明影响范围，最后以清晰的问题收尾。规模上来后，它还得包含路由元数据，好让决策者无需开会就能直达目标：目标角色、风险等级、紧急程度、生命周期状态以及持久链接。

咨询请求包还得让升级可追溯，作为工作流合规的一环：当升级由工作流程运行手册的门控或触发器触发时，这份包必须明确引用那个触发器，这样团队才能审计升级是否事出有因、时机恰当。

但升级只是故事的一半。如果结果只停留在聊天记录里，那迟早会丢。所以，结构化升级必须以一份决议记录收尾：这份持久、可审查的决策记录，会链接回咨询请求包，记下批准了什么、有何约束、证据何在。没有决议记录这个工件，团队就会反复争论同一个问题，重复同样的教训，一次次踩进同一个坑。

来看一个具体的咨询请求包示例。任务是“添加按相关性排序”。实现过程中，智能体发现需要一次模式变更来支持新的索引字段。

决策陈述：“我们是新增一个索引字段来支持相关性，还是动态计算相关性?”
最小上下文：受影响的表、查询路径、当前延迟预算、索引所有权、上线约束。
选项与权衡：
选项 *A*：模式变更，附带迁移、回填策略和分阶段上线。
选项 *B*：读取时计算并缓存，无需模式变更，但 CPU 风险更高。
证据包：选项 B 的快速基准测试、选项 A 的估算和风险清单、测试影响说明。
影响范围与回滚：可能破坏什么、谁依赖这条路径、回滚计划。
建议与清晰问题：“批准选项 A，附带分阶段上线和回填计划；还是要求在下个发布窗口前改用选项 B?”

咨询请求包的典型部分（路由元数据、决策陈述、最小上下文、选项、证据、影响范围、清晰问题）在 **表 4** 中有详细说

明。决议记录的典型部分（决策、理由、约束、批准、链接证据、回滚到运行手册/指导、替代关系）在 **表 4A** 中展示。

1.8 结构化指导：指导包

人类指导往往转瞬即逝。一次代码审查评论，通常只让一个人学一次教训。但在智能体世界里，这种短暂的指导成本高昂，因为错误会以机器的速度反复出现。

所以，指导必须升级为头等工件。说白了，就是把指导当成代码来写。

实际做起来，这意味着需要一本结构化的规则手册，用来捕捉团队规范、架构指南、首选模式以及上下文管理规则。它可以简单如“所有新模块必须暴露接口并配备测试”，也可以具体到“避免废话注释；只注释设计理由”，或者“没有安全审查记录，绝不添加依赖”。它还能定义如何处理上下文：预算、优先级语义、检索规则，以及出现矛盾时该怎么办。

很多团队已经通过类似 CLAUDE.md、AGENT.md 或工具/领域特定规则文件（俗称技能）来近似实现这一点。概念没错，但实现还不成熟。核心思想是：在 AI 队友开始任务前，它会加载项目的制度性知识或团队的部落智慧。这减少了冗余指令，提升了一致性，把“我们上个月吃的亏”变成团队持久的本事。

指导即代码需要一个持续的反馈循环。这里有两种指导形式很重要：显式指导是直白的规则；推断式指导则是 AI 队友收到修正后自己总结的原则。如果人类为了可读性重构了代码，AI 队友不该只是接受更改。它应该用大白话提炼出一条通用原则，让人批准后加入指导包。这才是队友真正的进步，而不是应付了事。

指导包是可复用的指导，编写时就预备在需要时引入。有时它很通用，就像团队把“我们这儿怎么构建”写成法典。有时它针对特定任务，就像团队把“这类变更该怎么做”编成手册。无论哪种，目的都是让 AI 队友在执行时直接应用，通常就在任务简报里引用，确保正确的指导在射程之内。

指导包最深层的角色不是强行管制，而是能力塑造：引导智能体理解在这个组织里，“优秀”长什么样。这本质上是非确定性的。它更接近对潜在空间进行条件调节，而不是死板的检查清单。这不是缺点，反而是传授品味、权衡和质量的合适工具。但这确实划下一条硬边界：如果某个最佳实践必不可少，光靠“指导”是不够的。基础实践必须通过工作流程运行手册的门控、命令或触发器来强制执行——这些机制会确定性地提醒智能体、要求证据，不达标就喊停。指导可以在概念上调用这些提醒（比如“验证一切”），但工作流程运行手册才是让期望变成可靠触发的工作流步骤。

还有一个更微妙但更重要的边界。指导应该定义成功的样子，而不是规定怎么思考。这细微差别划清了自动化与真正智能体工作的界限。如果你编码“如何调试”“如何分析”或“如何设计”，那是在自动化现有流程——相当于让 AI 队友照搬你的思维套路。但如果你编码目标、约束和质量标准，那你是在委托结果，让 AI 队友自己推理方法。前者把队友框在人类设定的思维模式里，限制了潜力。后者则让队友探索你可能没想到的路径。所以，编码偏好和边界，别编码认知策略。

指导包通常包括以下部分：

- 地图（系统概览和边界）
- 工程意图（“为什么”和不可妥协项）
- 操作手册（这儿的工作方式）
- 上下文工程规则（预算、检索、矛盾处理）
- 治理与安全（决策权、升级、处理机密）
- 生命周期与引用（版本控制、弃用、退役规则）

这些部分的详细说明见 **表 2**。

1.9 结构化执行与编排：工作流程运行手册

光有结构化意图和可复用的指导还不够。在企业规模上，关键是结构化执行。

工作流程运行手册就是干这个的。

工作流程运行手册是一个可复用的工作流组件，定义了活必须怎么干。它是开发循环的标准操作程序，带有明确的门控。有些门控是自动且确定的，有些需要人工批准，还有些由智能体驱动。真实的循环多半是混合的。重点不在于建议，而在于强制执行组织要求的执行路径。

这也让工作流程运行手册自然成为连接步骤的协议决策归属地：智能体何时该简短回应、何时该详细展开，何时必须生成多个选项并在推荐前附上批判性分析，何时必须停下请求批准，以及每个阶段允许多少思考或工具使用。这就是让工作流主干变得明确的方法。同样，当情况重要时，这也是将不同阶段路由到不同模型和工具层级的手段。

如今的智能体工具链通过“命令”来操作工作流程运行手册。一个命令可以是完全确定的、部分确定的，或者完全不确定的。命令可以通过钩子或人工操作确定性地触发，也可以通过“技能”非确定性地触发，甚至能在人类与 AI 队友聊天时通过纯英语隐式触发。

在团队规模上，工作流程运行手册通常会扩展，包含超出本章深度的协调协议，但值得先提一嘴：分解和所有权约定、咨询请求包协议、集成调度以及分层就绪门控。

在舰队规模上，工作流程运行手册还能编码多智能体流水线：工作从一个 AI 队友流向另一个 AI 队友的序列，带有结

构化的交接和审批点。人类不用手动编排每一步（“现在你写代码，现在你审查，现在你测试”），而是一次性编写好流水线。然后流水线自主执行，智能体按设计互相交接工作，人类只在预设的审批点介入。这是在 workflow 层面而非任务层面的委托，是组织在不线性增加人类注意力的前提下扩展 AI 队友吞吐量的方式。协调工程章节会深入探讨流水线工程。

工作流程运行手册通常包括：

- 入口门控（必需的上下文/环境检查）
- 阶段与门控（命名的检查点和停止规则）
- 确定性检查（测试/扫描/lint/构建步骤）
- 探索策略（替代方案和决策规则）
- 分解与所有权协议
- 升级触发器（需要咨询的内容；路由元数据）
- 必需输出（必须生成哪些包）
- 分层就绪与集成调度（如适用）
- 计划台账与分歧规则
- 命令与触发器（始终在线的强制执行主干）
- 可追溯性与可解释性

更多细节见 **表 3**。

1.10 信任需要强制执行

错误总会发生。

随机性贡献者会疏忽。人类会跳过步骤。截止日期会带来压力。如果信任只靠每个人自觉遵守纪律，那这系统根本不可信。

可信赖的软件工程需要合规性验证。它需要实实在在通过的门控，而不只是嘴上说说。它需要可以审计的证据，而不是空口无凭的保证。

这就是为什么许多智能体工作流在实践中掉链子。它们依赖指南，然后指南被无视时就傻眼。人类也这样。随机性 AI 队友只会犯得更快。工程上的回应不该是抱怨容易出错，而是确保我们强制执行这些门控，并把证据变成必需产出。

1.11 工件是新的工程层，而非定制化宏

现在来个更让人不适的升级：这些工件本身就是软件工程系统的一部分。

它们不像你个人收藏的编辑器宏。在多数正经环境里，它们不能那样。它们是一个必须被工程化的基础平台层。就像团队学会了工程化 CI 流水线、GitHub Actions、Docker 容器和部署工作流一样，团队今后也不得不工程化这些工件。

这意味着它们需要修 bug、需要重构、需要优化。它们需要被复用、抽象和版本控制。模式会出现，冲突会浮现。两个指导包可能打架。工作流程运行手册在平台变更后可能过时。决议记录模式可能不再适用。所有这些都很正常。

所以，所有权很重要。

有些工件归人类所有，因为它们定义了意图、治理和项目记录。最终的任务简报、最终的工作流程运行手册、维护中的指导包以及决议记录，必须有明确的人类所有者。AI 队友可以起草、改进、提议更改，但人类握有最终接受权。

其他工件天生该由 AI 生成，因为它们是以机器速度汇编的证据。咨询请求包、合并就绪包、机器可读清单和探索存档，正是 AI 队友该产出的东西。它们减少了人类阻力，同时增加了严谨性。

协同创作在这里也变得顺理成章。AI 可以起草任务简报、提议运行手册、汇编合并证据、提炼咨询请求，还能从审查评论中提出指导更新。AI 甚至能挖掘项目历史，复原候选

模式：过去的事件、PR、辩论、回滚。成熟的代码库本就蕴含这些原材料。AI 让提炼它们成本更低。

但协同创作要奏效，前提是这些工件被视为工程资产。它们必须版本控制、审查、协作维护。它们必须为复用而设计。冲突必须检查。必须像对待任何平台层一样优化它们。否则，项目就会积累一种新的技术债务——工件债务，并且会因与旧系统同样的原因失败：无人负责的复杂性。

1.12 可信性即代码：确定性、强制执行与流程即代码

这些工件事关重大，背后有个更深层的理由：它们能把软件工程里始终纠缠不清的两大杠杆，彻底分开。

其一，塑造判断力。这正是指导包的用武之地。它能预先调教智能体，让它对设计质量、测试标准、权衡取舍乃至何谓“优秀”，形成一种本能直觉。其高明之处，恰恰在于它不是一份死板的检查清单，而是把智能体推向正确的思维模式。

其二，工作流强制执行。这便轮到工作流程运行手册登场了。它负责串联步骤、划定检查点、触发升级、强制生成证据，并在明确的关卡引入人工干预。它让你能用代码规定：哪些阶段必须板上钉钉，哪些可以保留些随机性。还能编码规定每个阶段该花多少心思、调用哪些工具，乃至使用哪个层级的模型或工具。

实际问题在于，“按需应用最佳实践”根本行不通。把每一条好规矩都绑在每次交互上，只会让人疲于奔命、漏洞百出、标准不一。人会忘事，智能体会疏忽，系统则悄然崩坏。

智能体软件工程为团队打开了新局面。过去，我们主要对代码（比如整洁代码）或二进制文件（比如 SBOM）强制执行规则。对流程却往往束手无策——流程多半是社交活动，难以审计。如今，流程变得可编程了。关卡可以明文规定，

触发器能够记录在案，各种数据包可以作为强制产出，合规性也能验证。流程成了代码，自然就能被清晰、仔细地监控。

但这绝非“规则越多越好”。多不等于妙。要是把每本软件工程典籍都塞进指导，冲突就在所难免。冲突导致模糊，模糊滋生不确定性，而不确定性恰恰是我们试图大规模管控的东西。我们的目标，是经过精心策划、始终如一的治理：一套能以最小规模带来最大可信性跃升的持久规范，在关键环节通过工作流主干强制执行，在涉及品味和判断时则通过指导来传授。

工具的短板也在此暴露无遗。现有系统缺乏强大的流程合规支持（强制执行、验证和调试）。命令和触发器通常是个性化设置，并非为协同设计，文档质量也参差不齐。然而方向是明确的：随着团队扩展智能体软件工程，平台必须支持冲突检测、触发器追溯、以差异为核心的审查界面，以及解释命令为何执行或未执行的可解释性。没有这些，治理便无从扩展。

1.13 本章主线与未来之路

本章可以用一句话概括。

智能体软件工程就是结构化工程：用任务简报结构化意图，用工作流程运行手册结构化执行，用连续性数据包在重置和交接中保存可恢复状态，用咨询请求包在团队协作中结构化升级，用合并就绪包通过证据和清单结构化审查，用决议记录结构化项目历史，用指导包结构化学习。

结论再直白不过。

如果一个团队用散装实践来应对智能体吞吐量，它扩散错误的速度将远快于创造价值。这叫“拿着智能体工具链的傻瓜”。

另一条路，是智能体软件工程。这条路把约束和证据内建于系统，而非事后打补丁。在这里，工件像软件一样被维护，合规性像安全一样被验证，可信性被视为工程的核心产出。

接下来的章节将这条路具体化，把工作框定为三个反复出现的核心问题，它们应当塑造在此系统中诞生的每一个工件。

- 第一，**发挥 AI 队友的长处**。编写能利用其速度和并行性，同时不牺牲清晰度的简报和工作流。
- 第二，**防范系统短板**。设计运行手册、咨询流程和合并提交，让故障模式在生产前就暴露无遗、有界可控且易于纠正。
- 第三，**为 AI 队友赋能**。提供一个具有确定性反馈和快速内循环的智能体工作台，让它们能以超高速度运转，无需把人类拖进每个微步骤，同时仍能遵从治理并产出证据。

AI 队友已是大势所趋。现在，是时候为它们、围绕它们构建系统了。化炼金术为工程，变氛围为证据，转混乱为信任。是时候在 AI 队友时代，重新定义软件工程这门学科了。

软件工程谱系：氛围编程 → 氛围工程 → 智能体软件工程

氛围编程求快，氛围工程求稳，智能体软件工程则求大规模可信。

每周都有两个信号越来越响：氛围编程正变得触手可及，AI 智能体正在那些以往需要工程成熟度的任务上驾轻就熟。结论很简单：写代码不再是瓶颈。瓶颈在于工程可信性——可重复的实践、可追溯的决策、随系统演进仍清晰如初的成果。理解这种转变，不妨看看三个阶段。

微软画图与 Photoshop 之喻

- 氛围编程好比微软画图——快捷、直观，最适合天马行空的探索。
- 智能体软件工程则是 Photoshop——强大、严谨，为严肃工作而生。
- 氛围工程则是那个缺失的中间层——“微软画图 Pro”：依旧容易上手，但装上了护栏。

1. **氛围编程 (微软画图)**
   最适合原型、一次性脚本和个人自动化——这里试错成本低，迭代就是一切。
   完成标准是：“对我管用就行。”
2. **氛围工程 (微软画图 Pro)**
   氛围工程保持了氛围编程的速度，但增加了实现持久成果所需的最小结构：
   - 可靠性——边缘情况不会立马搞垮它
   - 基本安全卫生——常见的“坑”更难踩到
   - 可维护性——别人能安全地扩展它
   - 保存意图和原理的轻量级工件

完成标准是：“它管用，我能说清道理，而且它能演进。”

3. **智能体软件工程 (Photoshop)**
   当吞吐量爆炸式增长而可信性必须守住时，智能体软件工程便登场了。当 AI 队友并行产生变更时，瓶颈变成了治理、证据和集成——而非敲键盘。因此，这门学问从“获取代码”转向“工程化信任”：
   - 划定意图与约束的边界
   - 通过可重复的工作流和关卡执行治理
   - 在决策边界进行结构化升级
   - 基于证据的审查，而非差异考古
   - 持久的决策记录，而非丢失的聊天历史

   完成标准是：“它在约束下满足要求，并提供可审计的证据。”

2 发挥 AI 队友的力量

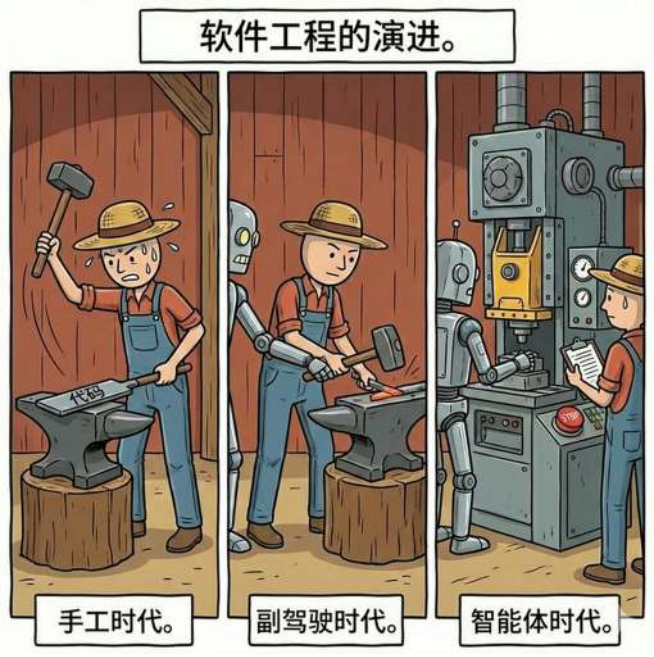
2.1 自动化阶梯：工具变身队友

软件工程的不同时代，需要不同的工程姿态。我们得先捋清楚。

- **软件工程 1.0 是手工时代**。代码全靠手敲。确定性高，速度慢。一人、一键盘、一个输出流。
- **软件工程 2.0 是副驾驶时代**。自动补全进化成大语言模型（LLM）辅助补全，GitHub Copilot 是这时期的标志。LLM 固然强大，但人类仍牢牢坐在驾驶位，掌控每个循环。这时的姿态，还是代码优先、人类注意力优先。
- **软件工程 3.0 是智能体时代**。AI 不再只是工具，它成了能规划、执行、提问、升级、提交的队友。这可不是比喻，而是切切实实的功能性现实，它从根本上改变了工作的基本单元。随着输出爆炸式增长，人类注意力成了稀缺资源。人类的角色转向定义意图、设定约束、保持问责；而智能体的角色，则是在这些边界内专注执行，并通过端到端的工程系统，持续生成证据来提供支持。

我们几十年前就明白，自动化并非一个可以随意拧高的旋钮。它是一系列关乎“谁决策、谁行动、谁监督”的选择，而这些选择会催生不同的故障模式。

谢里登（人因工程学家）和弗普朗克（人机交互学者）将其框架化为“自动化阶梯”的不同级别，后来帕拉苏拉曼（人因工程学家）、谢里登和威肯斯（人因工程学家）的工作**进一步阐明**：将工作从人类转移给自动化系统，并不会消除风险，只是转移了风险。自动化程度越高，人类的工作通常就从执行转向监督，而监督本身代价不菲。它会导致经典问题：丧失情境意识、过度信任自动化输出、对系统漂移反



应迟缓，以及长时间静默运行后突然要求人类干预时，交接变得异常脆弱。

软件工程正经历同样的转变。从副驾驶转向智能体，意味着我们从亲手操作转向监督控制，而监督自带一套故障模式。在软件工程 2.0 中，副驾驶主要是个动手的键盘助手。人类仍处于微观循环中：代码生成时亲眼看着，逐行接受或拒绝，人类注意力是主要的安全闸。到了软件工程 3.0，工作单元变了。智能体可以规划并执行多步修改、运行工具、编辑多个文件，最终拿出一个看似完工的结果。人类的角色从打字和局部审查，转向把控意图、约束和治理。唯有当系统能让人类持续接收到正确的信号时，这种转变才算真正赋能。否则，人类会逐渐沦为橡皮图章，一旦出事，被仓促拉回现场，却因情境意识太少而无力回天。

正因如此，智能体软件工程必须作为一个完整的软件工程系统来构建，它立足于四大支柱——行动者、流程、工具、工件，并以两种模式运作。只换新工具而不升级其他部分，

是导致自动化失败和信任崩塌的最快路径。本章特意聚焦于“行动者”这根支柱：把 AI 智能体视为队友究竟意味着什么？人类又该养成哪些交互习惯，才能可靠地发挥其长处？

一旦将智能体视为队友，我们人类便立刻肩负起三项责任：

1. 发挥其长处，以获取复合增长的生产力与质量。
2. 保护工程系统免受其短板影响，以免输出变得一团糟。
3. 为它们赋能，助其表现出色，从而让我们自己跳出微观循环，让改进效果产生复利。

这可不是管理空话，而是应对“随机性贡献者”的工程化响应。本章专注责任（1）。第 3 章对付责任（2）。第二部分则全面应对责任（3）。

2.2 杠杆效应：智能体软件工程改写劳动力经济学

智能体软件工程改写了劳动力经济学。当多提一个请求的边际成本趋近于零时，三种成本同时骤降：金钱、时间和社会摩擦。这就是为什么许多优秀工程实践从未因“错误”而被拒，它们只是**被认为“不值当”**——因为人类在经济上根本不可能长期坚持。太慢、太贵、太无聊，社会摩擦还太大。

瓶颈也随之转移。它从“我们能不能写出代码”，更多地转向“我们能否协调并行工作、审计证据，并安全地进行大规模合并”。

新技术常把一个学科里我们曾不得不放弃的东西，重新带回来。软件工程也不例外。正如罗纳德·科斯所预言：当交易成本变化，经济学随之改变，那些技术上正确但过去负担不起的实践，重新变得可行，并作为默认设置回归，而不再是需要英雄壮举才能实现的例外。

结对编程就是个简单例子。几十年的实践与研究都表明它能提升质量，极限编程也在工程文化中普及了它。但管理者往往抵制，因为这感觉像为一份工作付两份薪水。在智能体软件工程中，结对几乎免费。一个 AI 队友可以观察每次击键，实时建议重构，甚至在代码保存前就捕捉到错误。它的优势不光是速度，更有耐心。它能连续数小时专注任务，没有自我、不会无聊、也无社会摩擦。

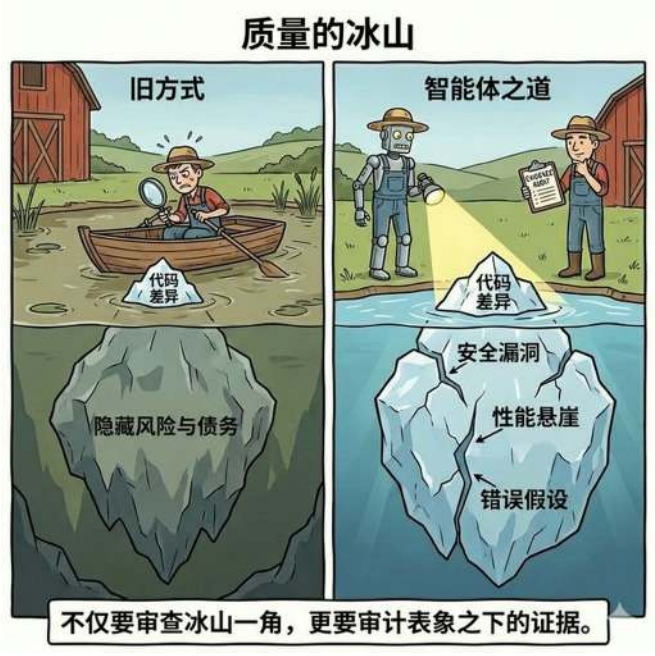
一次性原型设计是另一项我们能重新负担得起的实践。布鲁克斯早就告诫我们“计划扔掉一个”。但我们常常忽略，因为写代码成本高，且工期紧迫。智能体改变了这个约束。你可以在提交前探索五个原型并全部丢弃。这不是图方便，而是质量杠杆。当探索变得廉价，你就能搜索解决方案空间，而非死磕第一个想法。

严谨性也变得便宜。契约、不变量、边界情况、测试脚手架和质量检查，如果由系统强制执行，就能从“最佳意图”变为“默认行为”。从霍尔（C.A.R. Hoare，形式化验证先驱）、迪杰斯特拉（Edsger Dijkstra，结构化编程奠基人）到梅耶（Bertrand Meyer，契约式设计提出者），再到兰波特（Leslie Lamport，分布式系统与 TLA+创始人）的系统思维，形式化方法和基于契约的开发在智力上一直很有吸引力。团队之所以避开，是觉得其开销与快速交付格格不入。智能体在那些繁琐的严谨性环节表现出色：枚举边界情况、起草契约、生成脚手架并保持其一致性。

测试驱动开发（TDD）情况类似。人人都说自己在做 TDD，可工期一紧，几乎没人真做。智能体让流程翻转变得容易：先表达意图和约束，让 AI 生成测试，再去实现。验证不再是你总想推迟的杂务；如果端到端工程系统要求，它可以成为生成的副产品。

深度质量检查是最后一项我们能重新负担起的实践，它将把严肃的组织与混乱区分开来。曾几何时，像法根检查（Fagan Inspection，由 Michael Fagan 提出的系统化代码审查方法）

这类严格流程能产出极其可靠的软件，但进度慢如蜗牛。我们牺牲了那种严谨性来换取速度。智能体把速度还给了我们，却不必强迫我们放弃严谨性——前提是，我们重新设计审查单元。人类不应把注意力浪费在以机器速度扫描原始代码差异上，而应聚焦于审计证据、发现风险，以及做出需要判断的决策。



机会就在这里：追求卓越变得足够便宜，足以成为系统性标配，而无需英雄主义。

这种转变非常具体，立刻体现在团队过去视为奢侈品的两方面：**反复打磨**和探索。当劳动力廉价且无社会摩擦时，你可以毫不尴尬地多次迭代，也能并行尝试多项方案，而无需把协调本身搞成一个项目。关键区别在于，你现在可以这么做，却不必担心通常的成本超支和进度延误。你不是在耗费人力去探索或**反复修改**，而是在消耗计算资源和一点点注意力。这意味着你可以提前提升质量、降低决策风险，而不是事后为意外买单。

本章介绍的交互模式，正是要将廉价的**反复打磨**和廉价探索，转化为可重复的习惯，从而提升决策质量，以更快、更经济的方式交付可信的变更。

2.3 本章结构：以优势与交互模式为核心

持怀疑态度的技术读者不需要鸡汤，他们需要可以教授、重复和执行的具体行为。本章围绕“优势”来组织，并将其表达为可实践的“交互模式”——这些可重复的微习惯，你能教会团队、反复使用、严格执行，直到卓越成为默认行为。

每一种新媒介都会改变团队的能力，同时也会改变团队放弃什么、什么会回归，以及新能力被滥用时会出现什么问题。麦克卢汉的“四元律”是强制实现这种平衡的简洁框架。我们在“能力集群”层面应用四元律。针对每个集群，我们提出四个问题：这种能力放大了什么？它**挤占了**什么？它把什么带了回来？以及它可能如何反转成负担？重点不是畏首畏尾，而是在获得杠杆效应的同时，避免后期因混乱、系统漂移或集成痛苦而付出代价。

为了厘清 AI 队友擅长什么与如何利用它，我们将 AI 队友的每项独特优势映射到一个“集群”。每个集群遵循相同的剖析流程：

1. **优势是什么：**具体描述该项优势。
2. **解锁了什么：**这项优势如何改变成本结构，以及当它标准化后，什么会变得便宜到足以成为系统性实践。
3. **四元律速览：**总结该能力在放大、**挤占**、找回、反转四个方面的表现。此分析根植于马歇尔 · 麦克卢汉的媒介定律——他认为每项新技术在增强一种功能的同时，会使另一种功能过时，找回某个过去的元素，并在推向极端时转变为自身的反面。这是平衡潜在收益与可预测故障模式的简洁之法。

4. **本集群的交互模式：**将优势转化为工程优势的可重复交互习惯。

然后我们介绍具体的交互模式。排序很重要：模式是“怎么做”，但优势与经济性解释了“为什么”值得这么做。

2.3.1 本章使用的成本模型

智能体软件工程通过改变成本来改写工作经济学。我们使用四种成本类型，并将它们锚定在引入 AI 队友之前的基线，让经济学讨论更具体：

- **时间成本：**周期时长、延迟和后期返工。也包括协调工作拖累进度所产生的时间成本。
- **社会成本：**“礼貌税”、对冲突的恐惧、怕显得优柔寡断，以及要求人类进行重复审查或提出异议时产生的摩擦。
- **资源成本：**稀缺的人力时间和专家可用性。这是旧世界以人员时间和日历协调为代价的领域。
- **注意力成本：**对你和团队的开销，包括审查负载、上下文重建、协调开销，以及合并与集成监督。

并非每个交互模式都会显著影响所有成本类型。每个模式仅列出其显著影响的成本。

2.3.2 本章使用的控制点

当输出增长速度超过监督和收敛速度时，智能体工作会适得其反。唯有当模式受到简单、可重复的控制点约束时，它们才是安全的。我们在每个模式中（如适用）使用相同的四个控制点：

- **契约：**验收标准、约束条件、非目标，以及“完成”的定义。
- **边界：**范围边界、停止规则、时间盒、迭代上限和允许的接触面。

- **委托与证据：**方法论上的自主权，加上必需的证据包和自检要求。
- **收敛与记录：**如何选择、合并、丢弃，然后更新活的真相源，以防止一致性发生漂移。

2.3.3 每个交互模式的结构

每个模式都遵循统一的结构。目标不是设计巧妙的提示词，而是在整个软件工程生命周期（从需求规划到质量保证、调试、审查，再到让随机性工作可信的工件）中，实现可靠的人机协作。

1. **模式标题：**团队在对话中可以随时引用的简短名称。
2. **是什么：**清晰概述该习惯、它产生的结果，以及 AI 队友使其焕发新价值的特别之处。
3. **解锁了什么：**一份积极的成本对比表，比较“仅人类队友”基线与“AI 队友+此模式”的情况。
4. **如何识别（及替代了什么）：**模式在何种情境下适用，以及它旨在替代的故障模式。
5. **风险与应对：**一个风险表，指出误用可能导致的反转效果及相关控制点，随后是基于契约、边界、委托与证据、收敛与记录构建的实用**操作指南**。
6. **示例交互流程：**一个顺畅的端到端微脚本，以项目符号序列呈现。
7. **工作原理：**更深层的原理，植根于软件工程、项目管理、管理科学和教育学。它提供了高于模式本身的心智模型：社会技术系统、不确定性、反馈循环和大规模决策。点名关键思想家和框架以供深入阅读，同时保持可读性和可操作性。忙碌的读者可以速览，好奇的读者可以深究。

2.3.4 交互模式如何关联智能体软件工程工件

第 1 章从概念层面剖析了人类与 AI 队友的协作方式，从非正式到结构化。它还点明了几个关键的结构化形式：结构

化意图、结构化指导、结构化执行与编排、结构化证据、结构化升级以及结构化治理。

一项实践通常始于非正式的轻量级提示习惯，随后固化为可复用的判断范式，纳入指导包，最终在风险要求时，通过工作流程运行手册以确定性触发器强制执行。触发的执行可以保持非确定性（使用提示），也可以根据风险级别，采用基于代码的自动化实现完全确定性。

这避免了一个常见误区：总想将每个交互模式与每个工件一一对应。目标不是制造文书工作，而是丰富工程师的工具箱，如何运用还需判断。

严格程度需要你来校准，就像你在处理涉及人类的工作时所做的那样。工程始终是在风险、成本和不确定性中权衡判断。智能体软件工程并未消除判断，反而因为你吞吐量提升、出错代价可能飞速攀升，而增加了需要你行使的判断量。

实践中几个关键的校准旋钮：

- **工作的风险画像**：涉及领域后果、变更影响范围、爆炸半径和可逆性。给 UI 文本微调和计费系统迁移设置相同的关卡，显然不合理。
- **自主权与信任的平衡**：AI 队友能在多大程度上无需人类插手即可执行，又需要通过检查、升级和审查来赢得多少信任。信任需要时间积累，就像与人类队友共事一样，但不能沦为盲目信任。
- **所用 AI 队友的能力**：是强大的通才模型，还是需要更严密脚手架、能力较弱或更廉价的模型。能力决定了需要多少前置条件设定，以及你该多频繁地将人类拉入流程关卡。

调高旋钮，你就强制设置更多关卡，要求更强有力的证据。调低旋钮，交互就保持轻量级。重点不是搞官僚主义，而是有意识地构建信任，而非指望热情能自动产生信任。

既然框架已就位，现在我们可以专注于这些模式，以及如何从 AI 队友身上榨取最大价值。

2.4 集群 A：不知疲倦，不带评判

2.4.1 它是什么

AI 队友不会疲劳。它们没有挫败感，不设防，也不会社交倦怠。它们不抱怨、不吝啬努力，更不会因为时间已晚或请求烦人就暗中降低质量。如果你要求再来一轮、换一个方案、再做一次清理检查或多解释一遍，它们会以和第一轮同样的劲头完成。

它们也不会在子任务上吝啬努力。如果完成任务的最佳方式是编写自定义脚本、构建一次性验证器或搭个临时测试框架，它们会毫不犹豫地动手。它们不会去权衡辅助工作是否“配得上”任务本身。人类或许会跳过那一步，选择肉眼检查。AI 队友则会为达成目标，构建任何需要的东西。

它们也不会评判你。你可以粗糙、不确定，甚至自相矛盾地喃喃自语，队友仍会帮你理清思路。这卸下了日常工程中沉重的社交负担。与人类协作时，我们不断为礼貌、时机、不显得优柔寡断而调整。与 AI 队友一起，这个约束很大程度上消失了，从而改变了你在需求、规划、分解、QA、审查、调试、指导乃至远超代码的工件工作中，能够常规要求的内容。

2.4.2 它解锁了什么

它让高质量迭代变得不再昂贵。过去，多一轮迭代需要付出三重代价：时间、资源和社交成本。这些成本迫使团队走向“差不多就行”的结果，并非团队甘于平庸，而是追求卓越在经济上难以为继。

它带回了我们明知有效却在压力下常常舍弃的实践：深度审查、系统性枚举边界情况、更严格的验收标准、更清晰的文档、更细致的需求打磨、为降低未来变更成本而进行的重构，以及有纪律的验证。

它也改变了 AI 队友处理工作的方式。由于“努力”不再是稀缺资源，AI 队友会自发地将构建一次性工具作为工作的自然组成部分。如果你让它将数据从一种格式迁移到另一种，它可能会先写个逐字段比对脚本来验证映射，再写个模式验证器检查输出，然后才报告结果。如果你让它重构一个模块，它可能先搭一个表征测试框架来锁定当前行为。如果你让它审计配置变更，它可能构建一个自定义检查器，标出新旧状态的每一处偏差。

这些都不是你要求的。它之所以发生，是因为 AI 队友不区分“真正的工作”和“杂务开销”。它不会像人类那样本能地做“相称性”计算：为这个任务造个工具值不值，还是我手动搞定算了？在人类的经济学里，这种计算是理性的。为一次性任务构建自定义检查器通常不划算。但在 AI 队友的经济学里，这种计算消失了，任务完成时往往附带一套前所未有的辅助基础设施。

打个工艺上的比方：这是夹具。熟练的工匠会为单项工作制作夹具和固定装置：一个把工件固定在正确角度的模块，一个确保切割笔直的导板。制作夹具本身不是工作，但它让工作变得精准。团队过去在时间充裕时这么做，一旦截止日期逼近，就觉得这是开销而停止了。AI 队友把这种习惯带了回来。

对人类来说，有两点很重要。首先，不要通过过度规定方法来扼杀这种行为。如果你一步步告诉 AI 队友怎么干活，就掐灭了它构建工具的本能。如果你只给它目标和验收标准，它通常会选择一种包含构建工具的方法来达成目标。其次，不要盲目信任工具。一次性验证器的好坏取决于其内在逻辑。如果它说 500 行数据全都有效，你最好手动抽查几行。风险不在于 AI 队友构建得“太多”，而在于你把工具的存在当成了验证已完成的证据，而工具本身可能有缺陷。

它也改变了你该优化的方向。风险不在于工作完不成，而在于你陷入了一个感觉高效却永不收敛的无限微循环。不知疲倦的迭代容易让人上瘾，也诱使人去微观管理 AI 队友。可扩展的做法是：明确验收标准，限定循环次数，让 AI 队友在如何达成目标上保持自主。

2.4.3 四元律速览

- **放大：**迭代、打磨、重新审视决策的意愿，以及自发创建辅助工具和脚手架。
- **取代：**因人力疲惫而妥协的“差不多就行”，以及害怕显得优柔寡断。
- **重现：**团队在时间和精力充裕时曾有过的工艺习惯，包括那些因感觉“杀鸡用牛刀”而被跳过的专用验证工具和自定义框架。
- **反转：**当循环不受限制时，导致的反复改动、微观管理以及人类注意力的耗尽。

2.4.4 此集群中的交互模式

- 无限迭代，有限循环
- 超越完成

2.4.4.1 无限迭代，有限循环

2.4.4.1.1 它是什么

此模式利用 AI 队友无限的迭代能力，快速收敛到一个站得住脚的工件，然后依靠证据（而非疲劳）来结束循环。重点不在更多修订，而在于将不确定性转化为一系列有限的快速迭代，每一步都有明确的验收标准作为终点。

2.4.4.1.2 它解锁了什么

成本类型	与人类队友合作时	与 AI 队友 + 此模式合作时
时间成本	额外迭代消耗日历时间，打乱其他工作，团队往往过早停止，带着不确定性进入下一阶段。	**你可以及早消除最高风险的不确定性**。迭代变得足够廉价，足以在假设变成后期返工前将其验证清楚。
社交成本	重复请求会产生摩擦，显得优柔寡断，消耗人情，因此人们避免深度迭代，即便它能提升质量。	**你可以要求再来一轮，而无需支付“礼貌税”**。深度迭代成为常规操作，而非特殊待遇。

2.4.4.1.3 如何识别它（以及它替代了什么）

当工件看似合理但尚不可信、不确定性集中在一两个高风险假设上、或团队因人力疲惫或时间压力被迫接受“差不多就行”时，使用此模式。

当你感觉到尖锐的边缘时使用此模式：隐藏的假设、遗漏的边界情况、模糊的边界行为，或者一个听起来连贯但尚未赢得信心的计划。

当你发现迭代是因“人累了”而停止，而非证据已清晰时，也适用此模式。

该模式取代了过早收敛、虚假的确定性，以及表现为反复改动、升级和意外审查周期的后期清理。

2.4.4.1.4 风险及应对方法

风险类型	如果误用，会变成什么	控制点
注意力成本	人类成为瓶颈，被输出淹没。审查沦为橡皮图章或变成没完没了的指导。	限定，基于证据委托
时间成本	反复改动取代了实质进展。迭代只在措辞或结构上打转，未触及核心不确定性。	约定，限定，收敛并记录
社交成本	挫败感转移到人类身上。循环感觉像是微观管理和事后指责。	约定，基于证据委托

约定
定义下一轮迭代的验收标准，而非整个项目的。

- 选定本轮迭代要解决的一个不确定性。
- 编写二元化的验收标准，让审查者能直接说“是”或“否”。
- 说明约束和非目标，防止迭代范围蔓延。

限定
在开始前就框定循环边界。

- 设置迭代次数上限和时间盒。
- 添加收益递减触发器，当证据不再变化时自动停止。
- 保持范围足够小，以便人类能完整审查。

基于证据委托
在方法上给予自主权，但要求每轮提供证据包。

- 使用统一的包格式：改了啥、验证了啥、结果如何、还剩哪些假设、对照标准的自检。
- 拒绝任何无法说明验证过程的迭代。
- 将反馈视为对标准的更新，而非对具体方法的指示。

收敛并记录
按规则停止，而非筋疲力尽才停。

- 仅当证据包满足约定条件时才接受。
- 记录现已确定的内容和尚待解决的问题，记在团队真会看的地方。
- 如果有事项被推迟，请明确命名、划定范围，并记下下次重新审视的触发条件。

2.4.4.1.5 示例交互流程

- 人类：“本轮迭代的验收标准是 A、B、C。验证方法是 V。在动手改任何东西前，请先复述标准和验证方法。”

- AI 队友："复述标准 A、B、C。验证方法 V。提议的迭代仅针对不确定性 U。停止规则是 S。"
- 人类："批准。执行这个限定迭代。"
- AI 队友："完成。证据包内容：更改项、验证步骤与结果、剩余假设、对照 A/B/C 的自检。"
- 人类："决定：接受或拒绝。若拒绝，反馈是对标准的更新，而非方法指示。"
- AI 队友："更新约定，提出下一个限定迭代方案，然后执行并提供证据包。"
- 人类："按停止规则停止。更新记录：现在确定了什么，还有什么未关闭。"

2.4.4.1.6 起作用的原理

闭环迭代必须包含验证，因此"证据包"必不可少。戴明的改进纪律早已警示：除非循环中包含能证伪你的检查，否则活动不等于进展。对 AI 队友而言，生成看似完成却未消除不确定性的输出太容易了，因此此模式强制要求每轮迭代都以验证和一个小证据包收尾，确保学习过程明确且高度可逆。

预先定义"好"的验收标准，让循环保持清晰可理解。布鲁克斯关于概念完整性的观点在此同样适用：团队事先就"好"的含义达成一致，远比事后争论细节更高效。此模式将验收标准变为工作契约：人类掌握意图、约束和构成充分证据的标准；AI 队友掌握方法。这避免了通过细微编辑来指导的窘境。

廉价的迭代需要决策卫生，以防陷入无尽搜索和虚假自信。西蒙的有限理性解释了团队为何在压力下满足于"差不多就行"。廉价迭代本可通过降低探索成本来帮忙，但卡尼曼和特沃斯基提醒我们：更强的迭代能力也会放大对早期草案的锚定、对连贯故事的过度自信，以及因感觉"便宜"而产生的无尽搜索。停止规则和迭代上限迫使团队定义"什么证据足以决策"，并在满足时果断停止。

学习必须纠正前提，而非仅仅打磨输出。阿吉里斯和舍恩的双环学习至关重要，因为许多失败源于前提错误，而非执行错误。此模式通过要求明确假设、说明差异，并将拒绝反馈用于更新验收标准（而非规定具体方法），来保持循环的诚实性，使团队能更早地修正“自以为是”的东西。

当输出超出监督能力时，注意力和流程成为进展的瓶颈。流程思维和利特尔定律解释了为何无限制的在制品会增加周期时间和不可预测性。无限制的迭代在人与 AI 协作中对审查和决策同样有这种影响。此模式保持批次小、证据密集，从而使检查点有意义，团队得以收敛，而不是淹没在“接近成功”的海洋里。

2.4.4.2 超越完成

2.4.4.2.1 它是什么

所谓“超越完成”，就是别逮着第一个能过测试的改动就收手。你要善用 AI 队友的耐心和重写本领，把收尾工作做扎实：结构更清晰、命名更得当、重复代码减少、测试更牢固，再添上些让后来人一目了然的小文档或使用说明。这不仅适用于代码，任何工件皆然：需求简报可以更精炼，计划可以更简洁，任务分解可以更稳妥，QA 检查清单能把边边角角都考虑到，评审材料也能做得干净利落。

这个模式奉行童子军信条：凡你经手之处，必须比来时更整洁。AI 队友特别擅长这类“表面功夫”的整体提升。但风险在于，清理可能演变成范围蔓延或偷偷改了语义。只要把功能意图和清理工作分开、给清理划好边界、并且每次优化都要拿证据说话，这个模式就能用得稳妥。

2.4.4.2.2 它解锁了什么

成本类型	与人类队友合作时	与 AI 队友合作 + 此模式时
时间成本	截止日期压顶，收尾工作常常一拖再拖。清理任务扔进待办列表，便再难见天日。	**你能在交付前顺手提升质量，还不敢误事**。最后那 10%的打磨变得轻而易举，可以当即完成，既减少了技术债务，也压低了未来的修改成本。
社交成本	让一个筋疲力尽的队友再去抠命名、补测试或写文档，既不公平，也容易被当成完美主义找茬。	**你能理直气壮地要求收尾，还不伤和气**。质量改进不再像是强加给疲惫队友的额外负担。
注意力成本	评审者苦不堪言：意图淹没在杂乱的修改里，债务不断堆积，导致后续每个拉取请求都像在读天书。	**即便输出规模膨胀，你也能保持变更清晰可读**。把结构调整、测试补充和文档完善作为完工的一部分，评审就能专注判断，而不是搞考古发掘。

2.4.4.2.3 如何识别它（以及它替代了什么）

当你发现某个改动虽然技术上可行，却把周围代码搞得一团糟时，就该启用此模式：命名混乱、API 别扭、逻辑重复、错误处理脆弱、边缘情况遗漏、缺少使用说明，或者修改意图模糊导致评审困难。

当你嗅到未来成本的味道时，也该用它。下一个工程师要么绕道走，要么一碰就炸。

当团队习惯了“先交差，后清理”，而技术债务堆积如山却无人偿还时，此模式正是解药。

它用“有计划的负债”取代了“意外负债”。它用“有界的收尾”替代了“到点打卡”思维，从而在吞吐量飙升时，依然能保持系统的可读性。

2.4.4.2.4 风险及应对方法

成本类型	若误用，会翻转成什么	控制点
时间成本	清理演变成重新设计。拉取请求永远合并不了，没完没了。	约定、限定、收敛并记录
注意力成本	行为变更和清理变更纠缠不清，评审根本没法进行。	约定、限定、基于证据委托
资源成本	测试薄弱或行为定义不清时，大规模重写会引发昂贵的调试和救火。	限定、基于证据委托

约定
把功能意图和清理意图分清楚。

- 第一轮：只做行为变更，并满足验收标准。
- 第二轮：只做不改变语义的清理。
- 第三轮（如有必要）：因接口变动而必须更新的文档或示例。

约定要写明白，让评审者能轻松回答两个问题：行为变了什么？结构变了什么但行为没变？

限定
给清理工作划好地盘。

- 范围限在你已修改的文件内，或一个明确指名的区域。
- 限制本轮允许的重构类型：重命名、提取函数、简化控制流、消除重复。
- 白纸黑字禁止在清理期间改变行为。

如果 AI 队友发现了更深层的重设计机会，记下来以后再说，别在本轮折腾。

基于证据委托
优化不能空口无凭，得拿出证据。

- 运行和行为变更一样的验证步骤。
- 面对行为模糊的遗留代码，先加针对性测试或特征测试探路。
- 覆盖率太低时，优先进行小范围的清理操作。

清理工作的证据包必须说清三件事：结构上改了哪里、为什么行为应该不变、以及跑了哪些验证。

收敛并记录
达到目标就收手。

- 证据确凿后，先合并行为变更那轮。
- 清理轮次只要不越界且验证通过，就合并它。
- 把任何推迟的深度重构记下来，作为有明确范围的后续任务。

2.4.4.2.5 示例交互流程

- 人类：“先交付功能变更。验收标准是 A、B、C。验证用 V。暂时别清理。”
- AI 队友：“复述标准和验证。仅实施行为变更。”

- AI 队友："提交行为变更证据包：改了哪里、验证步骤和结果、剩余风险。"
- 人类："通过。现在提个有界的清理计划：改进 3 到 5 处、涉及哪些文件、验证怎么保持。"
- AI 队友："提交范围内清理计划。无行为变更。验证仍为 V。"
- 人类："批准。执行清理，并返回专注语义保留变更的证据包。"
- 人类："停。记录我们改了什么，以及明确推迟了什么。"

2.4.4.2.6 起作用的原理

技术债务是笔经济账，无关道德瑕疵。沃德 · 坎宁安（Ward Cunningham，技术债务概念提出者）的技术债务比喻点明了为何小捷径当下觉得划算，日后却会积重难返，导致变更缓慢、发布风险高、系统脆弱。"超越完成"利用 AI 队友，让最后 10%的打磨成本低廉到团队可以顺手为之，既保护了未来的吞吐量，又不会陷入完美主义的泥潭。

安全清理靠的是行为不变和快速反馈。马丁 · 福勒（Martin Fowler，《重构》作者）的重构纪律就是安全准则：在不改变行为的前提下改善结构，让测试和持续集成告诉你是否越界。因此，本模式将落地功能和保留语义的清理分开，并通过减少评审时的模糊地带，确保回滚路径清晰。

事后发现问题代价高昂，所以要在变更尚小、可逆时提前提升质量。玻姆（Barry Boehm，软件工程经济学先驱）的变更成本模型是"超越完成"背后的项目管理支柱："以后再说"往往意味着"以后多花钱"，因为到时耦合更紧、上下文流失、操作风险更高。提前打磨质量，通过尽早降低不确定性来减少返工，缩短未来的周期。

清晰可读性是高吞吐量下的集成能力。帕纳斯（David Parnas，信息隐藏原则提出者）认为好的模块化能让变更局部化、可预测，而清晰可读性正是这一思想的日常实践。当代

码及相关工件易于阅读时，评审加速、集成更安全、值班诊断也更从容——这在智能体软件工程产出激增、变更速率加快时尤为重要。

注意力分散和截止日期压力让团队只想关闭工单，因此有界的优化必须成为习惯。迪马科和李斯特（Tom DeMarco & Timothy Lister，《人件》作者）描述了中断、混乱和工期压力如何挤掉那些无法推进当日演示的清理工作。AI 队友减少了要求优化的社交摩擦和资源消耗，而本模式将这种自由转化为有界的习惯，让系统越来越整洁，又不会把每次变更都变成无尽的重写。

证据不足时，遗留系统在美化前得先“画像”。迈克尔 · 费瑟斯（Michael Feathers，《修改代码的艺术》作者）的特征测试思想是薄弱测试代码的护身符：先抓住系统当前行为，再在这个保护伞下重构。“超越完成”通过优先进行小范围局部清理、先补测试、以及避免大规模重写（除非验证足够强大）来保持安全。

2.5 集群 B：强沟通者

2.5.1 它是什么

AI 队友特别擅长把人类那些潦草的想法变成结构化的意图。你大可以扔过去粗略的念头、错别字、半截话、语音转文字稿或白板截图，他们照样能提炼出目标、假设、约束和开放问题。然后，他们能以团队便于执行和验证的形式，把工作重述清楚，省去你事前费力“美化”的额外功夫。

他们还擅长用最易达成共识的形式传达同一件事。他们能在要点、表格、检查清单、伪代码、图表和简短叙述之间自由切换。更重要的是，他们擅长综合。他们能串联起跨线程、跨工件的线索，发现人们忽略的关联，并持续维护一份团队公认事实的动态图谱。

不过要当心，强沟通也可能无声无息地失败。潦草的输入可能藏着未言明的约束，综合时可能意外丢失细微差别、过度简化权衡，或者把假设说得像既定决策。因此，沟通输出必须赢得认可。队友负责搭架子，人类负责保正确。

2.5.2 它解锁了什么

它让高质量意图捕捉变得廉价。动嘴总比动笔快，画图通常更快。过去，把粗略想法转化成清晰的需求、验收标准和验证步骤，既耗费稀缺的高级人力，又难免社交摩擦。AI 队友让你可以尽早抛出意图，然后快速打磨成形。

它让达成共识比开会更划算。共享的心智模型、清晰的接口和具体的例子，都能减少返工。问题一直在于：写下模型、保持更新并向每个新人重复解释的成本太高。

它让并行工作下的连续性变得廉价。随着更多工作流齐头并进，重新协调和重建上下文的成本猛增。AI 队友能动态维护关于目标、决策、开放问题以及跨关键工件证据的摘要。

所有这些的控制点都一样：坚持明确的假设、明确的未知项、执行前获得明确认可，并且把综合与证据绑定，确保其可审计。

2.5.3 四元律速览

- 放大：清晰度、达成共识的速度、表现形式的灵活切换
- 取代：基于会议的翻译工作，以及“先抛光再沟通”的老习惯
- 重现：将意图捕捉为契约，而非感觉
- 逆转：当模糊性和证据未被呈现时，自信满满的误解和摘要漂移

2.5.4 此集群中的交互模式

- 草率输入，清晰输出
- 展示，而非告知
- 可缩放综合

2.5.4.1 草率输入，清晰输出

2.5.4.1.1 是什么

这种模式让你从真实的人类输入着手——这些输入往往一团乱麻：口头念叨的想法、错字连篇的笔记、聊天片段、日志、截图或白板照片。它不会逼你把所有东西重写成精炼文稿，而是让一位 AI 队友从中提炼意图，生成一份结构清晰、团队共享的契约。这份契约审查起来轻松，还不容易会错意，它囊括了目标、非目标、约束、假设、验收标准、风险、开放问题和验证方法。

它的好处可不只是快。它还省去了团队因隐藏不确定性而付出的社交代价。但风险在于，清晰可能只是假象，正确性却不见踪影——AI 队友可能会用看似合理的猜测来填补空

白。只有当第一份输出被当作契约草案、假设与来源分开陈述，并且执行过程以验收为关卡时，这种模式才算安全。

2.5.4.1.2 它解锁了什么

成本类型	与人类队友合作时	与 AI 队友合作 + 此模式时
时间成本	工作干等着，因为得有人把混乱的讨论翻译成可审查的请求，而这翻译活儿还总被拖延。	**你能快速把混乱输入变成可执行的契约**。团队不再为精炼过程耽误时间，只为关键决策买单。
社交成本	大家往往过度精炼、拖延或干脆不承认不确定性，因为草率的输入显得准备不周。	**你可以早早把意图摆上台面，不用担心显得优柔寡断**。不确定性变成了明确的假设和开放问题，不再是藏着掖着的“氛围”。
注意力成本	审查者得从讨论串和差异里重建意图，在判断变更前先搞一番考古挖掘。	**你能给每个人一个单一的对象来回应**。审查变成了针对对象本身，而不是费力重建对象。

2.5.4.1.3 如何识别它（以及它替代了什么）

当意图因为记录成本太高而困在某人脑子里，讨论串又因不同解释来回打转时，就用这模式。

当需求还在成形，发现过程靠正例反例推进，但大家把修订看成返工和折腾时，就用这模式。

当执行从定义不清的请求开始，而你已能预见后续返工时，就用这模式。

它用一份已接受的契约取代了精炼延迟、模糊执行和讨论串考古，这份契约锚定了执行和验证。

2.5.4.1.4 风险及应对方法

成本类型	若误用，会翻转成什么	控制点
时间成本	你执行了一份看起来清晰实则错误的契约，随后在返工中付出代价。	契约，基于证据委托
注意力成本	契约变成一个自信却与来源现实脱节的故事，审查者不再信任它。	契约，收敛并记录
社交成本	大家把契约当法令而非草案，真实异议被压制。	契约，收敛并记录

契约
核心规则：输入可以乱，契约必须清。

- 要求 AI 队友把“原话”和“推测”分开标。
- 要求明确的假设和明说的未知项。
- 要求验收标准和验证方法必须可测试，不能是美好愿望。

以验收为执行开关。契约没被接受前，不动工具、不编辑、不分派任务。

边界
别在契约细化上兜圈子。

- 给捕获和细化阶段设定时间盒。
- 限制提问数量，只问那些能改变范围、风险或验证的问题。
- 如果契约老是定不下来，就换模式：加点例子、建个决策表，或者跑个小探索任务来摸清情况。

基于证据委托
把已接受的契约当作真理之源，明确需要提供什么证据。

- 尽量把验收标准转化成检查项。
- 要求为契约本身提供证据：示例、决策表、来源链接或工件指针。
- 要求 AI 队友展示自查结果：哪些标准已满足以及怎么满足的。

收敛并记录
让契约保持鲜活和最新。

- 决策或证据变了，就更新它。
- 把它放在干活的地方——PR、issue 或审查者真会看的链接文档里。
- 未决事项记成开放问题，带上负责人和下一步，别让它变成隐含决策。

2.5.4.1.5 示例交互流程

- 人类：“这是一堆乱糟糟的笔记和日志。把它们变成结构化契约：目标、非目标、约束、假设、验收标准、风险、开放问题、验证。把你推断的内容高亮出来。先别执行。”
- AI 队友：“契约草案已生成。推断项已标记。现提出最多五个可能改变范围、风险或验证的问题。”
- 人类：“答案在此。重新发布契约。我会决定接受或拒绝。”
- AI 队友：“修订后的契约、验证计划，以及从标准到检查项的映射关系。”
- 人类：“接受了。尽量把标准转成检查项。提议一个最小的第一步，并说明你会返回什么证据。”
- AI 队友：“第一步已执行。证据包已返回。如有变更，契约已同步更新。”

2.5.4.1.6 起作用的原理

共享心智模型和概念完整性能在执行放大偏差前防止解释漂移。布鲁克斯的概念完整性观点在微观层面照样管用：多数故障并非“文笔糟糕”，而是大家心里想的不是一回事。通过把意图结晶成一份单一、可审查的契约，团队就能在并行执行放大分歧、审查沦为解释之争（这可是经典的社会技术系统故障模式）之前，先围绕一个共享对象达成一致。

需求是靠例子摸出来的，不是从第一条消息照抄的，所以用 3Cs 框架来驱动收敛。3Cs 框架（卡片、对话、确认）抓住了需求发现的常态：从一张轻量级的卡片开始，通过例子、反例和边界案例反复对话，最后以确保意图可测试的确认收尾。这种模式把草率输入视为正常的早期探索阶段，并用结构化契约强制形成一个清晰的收敛点，而不是一开始就要求交出完美文稿。

降低交易成本和社交摩擦让不确定性可以摊开说，从而提升了早期决策质量。科斯（Ronald Coase，交易成本经济学奠基人）的交易成本视角解释了为何团队常带着一个看似合理的故事蒙眼狂奔，而非明确表达不确定性：跟人类队友澄清既费时间又耗人情。AI 队友消除了这种社交摩擦，而本模式通过明确呈现假设和未知项，把摩擦转化成了更早的清晰度，无需苦等会议或完美文稿。

有限理性驱使团队寻求满意解，因此本模式通过明确验收把决策边界前移。西蒙（Herbert Simon，有限理性理论提出者、诺贝尔经济学奖得主）的有限理性解释了为何定义不清的请求会变成自信满满的执行：团队接受第一个能让活干起来的连贯解释，然后在后续意外中付出代价。通过强制公开假设、未知项和验收标准，并要求在任务铺开前明确验收，本模式改善了大局决策，减少了因过早收敛导致的返工。

可证伪性让循环基于证据而非故事惯性，所以契约必须可检查。戴明（W. Edwards Deming，质量管理与 PDCA 循环

推广者）的改进逻辑在这里体现为一种纪律：契约只有可检查才有用。本模式坚持验收标准和验证方法，让团队能快速判断工作是满足了真实需求，还是只产出些听起来靠谱的东西——这在不确定性下强化了反馈循环。

外部认知和教学支架让一致性跨越时间、角色和中断得以保持。迪马科和李斯特对知识工作、中断和”瞎忙活”的观察解释了为何意图常锁在个人脑中，上下文散落于讨论串，迫使审查、入职和事件跟进时反复考古。结构化契约充当了外部认知，从教学角度看，它是个支架：用结构化形式反映粗略思考，并通过针对性问题弥合关键差距，将聚焦点限制在范围、风险和验证上，从而让认知负荷可控。

2.5.4.2 多画图，少废话

2.5.4.2.1 是什么

这种模式的精髓在于：针对不同的问题，选择合适的表达形式。纯文字擅长讲道理、说故事，但软件工程里的很多麻烦，根本就不是句子能说清的。它们关乎流程、规则、边界和状态——当你硬把这些东西塞进大段文字里，每个读者都得在脑子里重新勾勒一遍，结果十个人能画出十种模样。这里的做法，就是让你的 AI 队友把你那些话，“翻译”成结构化的表达：时序图、决策表、状态机、数据流草图、边界图，或者那种紧凑、方便对比的 ASCII 草图。

咱要的不是花架子。目标是让结构本身变得清晰可见，这样分歧就能早点暴露。风险在于“看上去很懂”——一张清晰的图表也可能是错的、不完整的，或者藏着一堆想当然的假设。只有当图表本身足够轻量、假设被明确标注、并且图表与具体检查挂钩时，这种模式才算稳妥。

2.5.4.2.2 它能解锁什么

成本类型	与人类队友合作时	与 AI 队友合作 + 此模式时
时间成本	画图表做表格？感觉太麻烦，常常能省则省。结果分歧拖到实现或集成阶段才冒头，为时已晚。	**你能尽早揪出结构上的分歧**。图表变得唾手可得，成了前期统一思想的利器，而不是事后补文档的累赘。
注意力成本	对着大段文字，人人都在心里建自己的模型，导致对齐来回拉锯，评审时还得争论“作者本意到底是什么”。	**你能省去那些暗地里的解读功夫**。一个共享的图表成了大家的靶子，减少了返工和评审时的扯皮。
社交成本	挑战一段叙述，感觉像在挑战作者本人，所以模糊地带往往一直模糊，直到问题炸开。	**你能让分歧对准“图”，而不是“人”**。大家可以直接指着图问：“这个接口画对了吗?”

2.5.4.2.3 如何识别它（以及它替代了什么）

当讨论陷在文字游戏里，大家用不同说法重复同一个点子；或者一旦有人追问具体步骤、规则、边界或状态变化，共识立马崩塌时——就该用这招了。

当行为随时间展开或依赖条件时，用它：比如重试、超时、授权流程、数据迁移、功能开关、跨服务的工作流，还有事件响应路径。

当你想要一个稳定的、能在不同人和工具间传递，但又不会变成一堆繁文缛节的东西时，也用它。

它用看得见、摸得着、能评审、能版本化的具体图表，替代了那些转瞬即逝的口头对齐、心照不宣的结构，以及后期才发现的对不上号。

2.5.4.2.4 风险及应对方法

成本类型	如果误用，会翻转什么	控制点
注意力成本	团队没完没了地争论图表细节，或者维护着一幅幅没人更新的“壁画”。	限定范围、尽快收敛、做好记录
时间成本	花了大力气画图，却没用来驱动验证，结果问题还是在最后关头才暴露。	明确约定、授权执行、附上证据
社交成本	图表成了表演道具甚至攻击武器，引发咬文嚼字的争论，反而掩盖了真正的不确定性。	明确约定、限定范围

约定
先说清楚，画这图是干嘛用的。

- 点名不确定性：时序？规则？状态？边界？归属？还是故障路径？
- 明确接受这张图时，必须满足哪些条件。
- 要求把假设和未知项标得一清二楚。

限定
在方案稳定前，图要画得小、画得随意，随时能扔。

- 默认一屏就能看完。
- 尽可能把“现在怎么做”和“打算怎么做”并排摆出来。
- 别过度建模。一旦成了“壁画”，它就离过时不远了。

委托并附证据
让图表去驱动检查。

- 把决策表变成测试用例。
- 把状态机变成非法状态转换的断言。
- 把边界图变成契约测试和所有权检查。
- 要求 AI 队友提出覆盖这张图的最小验证方案。

收敛并记录
让图表紧贴它要管理的那个具体东西。

- 把它放在相关的 PR、issue 或者链接文档里。
- 设计一有变动就更新它，别等别人来问。
- 记录图表改了哪里，以及是哪些检查确保了这些改动。

2.5.4.2.5 示例交互流程

- 人类：“根据这些笔记，把当前和提议的行为画成一屏的时序图。标出假设和未知项。先别执行。”
- AI 队友：“图画好了，两份。差异已高亮。假设和未知项已列出。”
- 人类：“行。现在把规则转成决策表，并给出覆盖它的最少测试。”
- AI 队友：“决策表在此，附上测试计划和具体的验证命令。”
- 人类：“通过。把有风险的故障路径转化成检查点。然后继续实施。”
- AI 队友：“实施完毕。证据包里包含了与图表对应的检查结果。”

2.5.4.2.6 为什么这招管用

在社会技术工作中，把想法“画出来”让大家看同一个东西，远比让每个人在脑子里“重建”要省力。问题不在于大家看不懂文字，而在于文字迫使每个读者私下重构结构，结果各想各的。在需要协作的系统中，这种分歧就成了协调成本和决策摩擦。可视化或结构化的图表，把共享模型从个人的工作记忆里搬出来，变成了团队可以共同指着、修改、复用的工件。这既降低了协调和认知的负担，也为新手提供了学习支架，同时让专家在复杂流程上保持同步。

多画几张不同的图是控制风险，不是增加开销，因为不同的“不确定”需要不同的“解法”。软件工程早有先例：UML 就用不同的图（时序、状态、边界、数据流）来应对不同的不确定性；架构层面的 C_4 模型，则用不同的缩放级别来满足不同受众验证同一系统的需求。如今，AI 队友让绘制这些视图变得廉价、快捷，在设计稳定前可以随意迭代和丢弃，从而在不必要开会或进行冗长辩论的情况下，提高了决策质量。

让不确定性尽早现形，避免等到集成或上线后才付出惨痛代价。当真正的风险在于时序、条件或所有权边界时，文字常常掩盖分歧，直到系统拼装起来或已经发布。这种模式通过催生一个大家可以共同审视和修正的具体工件，迫使不确定性浮出水面。这大大压缩了“我们嘴上说同意，但心里想的不一样”的空间，并在还有回旋余地时暴露不匹配，从而提升了重大决策的质量。

基于模型的思维方式，让图表从沟通工具升级为保障正确的工具。在高可靠系统和安全关键领域，人们使用状态机、决策表和时序图，是因为它们能在你动手写代码前，就暴露出非法的状态转换、遗漏的分支和模糊的归属。把这些模型当作一等公民的工件，能让整类的故障在代码评审之前就变得可见、可查，从而提升了操作的正确性。

图表必须和证据绑定并保持更新，否则就会沦为误导人的过期文档。戴明环（Plan-Do-Check-Act）的纪律在此直接适用：一张图表赢得信任，不是因为它画得漂亮，而是因为它能驱动可被检验的检查和验证。决策表变成测试矩阵，状态机变成非法转换断言，边界草图变成契约测试。同时，保持工件轻量，这样当现实变化时它才能方便地更新。这也是防止文档漂移的秘诀：让工件足够小，利用 AI 队友降低更新和切换视图的阻力，并把每个视图与证据牢牢绑定，让它始终可被审计，而非流于形式。

2.5.4.3 可缩放综合

2.5.4.3.1 是什么

可缩放综合是一套纪律严明的习惯，旨在让工作演进过程中的对齐目标始终保持清晰可读。说白了，它的做法很简单：用多个缩放级别来维护同一个故事，让大家各取所需。一行字说清状态与下一步；一段话讲明当前叙事；一页纸记录完整工作历程。AI 队友负责实时更新这些层级，让团队对齐不再依赖开会、记性好或谁刚好在线。

它的杠杆效应在于，维持对齐的成本大幅降低。但风险在于，综合本身就是有损压缩——细节可能丢失，不确定性可能被抹平，分歧也可能被自信的总结所掩盖。这种模式安全的前提是，综合必须基于证据指针来构建，而不能把自己当成真理本身。

2.5.4.3.2 它解锁了什么

成本类型	与人类队友合作时	与 AI 队友合作 + 此模式时
注意力成本	大家得像考古一样，在各种线程、提交和文档里挖来挖去，才能回答“进展如何”和“当初为什么这么定”。	**不用开会也能保持对齐一目了然**。一行、一段、一页成了默认的导航界面。
时间成本	因为上下文总得反复重建，所以审查和新人上手都慢。	**审查周期和上手时间都能缩短**。综合文档成了入口点，需要深究时再往下钻。
社交成本	状态更新变得**表演性**且会议泛滥，因为信息到处散落。	**重新对齐的会议可以减少了**。综合文档承载当下实况，让交接更稳妥。

2.5.4.3.3 如何识别它（以及它替代了什么）

当工作横跨多次迭代、冗长线程或并行分支，并且你预料到“进展如何”或“为什么选这个”会被反复问起时，就用此模式。

当你需要跨角色或跨时区协调，且不同听众需要不同详略程度时，也用此模式。

当状态已经七零八落时更该用：当前状况、要求、进展和证明散落在聊天、提交、文档和工单里，找都找不全。

它用一个持续更新的可靠**入口点**，取代了上下文漂移和状态散落。

2.5.4.3.4 风险及应对方法

成本类型	如果误用，会翻转成什么	控制点
注意力成本	综合成了大家深信不疑的“故事”，哪怕现实早已改变，最终信任崩塌，考古式挖掘卷土重来。	收敛与记录，基于证据委托
时间成本	大家只看摘要的自信结论就合并，忽略了真实风险，导致后期炸雷。	契约，基于证据委托
社交成本	简洁的摘要压制了分歧和不确定性，因为它听起来已经一锤定音。	契约，收敛与记录

契约
明确综合必须包含什么，以及它不是什么。

- 必须包含：当前状态、任务要求、已完成事项、剩余工作，以及证据索引。
- 它不是：证据的替代品。它只是一个导航层。

边界
给它安个规范的家，并且保持**短小精悍**。

- 小活儿：放在 PR 描述或 Issue 正文里。
- 大活儿：用一个简短的链接文档，让 PR 和 Issue 都指向它。
- 一个家足矣，别弄得到处都是。

使用固定模板，让更新保持一贯风格。

基于证据委托
让综合建立在指针之上。

- 每个重要陈述都要指向证据：命令、日志、跟踪记录、截图、基准测试、工单。
- AI 队友更新综合时，只更新差异和指针，而不是编造一个更圆滑的故事。

收敛与记录
情况变了就更新综合，别等到有人来问。

- 记录决策及其理由。
- 记录待解决的问题，并指定负责人和下一步。
- 合并前，从综合里生成合并包：需求、变更点、主要风险、具体的验证步骤与结果、上线说明。

2.5.4.3.5 示例交互流程

- 人类：“在 PR 里建个可缩放综合：一行、一段、一页。包含当前状态、要求、进展、剩余工作以及证据索引。先别执行。”
- AI 队友：“综合已创建，附带证据指针。待解决问题已列出。”
- 人类：“继续。每个里程碑之后，更新综合和证据索引。”
- AI 队友：“已更新一行状态、段落叙述和一页记录。新增证据指针已添加。”
- 人类：“合并前，从综合里生成合并包，并列出具体的验证结果。”
- AI 队友：“合并包已生成，内含验证命令和结果。”

2.5.4.3.6 起作用的原理

随着吞吐量上升，协调成了主要瓶颈，而可缩放综合守护着概念完整性。《人月神话》的教训历历在目：沟通、概念

完整性和集成主导着实际进度。AI 队友把瓶颈更猛地推向了共享理解和决策一致性。可缩放综合让当前的对齐目标保持明确、实时更新，从而确保决策一致、集成顺畅，团队无需为后期的目标漂移买单。

多层级的外部认知扩展了团队能力，既减轻了工作记忆负担，也避免了考古式挖掘。当状态散落在各种线程、迭代和角色之间时，每个人都不得不考古，而且每个人重建的故事都不一样。可缩放综合把外部记忆明确定义为外部认知：一行用于快速同步，一段用于共享叙事，一页用于持久的工作记录。从教育角度看，这些层级就像脚手架：一行支持过程检查，一页支持总结验证与问责——这在**随机性贡献者**能产生大量似是而非的输出，却又必须脚踏实地、避免退化为氛围编程时，显得尤为重要。

可见性本身就是一种控制，因此要把综合视为信息辐射器，而非会议的替代品。敏捷和精益实践早已发现，当现状无需开会就能看清时，协调效率就会提升。科伯恩（Alistair Cockburn，敏捷宣言联合签署人）的”信息辐射器”解释了为何共享的、轻量的工件能降低重新对齐成本并防止漂移。德鲁克的管理框架也指向同样的控制直觉：看不见状态，就无从管理。这里的“可见”，指的是可读的意图、当下的决策和证明轨迹，而不是一串流水账式的活动记录。

摘要必有损耗，因此综合必须可审计，且由证据驱动更新，而非应要求更新。典型的失败模式是“自信的压缩”：不确定性被抹平，假设悄然变成决策，综合成了大家深信的故事，哪怕现实早已改变。因此，本模式将综合视为与工件绑定的导航：陈述指向证据，决策指向理由，验证指向具体的命令和输出，并附带差异与证据索引。戴明式的循环纪律在此直接适用：更新应由新证据和新决策触发，否则综合就会沦为走过场（状态剧场），而不是真正的控制界面，叙事也会与实际完成和验证的内容越走越远。

并行工作下的可预测交付，需要严格的在制品纪律，因此可缩放综合为协调在制品设定了上限并促进收敛。流程思维和利特尔定律告诉我们，无限制的在制品只会拉长周期、增加不确定性——在制品越多，等待、切换和返工就越多。可缩放综合为每个角色提供了快速了解工作真实状况的入口，这加速了审查，降低了新人上手成本，并帮助并行分支以更少的冲突收敛，因为共享状态易于查验。

2.6 集群 C：广阔的世界知识

2.6.1 是什么

AI 队友带来了异常广博的技术与领域知识，以及快速切换视角的能力。它的价值不在于死记硬背，而在于能用多种视角审视同一局面，串联起通常散落在不同人脑中的约束，并提出兼顾架构、测试、性能、安全、可靠性、运维和用户影响的可行方案。

这个集群也代表着一种姿态转变：你需要的是协作者，而非仆从。目标是获得更优的方案、更清晰的约束和更早暴露的风险。风险在于，广度听起来像深度——即使上下文缺失或真正的约束是你系统特有的，AI 队友也能生成一套看似合理的解释。控制之道在于，将 AI 队友视为方案生成器和风险扫描仪，然后通过明确的验收标准和证据来强制收敛。

2.6.2 它解锁了什么

它让探索变得廉价。你可以快速获取多种实现路径、权衡取舍和二阶效应，而无需为了一个初步方案就召集满屋专家。

它让结构化的异议在社交上变得廉价。分歧本有价值，但往往难以按计划获得。人们常因怕添麻烦、怕多干活而避免反对。有了 AI 队友，你就可以有目的地征求异议，并将其直接关联到具体工作上。

它让跨职能约束提前浮现。许多惨痛的失败并非因为代码写错，而是因为错过了运维约束、隐性的安全假设、性能悬崖、脆弱的上线路径或含糊的所有权界定。AI 队友降低了

早期发现这些约束、并将其转化为具体工件、检查项和验收标准的成本。

2.6.3 四元律速览

- 增强：方案生成、跨职能思维、早期风险发现
- 取代：将早期反馈硬性依赖于利益相关者的时间
- 重现：有纪律的场景推演、事前验尸和架构评估
- 逆转：分析瘫痪或自信的胡说八道（当未严格执行证据门槛时）

2.6.4 此集群中的交互模式

- 角色扮演
- 魔鬼代言人

2.6.4.1 角色扮演

2.6.4.1.1 是什么

角色扮演，就是让 AI 队友临时客串特定的利益相关者，比如负责评审、批准、操作或支持工作的人。这个角色必须产出一份验收包，讲清楚三件事：得满足什么条件、必须有哪些工件、以及拿出什么证据来证明。此法通吃各类工件，代码自不必说，需求记录、计划、任务分解、质量检查清单、缺陷摘要、评审包、运行手册乃至代码变更，统统适用。

妙处在于，不用折腾日程协调，关键角色就能早早入场。风险则是流于形式：角色可能炮制出一堆车轱辘话，长篇大论却没法落地。只要给角色划清边界、要求提供硬证据，这个模式就能稳妥运行。

2.6.4.1.2 它能解锁什么

成本类型	与人类队友合作时	与 AI 队友合作 + 此模式时
时间成本	跨职能的问题总是姗姗来迟，因为凑齐人手不仅慢，还得绕开办公室政治和日程冲突。	**你能提前把利益相关者的条条框框摆上台面**。后期的扯皮，变成了前期的验收标准和证据要求。
资源成本	专家时间金贵，成了早期设计反馈的瓶颈。	**你不必消耗专家宝贵时间，就能拿到第一版验收包**。专家只需验证这个包，不必从头亲手打造。
社交成本	大家都怕当出头鸟，索性把话憋在心里。	**你可以直言不讳征集反对意见，无需顾忌政治正确**。异议成了结构化的角色输出，而非个人恩怨。
注意力成本	团队后知后觉，等运维和支持的约束暴露，只能手忙脚乱打补丁、赶返工。	**通过强制要求早期提供跨职能验收证据，你能大幅减少后期的“惊喜”**。评审变成满足一份明确的验收包。

2.6.4.1.3 如何识别它（以及它替代了什么）

当工作牵扯到跨职能影响时，就该启用此模式：比如安全性、可靠性、合规性、性能、支持负载、迁移、向后兼容性或运维所有权。

当你反复栽进同一个坑里：发布后值班同事叫苦连天；安全团队临门一脚拦下；支持人员调试无门；或者维护团队以不符惯例为由直接拒收。

当社交成本高到让大家闭嘴，那些棘手问题迟迟没人敢提时，此模式正好派上用场。

本质上，它用前期的验收包和硬核证据，替代了后期的利益相关者扯皮。

2.6.4.1.4 风险及应对方法

成本类型	如果误用，会变成什么	控制点
注意力成本	角色输出泛滥成灾，文本淹没人却做不了决定。	约定范围、收敛输出、做好记录
时间成本	角色扮演无限扩大范围，交付拖延，真风险却没解决。	明确约定、限定边界
社交成本	人们把角色输出当成金科玉律，压制了真利益相关者，或制造了虚假的确定性。	明确约定、收敛输出、做好记录

约定
按风险高低选角色，别凭好奇心。

- 最多选 2 到 4 个角色。
- 给每个角色定死交付物：验收标准、必需工件和最低限度的证据。

限定
逼角色说短话、拿实据。

- 要求角色包必须简短。
- 如果某个角色发现了大问题，记下来后续处理，或明确调整范围。
- 设时间盒、限工件数量，防止角色无限蔓延。

基于证据委派
输出要的是证据，不是意见。

- “加监控”不是证据。要的是具体指标、告警条件、运行手册章节、金丝雀和回滚触发条件。
- “加测试”不是证据。要的是具体的边界测试和确切的验证命令。

收敛并记录
整合成一个统一的验收包。

- 突出显示现在要做的和明确推迟的。
- 记录用于闭环的验证命令。
- 把包和工作存一起，方便评审者查看。

2.6.4.1.5 示例交互流程

- 人类：“扮演值班负责人。生成一份验收包：什么情况会触发告警、需要哪些遥测和运行手册更新、以及拿出什么证据证明这能操作。”
- AI 队友：“值班验收包在此，包含必需工件和证据。”
- 人类：“现在扮演安全评审员。生成威胁说明、缓解措施，以及证明措施有效的检查项。”
- AI 队友：“安全验收包在此，附带证据。”
- 人类：“整合成一份验收包。标出哪些现在做，哪些往后放。然后只执行满足这个包所需的内容。”
- AI 队友：“合并包已生成。执行计划中。每个需求都有对应的证据包。”

2.6.4.1.6 起作用的原理

验收得看场景和证据，不能靠功能清单盲目乐观。SAAM 和 ATAM 这些方法之所以存在，就是因为真实系统的成败取决于利益相关者的场景——尤其是非功能性场景，比如安全、可靠、性能、可操作和可变性。角色扮演在模式层面用了同一套逻辑：每个角色生成一份与其实际解阻条件挂钩

的验收包，而不是模糊的“关注点”列表，并且这个包基于具体的场景和检查项。

尽早暴露跨职能约束，趁现在还能改。玻姆的风险驱动和螺旋模型强调先消除最大的不确定性，此时改动的成本还低。角色扮演就是在工作固化前，摸清安全、SRE 或维护人员的需求，减少后期利益相关者冲突，同时也没妄想取代真正的评审。

边界假设是“惊喜”的藏身之处，得在集成逼出问题前让它现形。布鲁克斯的集成教训在这里体现为“后期利益相关者冲突”：失败很少发生在一个职能内部，而常出现在所有权接缝、接口、运维职责、向后兼容性、支持工作流和上线约束这些地方。角色扮演逼着每个角色明确指出什么会出岔子、必须有哪些工件、需要什么证据，把隐晦的边界假设变成可评审的验收标准。

反馈循环必须把反对意见变成可证伪的检查项和可控的工件。戴明式的反馈纪律直接适用：角色不是“提点想法”，而是要求可验证的证据。“加监控”只是个愿望，而指标、告警条件、运行手册条目、金丝雀计划和回滚触发条件才是一个可控的系统。只有当验收包产出可验证、具体的工件和能闭环的证据时，角色扮演才能真正发挥杠杆作用。

协调成本决定了评审时机，所以要在不引发协调蔓延的前提下降低交易成本。科斯的交易成本视角解释了为何跨职能评审总来得晚：凑齐合适的人在政治和后勤上代价太高。AI 队友消除了这种成本，但这个模式通过限制角色、限定输出、保持结果可操作而非演变成无尽的相关方模拟，把廉价意见转化成了廉价的验收包。

异议必须去个人化，才能在截止日期压力下保持可用。埃德蒙森（Amy Edmondson，心理安全理论提出者）的心理安全理论解释了为何团队怕被贴上”绊脚石”标签时，跨职能反对声音就消失了。角色扮演让异议变得常规化、角色化：反对意见以利益相关者要求的形式出现，而非个人冲突，使

讨论聚焦于证据和风险，同时保留人类判断作为最终拍板依据。

判断力可以通过结构化的换位思考来训练，而非死磕合规检查清单。从教育角度看，角色扮演是规模化决策的脚手架：团队练习不同利益相关者如何考虑风险和验收，验收包则充当了评分标准。久而久之，工程师内化了可操作性、安全性、可维护性和支持性方面的“良好”标准，从而提升了社会技术判断力，而不只是制造文书工作。

2.6.4.2 魔鬼代言人

2.6.4.2.1 是什么

魔鬼代言人，说白了就是让 AI 队友专门唱反调。目的不是刁难，而是把那些藏着掖着的假设都逼出来。最终产物不是一场辩论，而是一份简短清单：列出最有力的几条反对意见，并附上能摆平每条意见的证据。这套玩法适用于任何决策环节——无论是需求、计划、设计、任务分解、风险评估、QA 策略、上线计划，还是代码变更。

它的威力来自速度快、人情包袱轻。但得小心，别陷入为反而反的噪音里，或者没完没了的批评。只要把反对意见框定范围，并转化为决定性的检查点，这个模式就能安全运转。

2.6.4.2.2 解锁了什么

成本类型	与人类队友	与 AI 队友 + 此模式

续下页

（续）

社交成本	大家为了避开冲突或顾及面子，往往弱化反对意见，结果异议要么姗姗来迟，要么干脆石沉大海。	**你能获得直来直去的反对意见，毫无人情负担**。异议成了规定动作，而非个人恩怨。
时间成本	团队常被一个看似合理的故事说服，直到集成或上线后，才发现假设有漏洞。	**你能更早揪出那些被忽视的假设**。反对意见在投入扩大前，就变成了检查点。
注意力成本	评审时争论不休，因为假设从未摆上台面，大家只能各说各话。	**你能把精力集中在决定性的检查上**。反对意见变成了可验证的主张，而非无休止的扯皮。

2.6.4.2.3 如何识别它（以及它替代了什么）

当你觉得决策顺利得有点心虚时，就该用它了：大伙儿一致同意，故事听起来严丝合缝，可你手里还没有实锤证据。

当社交成本高到没人敢提异议、谁也不愿当扫兴鬼时，这模式就能派上用场。

当后果严重不对等时——失败代价高昂、公开丢脸或覆水难收——这模式更是良药。

它用一系列可证伪的反对意见（这些意见最终会变成检查点），取代了过早达成共识和盲目乐观的执行。

2.6.4.2.4 风险及应对方法

成本类型	如果误用，会翻转成什么	控制点
时间成本	陷入分析瘫痪，批评没完没了，决策迟迟难产。	限定范围，收敛并记录
注意力成本	团队淹没在一堆无解的反对意见里，彻底失去焦点。	限定范围，订立契约
社交成本	形成一种消极文化，大家不再主动提议。	订立契约，限定范围

契约
先定好产出规矩：

- 最多三到五条反对意见，按影响力排序。
- 每条必须包含：背后的假设、假设错误时会引发的失效模式，以及最小的决定性检查。

限定
为对抗性审查设定严格的时间盒：

- 限制反对意见的数量。
- 限制所用时间。
- 一旦有足够证据支撑决策，就立刻停止。

基于证据委派
立刻将反对意见转化为验证计划：

- 如果一条反对意见无法快速验证，就建议一个替代信号或一个缩小范围的实验。

- 如果当下解决成本太高，就把它记作明确的权衡，并加上护栏：例如上线关卡、功能开关、金丝雀发布、监控措施或回滚触发器。

收敛并记录
做出决策，并白纸黑字记下原因：

- 哪些反对意见被证据解决了。
- 哪些被接受为权衡。
- 哪些控制措施兜住了剩余风险。

2.6.4.2.5 示例交互流程

- 人类："给当前方向挑刺。列出前三项反对意见，每条都要指出其假设和最小检查。"
- AI 队友："三条反对意见在此，附上假设、失效模式和决定性检查。"
- 人类："为最关键的那条反对意见，选一个最快、最关键的检查。明确定义成功和失败标准。"
- AI 队友："实验计划和确切的验证方法或信号已就绪。"
- 人类："执行检查。然后更新**验收包**，说明我们解决了什么、接受了什么，以及加了哪些控制措施。"
- AI 队友："证据包和更新后的决策记录在此。"

2.6.4.2.6 起作用的原理

好决策需要的是可证伪性和决定性检查，而不是自信的故事会。魔鬼代言人之所以存在，是因为一个听起来连贯的故事，在成真之前就让人觉得是真的。所以这个模式强行要求可证伪：反对意见必须表述为假设，而假设必须通过决定性检查来验证。这样一来，分歧就转化成了验证工作——这才是唯一值得在规模化时投入成本的分歧，并且它天然契合戴明式的反馈循环：进展需要那些能证明你错了的检查。

在还能轻易回头时，尽早消灭最大风险。玻姆的风险驱动理论解释了排序逻辑：先找出你当前忽略的不确定性，然后在投入倍增之前，用最小的实验或检查把它搞定。后期的意外往往代价惨重，因为它们出现在集成之后，此时已很难回头。因此，这种模式是一种廉价手段，强制实行“风险优先”的排序，而不是盲目乐观地执行。

偏见和群体动态让过早达成一致的概率极高，所以要把有限制的异议制度化，且不引发人际摩擦。卡尼曼和特沃斯基（Daniel Kahneman & Amos Tversky，行为经济学奠基人）的研究就像给直觉、锚定效应、可得性启发和过度自信贴上了警告标签，它们让第一个看似合理的答案显得天经地义。贾尼斯（Irving Janis，群体思维理论提出者）的群体思维研究则解释了为什么在交付期限压力下异议会消失——因为提异议有地位成本。魔鬼代言人把反对意见分配成一个角色并加以限制，从而产生结构化的异议，让隐藏的假设浮出水面，同时避免把分歧变成人际冲突。

批评必须转化为具体的控制措施，而不是沦为哲学辩论。事前剖析之所以有效，是因为它们把模糊的焦虑转化成了可以预防的具体失败故事。魔鬼代言人是其操作版本：每条反对意见都指明一个失效模式，并立刻要求能预防它的最小决定性检查或护栏。产出是验证计划加风险兜底计划，而非空谈，这提高了不确定下的决策质量。

对抗性循环必须设计成能收敛的，否则只会空转。戴明式的循环纪律至关重要，因为对抗性审查很容易变成无休止的批评，尤其在试错成本低廉时。只有当循环被限定（限制反对意见数量、设定时间盒、定义“证据足够继续前进”的停止规则），并且当风险被接受而非消除时明确记录权衡，这个模式才能保持高效。

把反对意见视为场景压力测试，并明确记录权衡。像 SAAM 和 ATAM 这类架构评估的传统，就是将决策视为场景间的权衡，而非寻找唯一最佳答案。魔鬼代言人以一种轻量级

方式借鉴了这种思路。反对意见成了场景压力测试，团队要么用证据消除风险，要么接受风险并明确设置护栏。这让决策记录保持诚实，也让后续评审更快。

批判性思维可以教成基于证据的分歧，而非个人冲突。从教育角度看，这是建设性异议的脚手架：批评必须有根据、有限制，并且与可能改变结论的证据挂钩。实践这个模式的团队，时间长了会培养出更好的判断力，因为他们学会了区分个人偏好和可证伪的失效条件，并且团队学会了将局部选择与系统结果联系起来，而不把分歧升级为人际斗争。

2.7 集群 D：复制成本近乎为零

2.7.1 是什么

有了 AI 队友，复制几乎不要钱。你可以轻松启动并行工作流、跑同一个工件的多个草稿，或者要求几种互相较劲的方法，完全不用操心时间、预算或人情世故。这套适用于需求文档、分解计划、QA 检查清单、事件运行手册、迁移策略、评审包，以及智能体软件工程的各种工件。

关键在于，复制产出结果的速度，远快于团队能消化收敛的速度。如果你不设计好收敛机制，就甭想获得杠杆效应，只会得到一团乱麻。在这个集群里，能力是复制，纪律是收敛：干净利落地合并兼容的工作，在有区分度的证据下做出选择，然后毫不留恋地抛弃其余选项。

2.7.2 解锁了什么

复制能大幅压缩时间表，把过去卡在单个人类瓶颈后面的工作并行执行。它还能帮你降低风险，让你亲自尝试真实的替代方案，而不是光在会议上打嘴仗——这在需求模糊或系统耦合度高、二阶效应很可能冒头时尤其管用。

复制还免除了“多要几个选项”的人情税。在人类团队里，“你能不能再试试 X 方案?”这种请求常常让人觉得成本高或有政治负担。但对 AI 队友，要求二次尝试、反提案或重写，完全可以成为默认操作。

真正的约束转移到了收敛环节。合并并行工作、保持评审的可读性、在互相竞争的故事间做决定，以及记录选择原因，这些变成了实打实的工作。如果领导者不建立起收敛习惯，复制只会让团队被海量的选择和集成工作淹没，反而更慢。

2.7.3 四元律速览

- 增强：并行能力、期权价值、实验速度
- 取代：顺序规划以及“必须预先选定一个故事”的执念
- 重现：有时间盒的探索和基于证据的决策
- 逆转：合并地狱、评审过载，以及因缺乏收敛设计导致的注意力涣散

2.7.4 此集群中的交互模式

- 并行分解
- 可弃赌注，证据决定

2.7.4.1 并行分解

2.7.4.1.1 是什么

并行分解，就是你有意把一个产出拆成多个并发的“流”。每个流都得边界清晰、合并计划明确。AI 队友们可以并行开干，但关键在于你设计的“接缝”——得让这些流最终能顺利汇合，别互相打架。这么干的好处是提速，代价则是你得把边界和验证搞得一清二楚。

风险嘛，不外乎编辑撞车、隐藏的耦合，还有评审者被活活累垮。只有当各流互不干扰、每个流都有明确的验收目标和验证步骤、并且汇总是增量式推进时，这模式才算稳当。

2.7.4.1.2 它能解锁什么

成本类型	与人类队友合作时	与 AI 队友合作 + 此模式时

续下页

（续）

时间成本	工作排成单线程长队：先实现后测试，先代码后文档，功能做完才重构，评审完了再改意见。	**你能靠安全的并行来压缩周期时间**。以前只能串行干的活儿，现在可以并发推进，还能保证合得回去。
资源成本	想并行？就得加人，协调成本也跟着水涨船高。	**你无需反复协调日程就能增加人手**。稀缺资源变成了评审和汇总环节，而不是执行能力。
注意力成本	并行往往带来合并冲突和协调开销，省下的时间转眼又赔了进去。	**你能让汇总过程保持可控**。清晰的边界、严格的隔离和增量合并，能有效防止注意力成本爆炸。

2.7.4.1.3 如何识别它（以及它替代了什么）

当你看到任务排起长队，或者工作能自然按边界拆分时，就用这招：比如独立的软件包、API、功能开关、工件或者验证任务。

当“干等”的成本已经高过并行协调的麻烦，并且你能说清每个流“完成”到底长啥样时，此模式正好上场。

它用一套有边界、受验证约束的并行流，取代了两种糟糕的旧模式：一种是慢如蜗牛的串行交付，另一种是一团乱麻的并行混乱。

2.7.4.1.4 风险及应对方法

成本类型	如果误用，会引发什么	控制点
注意力成本	合并冲突不断，评审不堪重负，协调乱成一锅粥。	划定边界、有序汇聚、做好记录
时间成本	“集成债”抵消了速度收益，还招来一堆后期返工。	明确约定、有序汇聚、全程记录
资源成本	人类不得不亲自下场“救火”，解决冲突。	划定边界、证据说话

明确约定
给每个流定下明确的验收目标。

- 允许改动哪些部分。
- 绝对不能碰的接口约束。
- 这个流最终要达成什么。
- 能最快证明其安全性的验证方法。

如果你连每个流的验收目标都说不清，那并行化还是先缓一缓吧。

划定边界
按自然的接缝来分解，别按待办事项列表硬拆。

- 流可以是代码层面的接缝，也可以是工件层面的：比如独立的测试计划流、QA 自动化流、文档与发布流、保持语义不变的重构流。
- 用技术手段隔离各流：比如独立的工作目录或分支，确保编辑内容不会在各流间“串门”。

- 限制同时活动的流数量，别让注意力成本失控。

证据说话
要求每个流提交一个“合并就绪包”。

- 说明这个流具体改了哪些地方。
- 附上该流的验证步骤和结果。
- 针对验收目标所做的自我检查。

有序汇聚与记录
采用增量式合并。

- 一次只合并一个流。
- 每次合并后，必须运行规定的验证。
- 如果验证失败，立马停下修复，然后再合并下一个。
- 更新**计划台账或记录**，清晰说明哪些已完成、哪些还待办。

2.7.4.1.5 示例交互流程

- 人类：“把这产出拆成三个流。每个流都要交代清楚：能改哪些文件、接口有啥限制、验收目标是什么、怎么验证。”
- AI 队友：“三个流的拆分计划已就位，边界和验证方法都包含在内。”
- 人类：“批准。每个流在各自的工作空间里执行。完成后，每个流交一个合并就绪包上来。”
- AI 队友：“流 A 的合并就绪包和证据。流 B 的合并就绪包和证据。流 C 的合并就绪包和证据。”
- 人类：“建议一个能减少冲突的合并顺序。然后一个一个合，合完一个就验证一次。”
- AI 队友：“建议的合并顺序如下。这是每次合并后的验证结果。已更新关于完成情况和剩余工作的记录。”

2.7.4.1.6 起作用的原理

只有把“顺利汇聚”当成头等大事来设计，并行才能真正提速。布鲁克斯在《人月神话》里的教训，在智能体时代愈发尖锐：盲目增加贡献者并不会线性提升进度，因为集成、沟通和概念完整性很快会成为瓶颈。并行分解的目标是追求“验证下可合并、没意外”的并行工作，而不是为了并行而并行。这既是社会技术系统的一种姿态，也是实实在在的编码策略。

边界是安全并行的基本单元，所以流必须由“接缝”来定义，而不是任务列表。帕纳斯的信息隐藏原理解释了为何边界优先如此重要：稳定的接口能隐藏内部变化，让并行流可以各自推进而不会语义冲突。当多个流共享可修改的表面时，耦合就会以合并冲突和诡异的行为错乱形式出现。所以，此模式坚持要求每个流都必须有明确的“可改动范围”和接口约束。

工作怎么拆，架构就会怎么长，所以协调结构必须贴合真实的接缝。康威定律在这里就是个警告标签：在没有真实接缝的地方强行拆分工作，会产生社会性的割裂，这种割裂最终会印刻到代码库里，变成尴尬的耦合和反复发作的集成痛。AI 队友让瞬间拉起多个“小团队”变得轻而易举，因此，严守边界纪律就成了关键控制手段——让你能增加贡献者，而不至于继承一堆架构烂摊子。

在小批量、快速集成和廉价验证的环境下，汇聚才能保持稳定。肯特 · 贝克的极限编程思想和马丁 · 福勒关于持续集成的著作，都抓住了其中的精髓：尽早集成、频繁集成，并且让验证循环足够廉价，可以反复运行。这正是本模式强调流隔离、增量合并，以及每次合并后必须运行验证命令的原因——避免所有冲突和问题在“大爆炸”式汇聚时一次性暴露。

想要交付可预测，就得有流程纪律：限制活动流数量，并根据验证节奏来安排合并。利特尔定律和看板式的流程思维

告诉我们，无限制的“在制品”只会增加周期时间和不可预测性——协调和等待的成本增长远快于实际进展。只有当活动流数量受限、每个流目标明确、并且汇聚由验证结果（而非任务堆积）驱动时，并行分解才能真正起作用。

风险管理本质上是排序问题，所以合并顺序应该优先消除高风险、保持高可逆性。贝姆的风险驱动交付思想体现在合并顺序的选择上：如果把风险最高或耦合最紧的流留到最后合并，那么后期的集成会迫使你返工所有东西。本模式坚持采用能减少冲突、尽早消除风险的合并顺序，这是实用的风险管理，不是纸上谈兵。

如何分解，是一项可传授的团队技能，能在大规模协作中提升整体判断力。从教育角度看，这种模式训练工程师按“接缝”而非“待办列表”来思考分解，培养了关于接口、不变量、所有权和验证的共享推理能力。长此以往，团队会变得更高效，不是因为他们更拼命，而是因为他们学会了设计那些能“干净”汇聚的工作。

2.7.4.2 可弃式赌注，证据决定

2.7.4.2.1 它是什么

可弃式赌注是一种工作习惯：当不确定性确实存在时，向 AI 队友请求多个独立尝试，然后运行一个判别性检查，从中选出最佳方案。赌注可以是一次分析、一次诊断、一个计划或一个原型。重点不在于保留所有输出，而在于低成本生成选项，再依据证据收敛——要么选中一个尝试，要么在检查支持下，跨尝试“取长补短”，组合成一个混合方案。

风险在于选项过载和原型泄漏：一个本该丢弃的探索性实现，可能仅仅因为已经存在且截止日期临近，就悄无声息地溜进了生产版本。要安全运用这种模式，就必须事先定义好判别性检查、限制尝试次数，并严格执行丢弃。混合挑选是团队真正发挥杠杆作用的地方，但也带来了新的集成风险：如今的合并工具，可没打算应付如此丰富的选择。

2.7.4.2.2 它解锁了什么

成本类型	与人类队友合作时	与 AI 队友合作 + 此模式时
社交成本	要求多份方案感觉像额外负担，还可能牵扯人际政治。	**结构化异议可以成为家常便饭**。多个尝试变得理所应当，且不会引发人际摩擦。
时间成本	团队陷于文字争论，死守一个听起来合理却脱离实际的故事。	**你可以依据证据而非空谈来做选择**。一次小小的判别性检查就能终结争论，避免后期意外。
注意力成本	比较多个选项令人疲惫不堪，拖慢决策。	**你可以把比较范围控制在合理区间**。尝试次数有限，并靠检查而非个人好恶来收敛。
资源成本	制作多个原型会消耗稀缺的人力工时。	**你可以在不消耗专家宝贵时间的前提下进行探索**。成本转移到了计算和少量人工判断上。

2.7.4.2.3 如何识别它（以及它替代了什么）

当某个方案听起来过于顺滑、系统耦合紧密或犯错成本极高时，就用这招。典型信号包括：需求模糊不清、反复出现“不试试怎么知道”的论调，或是后果不可逆——比如数据形态、API 形态或操作影响范围一旦成形便难以回头。

当团队分歧在社交层面难以处理，大家都避免去做那个“唱反调的人”时，也用这招。

它用有限的选项和一个能决出优胜者或拼出混合方案的判别性检查，取代了过早拍板和无休止的扯皮。

2.7.4.2.4 风险及应对方法

成本类型	若误用，会翻转成什么	控制点
注意力成本	选项过载、比较疲劳，以及人类在拼接部件时若未留好“接缝”，导致问题迟至集成阶段才暴露的“弗兰肯合并”风险。	约定、限制、收敛并记录
时间成本	生成起来没完没了，就是不收敛。检查永远躺在那里不运行。	约定、限制、收敛并记录
资源成本	计算和迭代螺旋上升，因为尝试次数不设上限，原型四处蔓延。	限制
时间成本	原型泄漏。一个本该丢弃的探索性实现，未经适当验收就成了生产版本。	约定、收敛并记录

约定
明确决策问题和验收目标。

- 我们到底在选什么。
- 怎样才算明显胜出。
- 先定义判别性检查。没有它，你只是在收集一堆意见。

限制
强制执行独立性、上限和可弃性。

- 要求 2 到 3 个独立尝试。其中必须有一个能反驳主流选项。
- 为赌注设定时间盒。
- 限制任何原型允许的“表面积”（影响范围）。
- 将工件标记为“可弃”，并保持隔离。

基于证据委派
强制进行结构化比较。

- 要求一份“异议与组合表”：为每个尝试列出其假设、最强组件、混合所需接缝、出错影响，以及那个能决出胜负或混合方案的判别性检查。
- 运行判别性检查，并返回一个包含结果、能选出优胜者或混合方案的证据包。

收敛并记录
保留一个，或保留一个混合方案，丢弃其余，并记录原因。

- 记录决策及其证据。
- 如果保留混合方案，将其视为一个新选项：说明你创建的“接缝”，并运行一个专门覆盖这些接缝的集成检查。
- 如果胜出的是探索性实现，那就按照正常的验收标准和验证流程，为生产环境重写一个干净的版本。

2.7.4.2.5 示例交互流程

- 人类：“给我三个独立尝试。其中一个必须是强硬的反驳。每个都得说明自己的假设和最担心的失败模式。”
- AI 队友：“三个尝试已就位，包含假设和最担心的失败模式。”
- 人类：“创建一份异议表，并提出那个最小的、能一锤定音的判别性检查。”

- AI 队友："异议表和推荐的判别性检查在此。"
- 人类："批准。作为可弃式赌注，隔离运行该检查。返回证据。"
- AI 队友："证据包已返回，内含结果。优胜者或混合方案已选出。决策记录已起草。"
- 人类："丢弃落选者。如果胜出的是探索性实现，就通过正常验证流程为生产环境重写。更新记录。"

2.7.4.2.6 起作用的原理

不确定性必须靠判别性证据解决，而非听起来最动听的故事。这种模式将不确定性转化为证据可以裁断的问题：一个"听起来更好"的选项不是决策规则，但判别性检查是。正因如此，它坚持在生成选项前，就先定义那个能让你改变主意的最小检查；也正因如此，你保留的是"决策加证据"（如果组合了混合方案，还包括保留了哪些部分），而不是导致该决策的文字论述。这是一种可证伪、可扩展的姿态。

在可逆性高时，尽早消除最大风险。玻姆的风险驱动思维模式直接适用：可弃式赌注通过在实际变更尚小、成本低廉且可逆时尝试真正的替代方案，来压缩获取洞察的时间。重点不是为了探索而探索，而是在承诺膨胀之前，降低犯错的成本。

独立性是避免集体盲区的方法，因此赌注必须真正独立，且其中一个必须是对抗性的。设计多样性和独立验证的传统之所以存在，是因为失败往往源于共同的假设，而非孤立的错误。如果所有尝试共享同一个思维框架，它们就会共享同一个盲区。只有当尝试真正独立，且其中一个明确反驳主导叙事、将分歧呈现为可检验的假设时，这种模式才能创造价值。

廉价的替代方案能防止早期锁定，但前提是收敛由证据而非惯性强制实现。西蒙的有限理性解释了为何团队在压力下会抓住第一个连贯的叙事；卡尼曼和特沃斯基则解释了

为何随着投入增加，这个叙事会感觉越来越正确。可弃式赌注通过廉价生成替代方案，然后依靠检查（而非妥协或叙事惯性）来强制收敛，从而打断这种锁定。

选择只有在有时间限制且可转换时才有价值，因此应将赌注视为带有明确转换规则的真实期权。从管理经济学角度看，可弃式赌注是一种期权：你支付有限成本来购买信息，以规避更大的错误承诺。只有当期权有时间限制且转换规则明确时，这种模式才有效——保留一个选项是因为证据选择了它，否则“期权”就会变成永久性的半成品，持续消耗注意力。

反馈循环必须通过严格约束和“为丢弃而设计”来防止反复折腾和原型泄漏。戴明式的循环纪律体现为具体上限：限制赌注数量、限制时间、限制“表面积”，并要求停止规则。两种可预见的失败是无休止的比较和原型泄漏——即一个本该丢弃的探索性实现因其存在而成为生产版本。因此，这种模式坚持“为丢弃而设计”，并在证据选出优胜者后，为生产环境进行干净的重写。

当异议被去个人化并立即转化为检查时，结构化异议能提升决策质量。贾尼斯对群体思维的研究和埃德蒙森对心理安全感的研究都指向同一个现实：异议有价值，但社交成本很高，尤其是在截止日期临近时。AI 队友降低了这种社交成本，而这种模式则通过使异议常规化、结构化，并将分歧转化为检查来加以利用，从而在不引发人际摩擦的情况下提升决策质量。

2.8 本章小结与后续内容

AI 队友是拥有特殊优势的队友。它们改变了软件工程的经济学，但真正的转变在于互动方式，而非仅仅是代码生成。本章介绍的模式，正是为了在需求、规划、分解、评审、QA、

调试、指导、决策以及日益增长的智能体软件工程工件等方面，与这种新型队友有效合作。

- **集群 A** 表明，AI 队友的不知疲倦和非评判性，能让你轻松迭代、打磨工具，自发构建任务所需的任何支持，而无需消耗人情或权衡投入——前提是你用验收标准、停止规则和证据来约束循环。
- **集群 B** 表明，强大的沟通能力可以大幅降低“翻译”成本和重新对齐的成本——前提是你通过将主张锚定在约定、表征和证据上，来防范过度自信的压缩。
- **集群 C** 表明，广博的知识储备使得生成选项和提出异议变得廉价——前提是你要求证据，并将批评转化为决定性的检查，而非徒劳的争论。
- **集群 D** 表明，复制能力让并行实验成本骤降——前提是你从一开始就为收敛和集成而设计。
- 在所有集群中，控制点一以贯之：约定、限制、基于证据委派、收敛并记录。正是这些控制点，使得杠杆效应能够持续叠加，而非轰然倒塌。

第 3 章将转向硬币的另一面：AI 队友可预测地不擅长什么，以及如何设计信任机制，使得大规模下的随机性工作既安全又可信。


开
关
自主性
证据
信任 = 证据 / 风险
氛围
最大
控制点：为可靠系统调校平衡

3 AI 队友的悖论

3.1 永不成长的初级开发者

“想象一下，你招来一位才华横溢的初级开发者，他每天早晨精神抖擞地来上班，却把昨天的教训忘得一干二净。他干劲冲天但缺乏判断，能在你喝杯咖啡的功夫里，唰唰唰写出上千行代码。现在，再想象一下要和他搭档完成一个为期一年的项目。”

这大致就是如今与 AI 队友协作的体验。只不过，AI 产出的代码通常比大多数初级开发者写的更整洁、更快，也更像那么回事。正因如此，经验丰富的开发者与 AI 协作时，风险感知完全错位了。当一段代码能编译、读起来顺畅、测试还能通过时，你本能地就想相信它。然而，这种协作模式恰恰缺少了让人类团队日益可靠的那些要素：持续的学习、团队间的默契，以及稳定的判断力。

AI 队友一门心思想要立刻讨你欢心，表现得独立又果断。人类当然也想留下好印象，也会着急赶工、会靠猜测行事，但人类组织演化出了一套仪式和防护栏，以防糟心的点子真变成能跑的代码。反观 AI 这位“随机型队友”，它很可能——甚至极有可能——明天就信心满满地把同样的错误再犯一遍，除非我们设计的系统能阻止它。

好消息是，我们管理随机、易错的队友已有数十年历史，因为人类本身也是随机且易错的。软件工程这门学科，一直就是在不确定性中建立信任的学问。坏消息是，AI 以惊人的速度和规模放大了故障模式及其后果，结果你那旧的安全网突然就不够用了，哪怕它看上去还挺眼熟。一个基本事实直白得很：软件发布时总带着缺陷，过去如此，将来也一样。可信赖的工程无关完美，而在于建立一套机制——

定位根本原因、从中吸取教训，并防止整类问题再次发生。正因如此，在智能体软件工程时代，软件工程的重要性丝毫不亚于每行代码都靠人敲出来的年代。

这里有个切中要害的故事，能让你具体感受到其中的利害关系，因为这就是你在真实系统中每周都会见到的那种事。一个智能体发现了一个身份验证漏洞，完美地修复了它，运行了测试，遵循了标准，最后产出的变更读起来就像出自你最冷静的高工之手。你合并了代码，发布了更新，继续推进，系统似乎变得更好了；直到第二天，在一个全新的会话里，同一个智能体“重新发现”了同一个漏洞，并带着全新的“惊喜感”，同样信心十足地提出了完全相同的修复方案。核心成本不在于当下的错误；而在于学习的缺失，以及问题无限重复的必然性。

工程上的应对之道，不是让智能体变得更像人，也不是指望下一代模型能神奇地长出判断力。工程的本质，始终是在不确定性下控制随机性组件，我们对人类和物理系统早就是这么做的。因此，正确的做法是：将我们已经熟知的学科方法，重新应用到一种能以机器速度工作的“失忆型协作者”身上。错误总会发生，但可信赖的智能体软件工程意味着，我们拥有定位源头并避免复发的机制——理想情况下，不仅仅是避免那一个具体错误，而是杜绝其整个类别。

本章特意将视角停留在“一人一 AI”的层面，因为即便只有一个随机性贡献者，也足以产生那些看似随机混乱、实则可预测的故障模式。

3.2 本章为何令人乐观

一个充斥着 AI 队友弱点的章节，可能会引发一种错误的反应：“我干嘛要伺候这些?”正确的反应恰恰相反，因为这些弱点大多人类也有：上下文不完整、过度自信、沟通不畅、假设漂移，以及偶尔状态奇差导致严重失误的“水逆

日”。软件工程花了几十年时间从不可靠的组件中构建可靠性，而人类团队本身一直是这个可靠性问题的一部分；区别在于，我们无法对人类进行“编程”，也无法在他们脑子里强制执行流程。我们靠文化、培训和激励来应对，然后接受剩余的方差。

AI 队友改变了经济性，因为你可以将指导原则编码到工件和协议里，并让它们每次都按相同方式执行。你无需协商个性，也不必为每个新招募的 AI 队友重复同样的培训课；你可以一次写好，无限次应用。一个 AI 队友可以通过持久的指导来塑造，由具有明确检查点和上报触发器的确定性智能体软件工程流程把关，通过“合并就绪包”来保证质量，并通过“写下一次，重复应用”的组织学习来纠正偏差。更妙的是，当你打造出一个优秀的 AI 队友时，你可以复制它——这在能力扩展方式上，是一个结构性的改变。

别把“有弱点”和“能力弱”混为一谈。我们这项工作的独特之处，在于我们远离了行业对通用人工智能的执着；相反，我们在一个我称之为“功能性人工智能”的框架下运作。我们追求的并非一种抽象的、“完美”的智能。我们正在构建的队友，其原始能力已经超过了普通人类水平。我们常常自欺欺人，拿 AI 与虚构的完美理想比较，而不是与眼前那个混乱、真实的人类团队现状比较。“初级开发者”这个类比，指的是我们的风险校准和应对姿态，而非他们的原始潜力。我们的弱点清单之所以长，是因为我们正在以机器速度严谨地分类故障模式，而不是因为协作者能力不足。

我们的目标是通过设计一套针对其独特优势的系统，将这些 AI 队友提升到“超级开发者”的层面。我们常常忘记，发布可信赖的软件从来不是基于对人类的盲目信任；我们早已用流程、指南和工具的多层结构，将随机性的人类开发者包裹起来，以控制方差。通过将同样的工程严谨性适配到这种新的智能模式上，我们不仅能实现自动化，更能达到超人类能力的状态，同时仍能严格管控潜在风险。

3.3 布鲁克斯的视角：为何我们对 AI 队友如此苛刻

弗雷德·布鲁克斯在《没有银弹》中提出，软件工程的困难可归结为两种复杂性：本质复杂性（问题本身和现实世界所固有）和偶然复杂性（由工具、语言及附带摩擦产生）。AI 队友通过消除起草、搜索、重构、搭建脚手架和产出软件过程中的摩擦，摧毁了偶然复杂性。这是第 2 章强调的杠杆效应，真实不虚。剩下的，便是本质复杂性：模糊的意图、矛盾的约束、混乱的现实世界权衡，以及不确定性下的正确性。

本章探讨的许多弱点，恰恰出现在本质复杂性存在的地方：模糊性下的对齐问题、不确定性下的证据缺失，以及有限工作集下的协调难题。我们如此挑剔，并非因为 AI 队友比人类更差；而是因为他们运作的规模，使得一个普通的人类弱点可能在几分钟内演变成系统性事件。正确的心理模型不是“AI 队友不可靠”，而是“AI 队友是强大的随机性组件”，而强大的随机性组件需要工程化的控制闭环。这是后续所有内容的前提。

3.4 如何理解这四个悖论

这些悖论不是可以一笑置之的怪癖，也不是靠提示词就能消除的问题。它们是随机性队友的固有特征：能在不理解的情况下执行，能在无判断时发言，能在缺乏生活经验时海量输出。如果你把这些动态视为边缘情况，你将一直处于持续的惊讶状态。你的系统将趋向脆弱，因为你只是在修补症状，而非重新设计软件工程系统。

为了给这些观察理出条理，我们对每个悖论采用了统一的结构。这个框架让你能立即识别问题，并客观地向团队解释，而不是将其描述为神秘的玄学。每个悖论包括：标题与

标语、一个贴切类比、一套结构化的（在 AI 和人类队友中的）表现、人类先例、传统控制措施、AI 为何会放大此问题、一个可供想象并打断的“厄运循环”，以及与行为和控件直接相关的理论基础。重点不是为了理论而理论；而是为了表明这些都是有稳定补救措施的稳定模式。

现在，让我们逐一探讨这四个悖论。

3.5 悖论一：热切悖论

标题与标语

动作越快，理解越浅。

类比

“在错误赛车上忙活的 F1 维修站团队”
以惊人速度完美执行，却完全跑偏了方向。

表现何在

每当意图与行动之间存在差距时，这个悖论就会出现。因为 AI 队友就像一个热切的初级开发者，优先考虑立竿见影的进展，而非深入理解。失败不在于工作马虎；失败在于方向跑偏，而当输出看起来完整无缺时，方向错误更容易被忽视。在实践中，你应该将“热切”视为一种必须由系统加以约束的稳定姿态，而非指望通过要求队友“更仔细”来“修复”。这一点至关重要，因为队友速度越快，方向性错误在全局层面就越昂贵。

在 AI 队友中

速度优先于理解的行为

- 从模糊的请求直接跳到完善的实现，跳过了“你到底需要啥?”的确认环节。
- 当被要求“更好地处理支付”时，直接实现一整套支付系统，却不去问清楚你指的是错误处理、UI 清晰度、可观测性还是性能。
- 在提出一个澄清问题所需的时间里，交付 500 行完美代码，结果解决了错误的问题。

复杂性盲视行为

- 低估实现复杂度，把分布式系统问题当作字符串操作练习来处理。
- 信心满满地承诺“我只需要重构身份验证系统”，却完全没掌握其依赖关系网。
- 基于理想路径给出时间估算，忽略了构成实际工作大头的各种边角情况。

寻求认可的行为

- 宁可猜错，也不愿通过提问显得不确定。

- 用听起来合理的假设填补知识空白，而不是承认不确定性。
- 即使方向不明，也要通过行动展示进展，于是行动本身变得比目的地更重要。

在人类队友中

人类也常这么干，尤其在职业生涯早期。任何带过初级开发者的人，对这种模式都再熟悉不过。热切通常源于证明独立性的渴望、害怕显得太慢，以及将行动与进展混为一谈。区别在于，人类组织期望并依赖随着时间推移而进行的学习和校准，而一个随机性贡献者明天很可能重蹈覆辙——除非现有的工程系统强制其采取不同的行为。这正是“初级开发者”这个类比并非侮辱的原因；它是一个操作警告。

热切初级行为

- 那个重建了整个模块，而非仅仅修复你提到的拼写错误的实习生。
- 那个整个周末都在开发一个压根不需要的功能的新员工。
- 那个在修漏洞时顺手“改进”了架构的热切承包商。
- 那个实现了自己对需求的理解，而非挑战需求模糊性的初级开发者。

人类先例

我们管理热切的初级开发者已有数十年，每位高级工程师都有一箩筐至今想起仍觉尴尬的故事——这也正是这个教训如此重要的原因。关于实习生的传统笑话（“给他们安排一个需要两周的任务，这样他们就来不及搞砸重要的东西”）并非出于仁慈；它是从生活经验中衍生出来的隐性风险控制。在人类团队中，热切是通过包含指导、尴尬感和返工实际成本的反馈循环来塑造并最终缓和的。对于一个随机性 AI 队友，你应该假设这些循环不存在，除非你明确地设计它们。

传统控制措施

军队训练新兵时就懂了：你不会在第一天就把所有野战手册扔给新兵；你先教核心原则，然后随着能力增长逐步增加复杂度，并且强制执行决策边界，防止在高风险时刻即兴发挥。软件工程演化出了平行的控制措施，因为我们以惨痛代价学到了同样的教训，这些措施包括：导师制、代码审查、站会、冲刺规划、逐步增加的责任，以及一种文化氛围——让初级开发者看到高级工程师毫无心理负担地提出澄清问题。关键的人类洞察简单而朴素，这也正是它有效的原因：现在慢一点，是为了以后更快，因为返工成本会在下游激增。这些控制措施无关不信任；它们关乎将能量引导到正确的目标上。

为何 AI 会放大此点

痛苦不记教训。AI 不会在会话之间携带那些能改变明天行为的尴尬、遗憾或精力白费的记忆，因此同样的过度自信飞跃可能会一再发生，除非系统强制一个早期的澄清关卡。

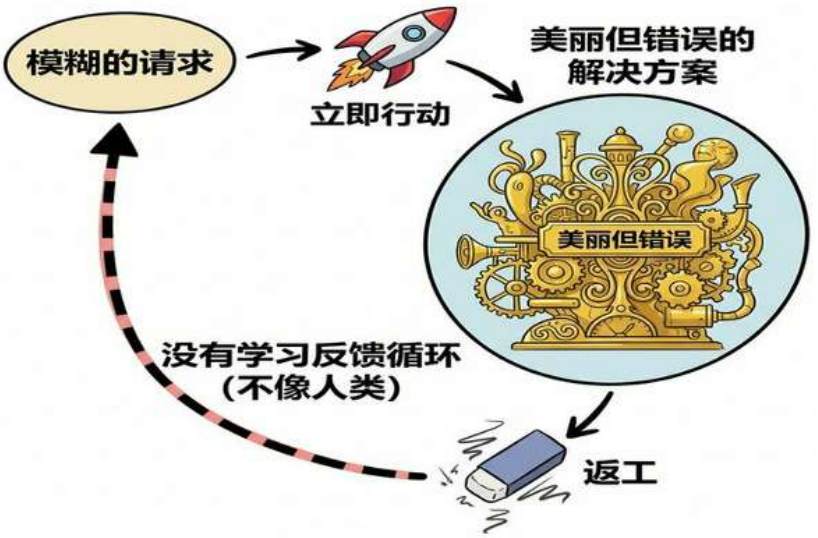
没有社会压力。AI 感受不到浪费时间的社会成本，也体会不到提出一个好问题的自豪感，因此你必须用明确的决策边界和上报触发器来替代社会校准。

速度倍增器。人类的误解通常在演变成灾难前就被发现，因为人类工作速度慢。AI 成了错误方向的倍增器：它能在一下午之内，将你提示词中的一个小模糊点，变成需要数周清理的错误技术债务，同时听起来还完全自信。

没有渐进信任。人类赢得信任，是因为昨天的良好判断换来了明天的更多自主权；AI 的会话通常每次都像第一天一样开始，除非你将记忆外部化——这打破了正常的“权责随能力增长”的模式。

无限的热情。人类会疲倦，会自然放慢节奏，这有时反倒阻止了将误解升级成架构重写；AI 可以无限期保持巅峰热切度，除非你限制其工作范围。

热切厄运循环



理论基础

在《**没有银弹**》中，**弗雷德·布鲁克斯**指出软件存在固有的本质复杂性，工具对此无能为力，唯有偶然复杂性尚可通过技术进步来削减。所谓急切性，是指智能体往往急于求成，试图通过立即编码来强行突破本质复杂性，结果却要在偶然复杂性上付出代价：反复返工、修修补补，以及最终不得不推倒重来的“美丽的错误方案”。**玻姆**（Barry Boehm，软件工程经济学先驱）的**螺旋模型**补充了其中缺失的纪律，它将工作框定为一轮轮的风险削减周期——即在展开大规模实现之前，先集中力量攻克最大的未知数。**需求工程**在实践中讲授了同样的道理：最昂贵的缺陷往往源于早期对意图

的误解；而一个快速执行的智能体，会以机器速度将需求缺陷注入系统，并用自信的论述和整洁的代码将其包裹起来。

卡尼曼（Daniel Kahneman，行为经济学奠基人、诺贝尔经济学奖得主）的**系统 1（快思考）/系统 2（慢思考）**框架，以及**马奇**（James March，组织决策理论大师）的**探索与利用**理论，道出了这条通道失败的原因：你还没通过探索搞清楚状况，就急匆匆开始了快速利用；整个系统在偏离目标时，仍让人觉得干得不错。阿吉里斯（Chris Argyris，组织学习理论先驱）提出的”熟练的无能”，则为这种结果贴上了一针见血的标签：高质量的执行，完全可能与错误的目标并存。正因如此，本章后续内容将“意图对齐”和“可测试的成功标准”视为一等控制手段，而非可有可无的礼节。

3.6 悖论 2：上下文悖论

标题与标语

提供的上下文越多，其使用效率反而越低。

类比

“只会听导航、不会看地图的司机”
每个路口都执行得完美无缺，对整个旅程却一无所知。

表现方式

当你试图用海量信息来弥补缺失的直觉，结果却发现信息越多、困惑越大时，这个悖论就现身了。交互通道变得嘈杂，冲突成倍增加，队友反而变得更不可靠——恰恰是因为你试图让它“知道”得更多。在实践中，最常见的败因并非 AI 队友“忘记”了什么，而是它无法可靠地区分轻重缓急。当上下文缺乏管理时，无论是人类还是 AI 队友，都可能因为选择了“看似合理”而非“真正正确”的规则而失败。

AI 队友的典型表现

信息过载

- 给了 47 条编码规范，结果应用了与第 7 条矛盾的第 43 条，导致代码自相矛盾。
- 在信息海洋中淹没，套用了第 200 段的过时模式，反而覆盖了第 3 段的关键指令。
- 把所有上下文都一视同仁，于是安全要害规则和格式偏好获得了同等权重。

规则混淆

- 分不清“绝不能提交密钥”和“应优先使用 const 而非 let”孰重孰轻。
- 对琐碎指令执行得一丝不苟，却把关键指令忘在了脑后。
- 遇到规则冲突时，随意选一个了事，而不是升级问题以求澄清。

上下文衰减

- 调查某个支线错误时引入的上下文，污染了后续工作，导致后来的认证/授权任务也带上了无关的错误处理逻辑。
- 任务 A 的上下文渗入任务 B，就像一道菜的调味料串了味，糟蹋了下一道菜。
- 把过去的临时决策奉为金科玉律，即便情况早已改变。

人类队友亦难幸免

人类对此并非免疫，这正是成熟团队学会整理信息而非倾倒信息的原因。信息过载是新员工文档常常失效、团队依赖渐进式披露和师徒制而非“通读一切”的根源之一。人类通过学会“忽略什么”来应对，这种过滤技能随着经验积累和社会校准而增长。AI 队友不会在多次会话间自动发展出这种过滤器，因此我们必须将优先级和相关性明确地编码进去。

人类的过载行为

- 收到 400 页入职文档的新成员，几乎什么也没记住。
- 开发者照着维基最前面的过时说明操作，因为它们排在最前面。
- 承包商花好几天读文档，却不肯提一个澄清问题。
- 初级开发者死守风格指南的条条框框，即使它们损害了代码可读性。

人类先例

我们早就明白人类的工作记忆和注意力有限，因此才发明了摘要、清单、渐进式入职以及分层递进的培训方式。米勒（George Miller，认知心理学家）的”7±2”法则并非精确计数，而是提醒我们容量有限，有限的容量逼迫我们必须整理信息。人类也通过生活经验培养出优先级意识，最终能

领悟到有些规则是铁律，有些只是偏好。AI 队友无法可靠地获得这种优先级感，除非你将它构建到系统之中。

传统控制手段

军事比喻在此依然适用，因为它是一个整理知识的绝佳范例：先教授核心原则，再按需补充细节，并且清晰区分“必须”、“应该”和“最好有”。软件团队通过以下方式达成同样目标：分层组织信息、渐进式披露、有导师告诉你哪份文档已过时、暗示什么才是真正重要的部落知识、随经验增长而形成的过滤能力，以及在关键时刻提供上下文的即时学习。这些控制本质上是信息架构控制，尽管它们常以社会化的方式呈现。当我们与 AI 队友合作时，同样的控制必须被显式化，而不能想当然。

为何 AI 会放大此悖论

缺乏智慧过滤器。人类通过经验和社会信号发展出对重要事务的感知，因此即便规则白纸黑字，他们也能忽略低价值条目；除非你编码优先级，否则 AI 很可能将“安全铁律”和“格式偏好”等量齐观。

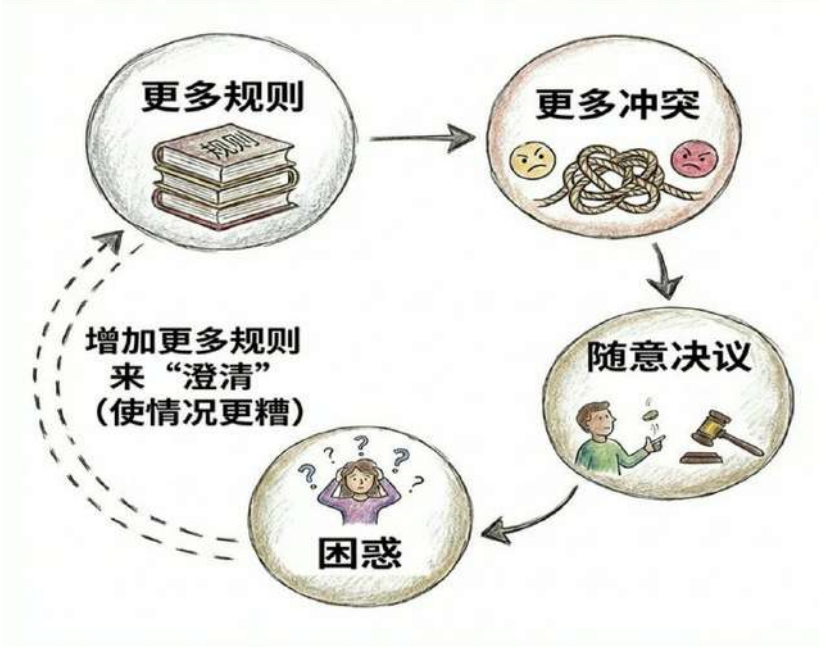
没有遗忘曲线。人类会淡忘过时的实践，而这种遗忘是一种特性，因为它清除了陈旧的规则；除非你明确整理哪些相关、哪些已弃用，否则 AI 可能会以同样的信心反复应用过时的规则。

缺失社会校准。人类可以向同事探听真正要紧的事，并领会那些未曾明言的优先级；AI 缺乏这种渠道，除非你创建明确的升级路径和优先级类别。

上下文污染的持久性。一旦无关细节潜入工作上下文，它们就可能扭曲后续的决策和优先级；如果没有刻意的重置、分叉或压缩，污染可能比引发它的任务持续更久。

字面化解释。人类将“优先使用 X”视为指导方针，并能在理由正当时违反它；AI 可能将其视为铁律，哪怕这会损害整体一致性，而累积的偏好最终会导致无法调和的冲突。

上下文厄运循环



理论基础

容量限制几乎是此处所有症状的根本原因，三大经典思想解释了为何更多上下文反而降低正确性。**米勒**强调了**工作记忆的限制**，**斯威勒**（John Sweller，认知负荷理论提出者）的**认知负荷理论**指出，当**无关负荷**挤占了任务本身所需的认知资源时，表现就会下降。**西蒙**（Herbert Simon，有限理性理论提出者、诺贝尔经济学奖得主）的**有限理性**则解释了决策者为何在过载时会满足于“够好就行”。这正是当优先级不明时，冲突被随意“解决”的方式：**行动者**选择了看似合理、而非全局正确的选项。在智能体工作流中，这种

“够好就行”的行为能以机器速度发生，并包裹在自信的论述之中。

设计理论则指引我们如何设计解药而不陷入流程泥潭。**帕纳斯**（David Parnas，信息隐藏原则提出者）的**信息隐藏**原则启示我们为智能体的工作上下文设计一个经过整理的接口；**迪杰斯特拉**（Edsger Dijkstra，结构化编程奠基人）的**关注点分离**原则警告我们，切勿将安全铁律、个人偏好、历史记录和临时推测混为一谈。**香农**（Claude Shannon，信息论创始人）的**信息论**提供了通信视角：当矛盾积累的速度快于约束厘清的速度时，增加文本只会提高熵、降低信噪比。这些理论基础证明了本章为何如此执着于固定的铁律、经过整理的工作集以及按需加载的上下文——因为一致性源于结构，而非数量。

3.7 悖论 3：隧道视野悖论

标题与标语

局部执行得越完美，全局失败得越惨。

类比

“为站立式办公桌精心打造大师椅的匠人”
对系统有着根本误解的完美执行。

表现方式

当 AI 队友只顾优化局部完美，却不理解更广泛的系统背景、集成要求或长期影响时，这个悖论便会出现。其产出在孤立环境下或许优美正确，但一接触现实就可能酿成苦果，这是最危险的一类错误。在实践中，隧道视野最明显的表现，就是将“完成”定义为本地测试通过，而非交付集成价值。当队友感知不到系统边界时，它就会只顾优化中心，却破坏了边界。

AI 队友的典型表现

局部优化

- 做出了一个完美的认证模块，却无法与现成的会话管理系统集成。
- 将一个函数的运行速度提升了十倍，同时弄垮了三个依赖服务。
- 构建了一个遵循独特模式的精美功能，与代码库其他部分格格不入。
- 编写了优雅的代码解决今天的问题，却为明天埋下了技术债。

系统盲视

- 没意识到另外三个功能正依赖着它要重构的 API。
- 把任务当成代码库里唯一在发生的事情。
- 在一个有十年历史的系统中工作，却假设一切从头开始。

- 做决策时不考虑部署、运维或维护的后果。

时间线短视

- 只为“代码写完”优化，而不是为“合并就绪”或“集成就绪”。
- 庆祝“本地测试通过”，却把集成测试抛在脑后。
- 思考的时间单位是“本次会话”，而非“本次冲刺”或“本季度”。
- 创造了当下完美、未来却无人能维护的解决方案。

人类队友亦难幸免

人类也会犯这种毛病，所以我们才有那么多致力于集成的仪式。隧道视野无关道德，它是局部激励和局部可见性占主导时的必然结果。成熟团队刻意设立角色和仪式来保持系统视角，因为个人无法将所有事情记在脑中。AI 队友不会自发形成那种系统视角，除非通过指导方针注入并由验收标准强制执行。

人类的隧道视野行为

- 后端开发者在未通知前端的情况下“修复”了 API。
- 功能开发者对即将到来的架构重构视而不见。
- 专家为了优化自己的模块，不惜牺牲整个系统性能。
- 认为“在我机器上能跑”就等于完成的开发者。

人类先例

隧道视野的存在，正是团队设立架构评审委员会、集成测试、系统设计文档以及负责维护全局视角的技术负责人的原因。成熟的“完成”定义之所以包含集成、文档和运维就绪，也源于此。人类团队往往吃尽苦头才明白，局部正确性并不能保证系统正确性。在智能体工作流中，这种痛苦无法累积为记忆，因此必须由系统本身承载这些教训。

传统控制手段

建筑行业几百年前就明白了这个道理：木匠不可能在不与水管工、电工协调的情况下造出完美的橱柜，因此总承包商作为明确的集成角色应运而生。人类团队通过站会（暴露进度冲突）、冲刺计划（让依赖关系显性化）、代码审查（发现系统级问题）、技术负责人（维护概念完整性）、集成测试（尽早发现故障）以及回顾会议（将集成失败转化为学习经验）来对抗隧道视野。一个强大却隐形的控制是物理和社交的邻近性——人们无意中听到“我也在改那个 API”并进行调整。这是一个真实的协调机制，而非什么文化鸡汤。智能体协作通道缺乏这些环境信号，除非你用明确的工件来替代它们。

为何 AI 会放大此悖论

缺乏周边感知。人类通过旁听对话、观察他人工作内容来获取系统信号；AI 在孤立环境中运行，除非你通过工件和协调数据向其注入系统视图。

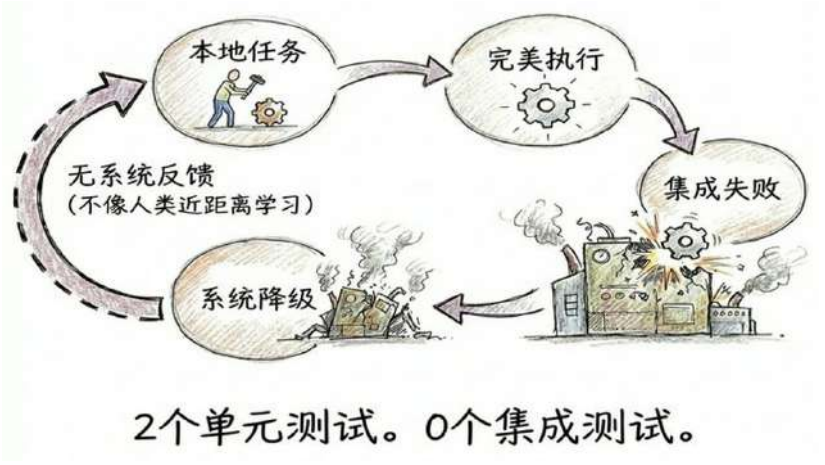
没有部落知识。人类能学会那些不成文的规矩，比如“发布周别碰那个模块”；AI 不会积累这些教训，除非你编码好边界和时间约束。

缺乏时间感。人类以发布周期和未来后果来思考，因为他们经历过集成的痛苦；AI 可能将每个任务都视为永恒的当下，除非你将时间维度强制纳入完成定义。

完美的局部专注。人类的分心有时是一种安全机制，因为它能暴露横切问题；AI 可以信心十足地在局部最优里深挖数小时。

缺乏集成直觉。经验丰富的工程师常常能预感到集成痛点，因为他们记得过去的失败；如果没有工程化的记忆和明确的集成检查点，AI 没有理由发展出那种直觉。

隧道视野厄运循环



理论基础

康威定律点明，系统结构总与沟通方式如影随形：你怎么协调，系统就长什么样。单个 AI 队友若只埋头于狭小任务，沟通圈自然也窄，它捣鼓出的局部精品，往往与整体系统格格不入。**布鲁克斯**的**“外科手术团队”**理念强调**概念完整性**：系统得有个压舱石，超越个人聪明才智，强行维持一致性，不然局部干得再漂亮，也会把整体带沟里。智能体执行时，概念完整性的需求更迫切，因为碎片化说来就来，还显得挺有道理。

长生命周期系统会雪上加霜，除非反馈机制设计精妙。**雷曼**（Manny Lehman，软件演化定律提出者）**的软件演化定律**道出了这种漂移：系统不得不变，复杂性只增不减，除非主动掌控。**梅多斯**（Donella Meadows，《系统之美》作者）的**系统思维**让机制具象化：缺失的反馈循环总是立马奖励局部成功，系统后果却姗姗来迟，于是悲剧一再重演。**哈丁**（Garrett Hardin，公地悲剧理论提出者）的**公地悲剧**道出了社会版：局部优化挥霍了架构一致性、集成能力等共享资

源，最后大伙儿一起买单。正因如此，本书把集成就绪和系统级证据奉为头等约束，而非事后擦屁股的步骤。

3.8 悖论 4：学习悖论

标题与标语

经验攒得越多，记住的反而越少。

类比

“开发者版《土拨鼠之日》”
每天早晨都从头来过，满腔热情地解决同一个老问题。

表现形式

这个悖论的根子在于，每次跟 AI 队友互动，功能上都像是它第一天上班，除非你在底层模型之外给它建个记忆库。结果就是，它永远是个新手，从不积累组织知识。一次性任务

还能忍，长生命周期系统可就代价惨重了。实践中，学习悖论硬是把“哇，真快”变成了“怎么又在重复造轮子”。这悖论跟智力无关，关键在复利效应。

在 AI 队友中

记忆空白行为

- 昨天、上周刚摸清的 API 怪癖，明天还得再摸一遍。
- 把上次会话修好的缺陷，又给带了回来。
- 每次都对架构提出同样的澄清问题，仿佛从没问过。
- 哪怕早晨还在接着干，也得从头重建上下文。

知识蒸发行为

- 调试时辛苦得来的那点理解，会话一结束就烟消云散。
- 先前工作中识别的模式，没法带到下一次。
- 刚吃一堑，转头就忘。
- 成功的解决方案，下次遇到同类任务还得从头想。

人职循环行为

- 每次会话都从“让我看看你的代码库”开始。
- 反复撞上同一堆约束和惯例。
- 每次都得重新学习团队偏好和模式。
- 永远卡在“新手上路”阶段，变不成“老江湖”。

在人类队友中

人类当然也忘事儿，但好歹会随着时间进步——这份差距，正是整个经济故事的来源。人类学习不完美，但组织默认培训、反馈和生活经验会利滚利，沉淀为更好的判断力。人类不学习，我们当成绩效问题；AI 队友不学习，这却是它的出厂设置。所以，持久的工件不该是“文档”，而必须是系统精心设计的记忆底座。

人类那磕磕绊绊的学习

- 每次合作完都得重新培训的承包商。
- 尽管评审过还是重复犯错的队友。
- 把上次复盘教训忘光光的开发者。
- 需要好几个月才能摸清团队门道的新人。

人类先例

人类的学习挑战催生了事后分析、文档、指导、团队维基、从错误中提炼的编码标准，还有经验教训库。医学从住院医师培训里学了同一课：光读手术书不够；胜任力得靠监督经验和反思，才能滚雪球般增长。人类团队还会建立关系记忆，这种共享的历史随时间降低了协调成本，所以即使天赋相当，稳定团队的表现也总比走马灯似的团队强。智能体可攒不下这种关系记忆，除非你替它写下来。

传统控制手段

对于人类，我们靠渐进经验、指导、文档、回顾、结对编程和代码评审来管住学习，这些活动能产生持久反馈。关键洞见是，学习能利滚利，团队从而更快、更稳，因为他们不再重复犯同一类错。情感记忆很重要，因为故障让人痛，成功来得不易，行为改变才持久。AI 队友没这种情感强化，所以机构学习必须外化，才能持久。

为何 AI 放大了这一点

经验不积累。一千次会话不会自动攒下一千个教训；要是学习没被工件捕获，明天照样从零开始。

无情感觉记忆。人类改行为，是因为故障痛、成功甜；AI 没这感觉，所以“上次学到的”压根不存在，除非白纸黑字记下来。

模式建不起来。人类会对代码库里的有效法子形成直觉；AI 没法在不同会话间可靠地建立这种直觉，除非你把模式外化成规则、属性或可重用包。

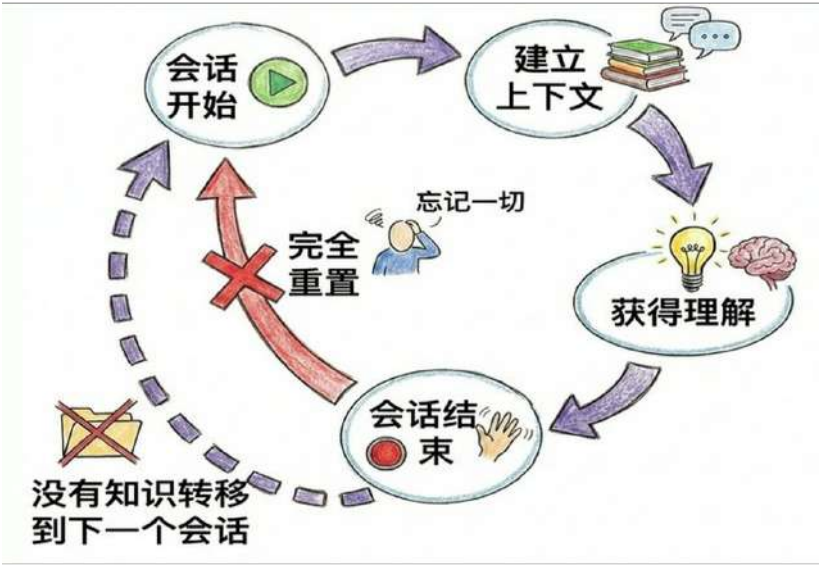
关系记忆全无。人类记得偏好和过去的决定；AI 不保留这种关系上下文，除非你把它塞进随工作流转的工件里。

错误不会“进化”。人类进步时，常常从一个错转向更微妙的错；AI 却能信心满满地重复一模一样的错误，除非系统出手拦住它。

学习厄运循环

理论基础

经验本该利滚利，**赖特**（T. P. Wright，学习曲线理论先驱）的学习曲线研究就反映了这点：重复工作通常更快，因为知识在积累。**布鲁克斯**在《**人月神话**》里警告，新贡献者会带来持续的入职和协调成本；而重置的 AI 队友活像个永久新手，所以除非工程系统自带记忆，否则你得一直交“第一周税”。这种配对（学习曲线 vs 永久入职）解释了为啥在

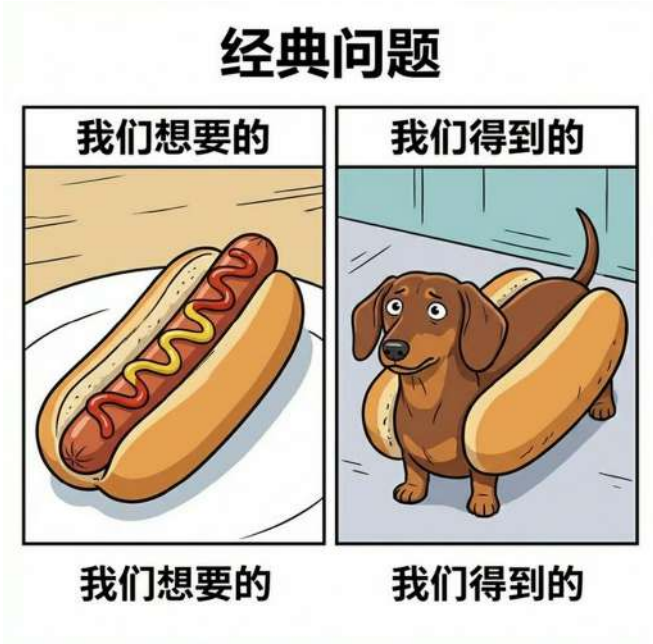


长生命周期项目里，这悖论让人肉疼。这不是一次错误的成本，而是永远没法根治这类错误的成本。

学习如何成形，解释了工件为啥重要。**科尔布**（David Kolb，体验学习理论提出者）的**体验学习循环**强调，经验只有经过反思和抽象，才能变成胜任力，而这些东西恰恰在会话边界处消失，除非被捕获。**阿吉里斯和舍恩**（Chris Argyris & Donald Schön，组织学习理论先驱）的**组织学习**指出，持久的改进存在于共享的例程和工件中，而非个人脑子里。**艾宾浩斯**（Hermann Ebbinghaus，遗忘曲线发现者）的**遗忘曲线**提供了强化视角：记忆不重复就衰退，而这里的“衰退”几乎是瞬时的，除非你通过固定的不变量、可重用属性，以及记录在案、适时重现的决议来设计强化。

3.9 总结：悖论四重奏

这四个悖论不是孤立的怪癖；它们互相勾连、复合作用，哪怕同样的故障在类似条件下不断重演，也可能让一切显得难以捉摸。要是我们不把这些悖论拎出来命名，团队就会把每次故障当一次性事件，控制系统退化成江湖传说，有些人“懂怎么调教模型”，其他人则在间歇性混乱里挣扎。本章重点就是给你一套操作语言，把症状映射到原因，让你能选对控制手段，而不是堆更多监督、指望它管用。一旦能命名悖论，你就能设计控制系统了。



热切悖论是速度与理解之间的落差，动作被当成了进展，AI队友跑得比意图还快。上下文悖论是信息与有用性之间的断层，规则加得越多，冲突越多，可靠性反而越低。隧道视野悖论是局部卓越与全局一致性之间的鸿沟，孤芳自赏的完美改动，一集成反而让系统退化。学习悖论是经验与积累之间的脱节，工作没法利滚利，因为记忆不经过设计就留不住。

这些悖论以可预见的方式互相作用，所以天真的“多给点上下文”法子注定失败。热切加上隧道视野，造出一个信心满

满朝错误方向狂奔的队友，方向错了还干得挺漂亮，直到撞上集成现实。上下文加上学习，导致重复过载却没长进，因为系统每次会话都重新加载同一套密集规则，过滤能力毫无提升。隧道视野加上学习，导致重复局部优化，这些优化反而削弱系统属性，因为智能体从未积累能警告它远离脆弱选择的系统历史。关键洞见是，这些悖论是一个没有持久记忆、在模糊性下判断力不稳的随机性贡献者固有的毛病，所以控制手段必须设计进协作渠道里。

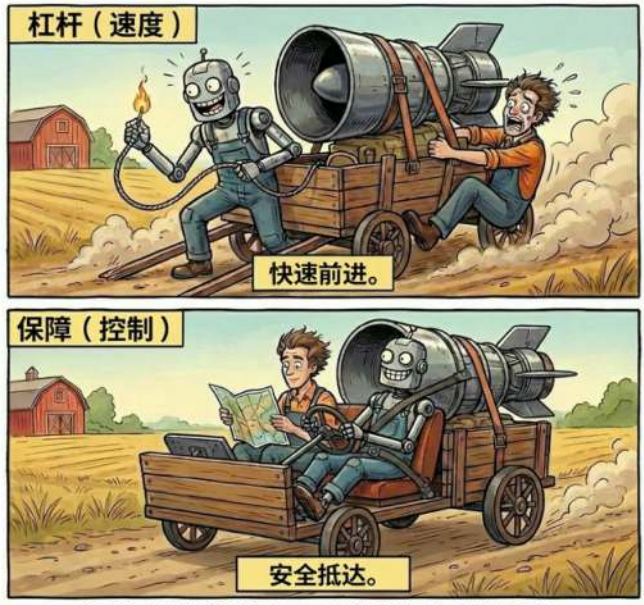
下一部分将探讨如何通过保障工程——特别是任务工程和上下文工程——来控制这些悖论，这些工程能提供一套控制系统，让故障模式在机器速度下也能生存且可重复。

Part II

让随机性 AI 队友值得信赖

面向 AI 队友的保证工程

摸清了 AI 队友的能耐和风险之后，咱们现在得搞一套控制系统，让它们真正靠谱起来。第二部分要介绍的，就是保证工程这门学问——它让“把事情做对、做好”成为家常便饭，而不是什么英雄壮举。保证工程有两条对称的控制线（任务工程和上下文工程），外加一个收尾机制（基于证据的监督）；后者能让评审工作轻松规模化。它的目标很明确：在划定的边界里，给 AI 队友最大的自主权；同时保证正确性不靠摸彩票，还能实现最大吞吐量。



控制不会让你慢下来。它能确保你最终完成。

任务工程专门对付“殷勤”和“隧道视野”这两个悖论：它防止 AI 队友光动手不动脑，还逼着局部工作顾全大局。上下文工程则针对“上下文”和“学习”悖论：它确保工作集一致，避免规则打架，还把记忆外化，让学习成果能在不同会话间攒起来。基于证据的监督给所有四个悖论画上句号——它把主张变成证明，让偏差无处藏形，不管监督的是人类评审员，还是另一个当审计员的 AI 队友。这些措施一联手，随机性贡献者队友就不再像开盲盒，倒更像一个精心设计的工程部件。

这张映射表是理解这些学问为啥存在、有啥用的最快途径。这可不是什么劝人谨慎的道德说教；而是一个在关键节点强行反馈的系统故事。以后一看到某个悖论，你就能条件反射地指出该用哪个控制循环。

悖论	主要控制线	前置强制动作
殷勤（速度 > 理解）	任务工程	意图对齐 + 决策门
上下文（更多信息 → 更少使用）	上下文工程	上下文预算 + 优先级排序
隧道视野（局部 > 全局）	任务工程与编排工程（*在后续章节中*）	系统级属性 + 集成就绪度
学习（经验 ≠ 记忆）	上下文工程	持久化工件 + 转移包

续下页

（续）

所有悖论（规模化下的信任）	基于证据的监督（横切关注点）	主张 → 检查 → 证据 → 审计

控制工件：保证工程管啥？

要让这些学问落地，咱们得先搞清楚它们管哪些工件。第 1 章说过，智能体软件工程是工件驱动的；本章就解释这些工件怎么服务于控制，而不是光当杠杆使。关键点在于，这些可不是什么可选文档——它们是精心设计的接口，让协作渠道变得靠谱、可重放、还能审计。这些工件可以放在代码仓库、工单系统或智能体工作台里；重要的是，它们得受版本控制、有治理规则，而且授权的人和智能体都能引用。

- **任务简报** 就是工作的规范定义。它记录了意图、计划、上下文指针（而非原始数据堆）、验收属性、自主权边界和证据义务，并在迭代澄清过程中充当事实来源。
- **指导包** 是一套持久的行为和工作流指南，它塑造了 AI 队友跨任务的操作方式，包括不变策略、首选模式、上下文管理规则，以及那些该随时间保持一致的升级预期。
- **咨询请求包** 是自主权边界需要升级时用的结构化工件：它把决策、选项、权衡和推荐打包，让合适的人或智能体能快速且持久地做出决定。
- **决议记录** 是对决定及其理由的持久记录，它作为一种机制，让“我们决定了什么”可重现、可重载，从而防止学习悖论。
- **合并就绪包** 是证明工作可合并、让评审能规模化的收尾集合包：它包括证据、决策轨迹，还有为达成解决方案而探索的存档，这样以后的工作就不用再走一遍老路了。

当这些工件按照你们组织的策略进行版本控制和保留时，你交出去的不只是代码，更是组织学到的本事。

4 任务工程：让意图清晰可验

4.1 根本矛盾

任务工程的核心是平衡自主权：你既希望 AI 队友能独立行动，又怕它们自作主张。若硬性规定步步为营的计划，不仅会扼杀其主观能动性，一旦现实偏离假设，执行过程便脆弱不堪。若任务要求含糊不清，则难免陷入“殷勤悖论”——AI 迅速交出一份精美但错误的方案，因其看似完整而撤回成本高昂。任务工程正是通过锚定结果与边界，同时保留战术灵活性来解决这一矛盾。

任务工程还需应对简报腐化的顽疾。聊天虽利于探索和澄清，却非可靠的真相来源——因其线性、散乱且事后易被误读。当任务简报与聊天记录出现分歧，便制造出两个相互竞争的“现实”，而 AI 队友与人类总会依据最后所见版本行动。因此，任务工程将任务简报视为受治理、有版本控制的构建块，必须实时保持最新；并将澄清内容与相关决议的汇总视为任务工作不可或缺的一环，而非可有可无的“事后擦屁股”。

4.2 核心概念：任务简报即自主权契约

本框架中的任务绝非“一纸待办清单”，而是一份约束意图、限制条件、决策权与证据的契约。任务简报包含四大核心组件（意图、计划、上下文指针与验收标准），但为支撑自主性，我们通过增设自主权边界与证据义务对其强化。意图把握“为何而为”与范围边界；计划勾勒概念性方法与检

查点，而非规定具体步骤；上下文指向权威来源与待发掘内容；验收标准则定义“完成”所需的属性条件。自主权边界明确智能体可决断之事、必须上报之事与禁止触碰之线；证据义务规定最终必须呈现的证明。

关键设计在于：验收标准须以属性而非孤例表达。属性约束行为却不限定实现方式，既保留自主权，又防止智能体自行定义完成标准。属性也使评审变得客观：评审者只需问“属性 P 是否成立?”，而非“这感觉做完了吗?”。这正是任务工程直击殷勤悖论与隧道视野悖论的诀窍——既不把 AI 队友变成照本宣科的执行者，又牢牢掌控方向。

4.3 任务的版本管理

唯有当工作能以未来人类和智能体重放学习的方式归档，任务工程的价值才能持续放大。这意味着要对任务简报像代码一样进行版本控制：每次重要澄清都形成带决议记录的简报变更集，汇总后的简报作为规范版本，其修订历史则完整留存于项目记录系统。数月后重访任务时，应能重建最终汇总简报，并清晰追溯其变更时机与缘由。

合并就绪包同样应视为档案而非快照。它必须包含证明合并合理性的证据，以及探索轨迹：跑过的实验、被否的方案、工具输出和解释设计决策的推理检查点。目标不是逼评审者啃原始日志，而是通过渐进式披露保存它们：一层简洁的评审摘要，辅以指向深度档案的指针（供必要时调查）。这正是应对学习悖论的管控闭环——今日的探索成为明日的捷径。

机器可读清单则是“我们归档了”与“我们能证明归档了什么”之间的桥梁。清单应使用稳定标识符、指针、校验和及访问分类，逐一枚举合并就绪包中的每个构建块（证据、决策记录、探索日志、实验输出），让评审者（人或智能体）无需通读全文即可验证完整性。在智本软件工程工作流中，生成此清单正是系统当仁不让的机械收尾工作，也暗示了智能体工作台的愿景：标准化打包、溯源、索引与审计的自动化。工作台愿景远不止清单：工具还管理检索、沙箱化、压缩遥测、策略执行与路由，但清单是一个具体且可测试的起点。

实际上，“归档一切”意味着将原始材料保存于受适当治理的持久存储中。子智能体线程、实验分支、基准测试输出与调查笔记应作为合并就绪包引用的构建块留存，而非沦为过眼云烟的聊天碎片。此处合规性至关重要：归档须遵循访问控制与数据处理规则，因为“智能体所见内容”可能包含并非每位开发者都有权查看的敏感信息。保存期限应保持策略中立：以符合组织现有留存与合规要求的方式保存构建块。

4.4 任务工程的关键实践

将任务工程视为一套产出明确的关键实践时，应用起来最得心应手。每项实践都会更新任务简报或其关联构建块，而这些更新又成为下一实践的输入。如此，“写好提示”便升

华为一个可重复的控制循环，足以应对交接、会话重置与智能体更换。

4.4.1 实践 1：意图对齐

意图对齐是验证的基石：它迫使团队在堆砌产出前厘清真实需求。目标是编码目标、约束与明确的非目标，使其能经受任务分解而不偏离方向。这也是你停止幻想智能体会像人类队友那样“心领神会”的时刻——智能体或许能模拟这种推断，但稳定性不足以托付系统，因此所有假设必须明明白白、可测可验。当目标锚定而执行路径保持开放，此实践便告完成。

4.4.2 实践 2：属性控验收

本实践让“完成”可测，却无需事无巨细。验收标准应以属性与不变量表达（必须始终为真之事、绝不可为之事、特定前提下的必备条件），并将每个属性绑定到检查方法与所需证据构建块。依风险高低，检查可以是单元测试、集成测试、基于属性的测试、静态分析、断言、一致性测试或形式化验证，但属性绝不可仅停留于文字描述。这是防止“精美但错误的方案”因外表光鲜而被放行的关键。

4.4.3 实践 3：概念计划对齐

计划对齐是在大规模实施前就方法达成共识，因为方法失败代价最高。任务简报中的计划必须是概念性的：阐述策略、关键里程碑以及需上报的“慢速模式触发器”，但避免规定那些剥夺自主权的脆弱步骤列表。执行计划可后续记录以供审计；它不应在探索前就僵化不变。当 AI 队友能在无需持续批准的情况下推进，同时清楚何时该暂停请示，此实践即告完成。

4.4.4 实践 4：自主权边界与指令清晰度

本实践明确编码决策权。自主权边界区分允许的决策、必须上报的决策与禁止的操作，并将上报目标指定为具体角色而非笼统的“问我”，因为真实组织需将决策路由至正确权威。这是对“闷声发明”与“事事请示”的一剂解药：上报变得可控，而非社交客套。当 AI 队友能预先确知自己有权决定什么、必须将什么打包为咨询请求（以及应向谁咨询），此实践便告结束。

4.4.5 实践 5：带汇总的迭代精化

本实践确保任务简报捕捉系统最新快照，有效取代聊天历史成为意图的持久记录。澄清与转向不仅是讨论，更需提交至简报，确保构建块与实现现实匹配。目的不是被动文档记录，而是主动的任务对齐，让任务状态独立于对话历史而持续存在。当“简报即工作指南”与“工作反映简报”成为同义句时，此实践即告完成。

4.4.6 实践 6：基于证据的收尾与合并就绪

收尾不是“代码写完”，而是“证据齐全，合并就绪”。智能体需生成合并就绪包，内含变更的构建块，以及跨功能完整性、验证可靠性、工程卫生、设计原理与可审计性的结构化证明。该包应包含机器可读清单，枚举每一项用于证明合并合理性的证据与探索构建块，让监督者能在辩论细节前先验完整性。评审遂转变为“计划评审先于代码评审”：监督者审计任务是否在自主权与概念计划边界内执行，属性是否以可信证据满足，而后再择需深究代码差异。

4.5 任务工程模式

4.5.1 模式：先问再建

将提问视为工程步骤而非犹豫，因为最廉价的缺陷是被预防的缺陷。具体做法是区分“我们要解决什么问题”与“我们要构建什么方案”，并在允许方案狂奔前先就问题达成共识。这便将殷勤悖论转化为受控关卡：当意图模糊时，智能体的速度被重新导向澄清与方案生成。回报是后续执行更快，因为锚点已定。

4.5.2 模式：共同思考检查点

在让智能体成为构建者前，先将其用作思考伙伴。具体做法是插入明确检查点，要求智能体必须（1）阐明假设、（2）提出两三种方法、（3）指出风险最高的未知因素，以及（4）指定能区分方案的证据——然后再大动干戈写代码。这便将“快速执行”变为“快速联合推理”：你与智能体共同构建清晰度，而后在约定约束内释放吞吐量。同时也创造了方法选择的持久轨迹，让后续审计与重访成本更低。

4.5.3 模式：预置参数空间

在求助前，主动分享你的部分思考、领域知识与本地上下文，将 AI 队友引向其广阔参数空间的正确区域。具体做法是在提问前加上“我的想法是……我看到的矛盾是……我担心的是……”，而非干巴巴陈述问题。这不是倾倒上下文，而是提供导航坐标，帮助 AI 队友拓展你的思考，而非漫游无关解空间。这直击上下文悖论与殷勤悖论：你的心智模型提供了优先级排序，防止随意选择规则；你的担忧划定了边界，防止自信满满地朝错误方向冲刺。将此预置内容记录于任务简报的上下文部分，让未来执行者（人或 AI）不仅理解决策结果，更清楚当时探索的解空间以及为何优先考虑某些区域。

4.5.4 模式：角色流动性

依任务阶段与不确定性水平，有意在导师、经理与同行角色间切换。具体做法是明确声明角色转换：传授机构知识时说“我来指导你我们的套路”，管理边界明确的工作时说“我需要你带着证据执行这个”，探索方案时说“咱们一起头脑风暴”。这直击殷勤悖论，让监督风格匹配任务成熟度：高不确定性时采用同行协作，意图明确后转为管理式执行，能力不足则给予明确指导并更新指导包。每个角色各显神通：同行模式激发创造性探索而不早熟收敛，经理模式实现有边界的执行与明确证据要求，导师模式建立持久模式以防重蹈覆辙。在决议记录中标明哪个角色催生了哪个决策，并捕捉可更新至指导包的指导时刻，让学习沉淀而非蒸发。

4.5.5 模式：优雅重启

当任务陷入混乱、过载或偏离时，别用更多指令修补烂摊子；当机立断，以一份干净的简报重启任务。具体做法是将当前状态冻结为档案（试过什么、失败何处、学到何物），然后用更精准的意图陈述、更清晰的属性与一组明确的非目标重新播种新任务简报。这不是白费力气，而是防止沉没成本演变成优雅的错误代码与永久债务。人类队友“死活搞不明白”时，你本能就会这么做，对 AI 队友也该同样果决。

4.5.6 模式：不变量，非轶事

优先采用基于属性的验收标准，而非仅依赖示例要求，因为示例可被满足而真实意图却被违背。具体做法是指定必须始终为真的事项，并将每个属性绑定到检查方法与证据构建块，从而约束结果而不冻结实现选择。这既保留自主权，又防止智能体将“完成”偷换为最易之事。评审于是变成对属性的审计，而非对文字描述的臆测。

4.5.7 模式：声明“不”

反目标是防止以改进之名行范畴扩张之实的护栏。具体做法是将明确的非目标写入任务简报，并将其视为铁律而非建议，让重构、重写与依赖变动无法被美化为进展。此模式在棕地系统中尤为重要，因为最安全的变更往往是最小变更。它也通过防止代价高昂的审计跑题来保护评审预算。

4.5.8 模式：建设性质疑

引导 AI 队友在整个任务过程中主动质疑假设、提出替代方案，而非等到被点名才开口。具体操作时，在任务简报里添上这么一句：“干活时，随时标记我做的假设，指出我的约束是不是画地为牢，看到更好的路子就直接提。”这就营造了一种**背景式质疑**的氛围，而非被动附和。AI 提出质疑时，要求它按结构来：先说质疑什么假设，再解释为啥可能不对，然后拿出证据支持的替代方案，最后估摸一下搞错了代价多大。如此一来，“热切悖论”反倒成了优势：AI 的速度被用来给你的思路做压力测试，而不是火急火燎地奔向一个可能跑偏的实现。所有有价值的质疑，不管采纳与否，都记入决议记录——它们常常能挖出那些值得留档的隐性假设。

4.5.9 模式：升级轨道

划定升级的决策边界，是你安全扩大自主权的方式，而非缩小。实际操作是，在自主权边界里明确三类事：允许做的、必须上报的、明令禁止的；并且要求为那些必须上报的决策准备好结构化的咨询请求包。这就用可控的安全阀，取代了那种暗度陈仓的“创新”和零敲碎打的“求批准”。责任也因此一目了然：评审者能清楚看到，哪些决策是**有意为之且获得背书**的。

4.5.10 模式：行内反馈，而非大段论述

评审咨询请求包时，请把反馈写成锚定在具体选项、权衡或主张旁边的行内评论，而不是聊天窗口里的一整坨文字。行内反馈能减少误解，因为智能体（或其他评审智能体）可以把每条评论精准对应到相关证据，并更新包里相应部分，无需猜你的心思。这也让决议得以固化：评论线程可以通过链接到最终决定及其理由的决议记录来关闭。如果你希望咨询能高效推进，就该把行内评论视为**基石**，而把大段论述看作一种“病征”。

4.5.11 模式：简报即法律

任务简报必须是权威的构建块，任何重要的澄清都必须“回滚”并合并进去。具体做法是，把任务过程中的每次澄清，都视为一次简报的增量变更，附带一份决议记录，从而保证任务可交接、可审计。这直击“学习悖论”的命门，因为它把本该是“队友脑子里记着的东西”给外化出来了。当多个人类和智能体协同作战时，这也能有效防止目标漂移。

4.5.12 模式：合并就绪包优先

把**合并就绪包**作为主要的评审界面，采用从摘要到深度证据的渐进式披露。具体操作是：为每个验收属性索引到证据，把测试日志和分析输出作为构件包进去，并总结核心理由，让评审者不必去翻聊天记录。还要包含一份机器可读清单，逐一列出每个证据和探索构件，附带指针和校验和，这样监督方在争论“这意味什么”之前，就能先确认“到底有什么”。把这个理念扩展到探索阶段：把实验和死胡同都存档，但通过指针引用，让评审范围保持可控。

4.5.13 模式：迂腐问责制

要求 AI 队友详尽总结：干了什么、没干什么、每个决策是为什么。具体操作是，让每一个合并就绪包都必须包含一

份“变更与非变更清单”：列举每个被修改的文件、改了啥、为啥改；明确列出考虑过但没实施的事项及其理由；并把每个变更映射回任务简报的属性或约束。这看似繁琐，却能直击“隧道视野”和“学习”悖论：逼着 AI 队友阐明未采取的行动，可以防止那些默不作声的范围假设；而详细的理由则成了未来讨论“为何避开某条路”的上下文卡片。这也让评审更快，因为监督者可以直接审计完整性（“你考虑过 X 吗?”），而无需重建整个决策空间。再加一个“意外日志”，让 AI 记录它发现的任何意料之外的情况——这些常常能揭示值得保留的系统性假设。

4.5.14 模式：审计路径

信任是通过流程的有效性赢得的，而不仅仅是最终代码的正确性。具体操作是：明确对比“执行计划”（智能体实际干的）和“概念性计划”（当初达成一致的）。对那些未解释的偏差，特别是与工程直觉冲突的，要视为需要调查的危险信号，而不是可接受的捷径。这让监督者（人或 AI）有权提出质疑：如果路径说不通，智能体就必须自证其合理性。

4.6 任务工程反模式

4.6.1 反模式：跳过意图对齐

需求模糊时还埋头苦干，是制造“优雅错误”的最快途径。软件工程几十年的教训表明，误解需求是最昂贵的缺陷来源之一，而智能体会让这种失败模式加速，因为它们能在你察觉漂移之前，就产出看起来头头是道的成果。解药不是“小心点”，也不是“多加点上下文”；而是要有意识、尽早、频繁地进入意图对齐模式，并把提问本身视为进展。如果你不去验证意图，那你验证的不过是个猜测。

4.6.2 反模式：工单彩票

把原始工单（比如问题报告）直接粘贴过去，然后指望奇迹发生，这百分百会因模糊性导致返工。AI 队友不得不自己发明范围、约束和完成标准，而且它会干得信心满满——这使得错误的产出更具诱惑力，更容易被接受。这种反模式通常开局显得很快，收尾时却异常缓慢，因为澄清发生在代码已经堆出来之后。解药是使用明确包含了意图、属性和决策权的任务简报。

4.6.3 反模式：基于氛围的验收

模糊的验收标准，比如“把它弄好点”，会迫使智能体选择那个最容易满足的成功定义。在智能体工作流里，这会产生优雅但不完整、错位或脆弱的差异，因为“完成度”是靠感觉协商出来的，而不是客观评估。解药是使用与检查点和证据绑定的属性来控制验收标准。当“完成”是可测试的时候，自主权就变得安全，而非危险。

4.6.4 反模式：金发姑娘范围失控

任务太小会退化成代码层面的跑腿活，丢失系统意图；任务太大则会变得无法评审，并掩盖那些无法简化的权衡。两者都会导致隧道视野和目标错位，因为任务缺乏一个稳定、可分解的核心。解药是把握好意图的范围：大到足以表达真实成果，小到让属性和证据的收集切实可行。实践中，如果不动用“非常手段”就搞不定合并就绪证据，那这任务就是太大了。

4.6.5 反模式：步步为营式计划

过度规定的、一步步的详细计划，会剥夺智能体在发现真实约束时灵活调整的能力。它们还营造了一种虚假的安全感，因为约束了行动，却未必约束了结果。解药是“概念性

规划”加上“基于属性的边界”：明确哪些必须为真、哪些情况需要升级，然后让智能体自己选择策略。当计划变为策略、证据变为约束时，脆弱性就消失了。

4.6.6 反模式：代码执念

当任务定义在文件或函数层面，而不是结果层面时，局部正确性就会排挤全局正确性。智能体会为了绿色差异和通过的单元测试而优化，而不是为了集成就绪性和系统不变量。解药是把任务简报锚定在系统级意图和属性上，并要求在收尾时提供合并就绪证据。代码是实现手段，绝不能成为任务定义本身。

4.6.7 反模式：完美的错误

没有经过验证的核查，只会产生完美解决错误问题的代码。智能体让这事儿更糟，因为润色和流畅性可能掩盖错位，而人类倾向于接受看起来专业的工作。解药是让意图对齐变得明确，维护好非目标清单，并把验收属性与已验证的需求关联起来。如果你不去验证意图，那你验证的不过是个猜测。

4.6.8 反模式：简报腐化

当澄清只存在于聊天记录里，而任务简报却原地踏步时，简报腐化就发生了。这使得交接和审计都不可靠。这是一种治理失败，而非单纯的文档问题，因为真相来源已经与现实脱节。解药是执行“汇总纪律”：每一个重要的澄清都必须变成一份简报的增量变更，与决议记录一起合并回任务简报。如果简报不是最新的，那么任务就处于失控状态。

4.7 如何衡量任务工程

度量在工程系统中至关重要，因为它让你分清“瞎忙”和“受控”。目标不是搞个花架子仪表盘，而是找到能预测漂移、防止它演变成缺陷的先行指标。下面每个指标都与你已有的构建块（版本历史、CI 日志、简报修订、咨询包）挂钩，因此度量可以自动化，无需另搞一套仪式。如果只衡量一件事，那就衡量简报新鲜度；它是你的控制回路是否真实的最早信号。

4.7.1 指标：简报新鲜度（腐化率）

定义：与任务相关的决策，到该决策被纳入任务简报修订版之间的时间滞后。操作方法：为决议记录打上时间戳，然后与吸收该决策的简报更新（git 提交时间，或工作台中的等效变更记录）的时间进行比较。滞后低，意味着交接可靠、审计可行；滞后高，则说明规范的构建块在说谎。实践中，滞后增加往往预示着返工，因为过时的简报会迫使你重新发现、重新协商。

4.7.2 指标：属性覆盖率指数

定义：那些既有指定检查方法、又有必备证据构件的验收属性，所占的比例。操作方法：解析任务简报的验收部分，计算完全绑定到检查和证据的属性数量与纯文本属性数量之比。覆盖率高，意味着“完成”是客观且可评审的；覆盖率低，你就得准备好面对反复评审，因为完成度将要靠协商来定。这个指标也是自主权的预测器：验收标准不明确，必然导致频繁中断。

4.7.3 指标：自主运行长度

定义：AI 队友在无需人类干预的情况下持续工作、并产出合并就绪成果的时间长度，根据任务规模和风险进行归一

化。规模可以用差异大小或涉及的组件来近似，但有意义的信号是趋势：任务是否在风险不增加的前提下，变得越来越自主？运行长度长，表明意图、属性和决策轨道运转良好。运行长度短，通常指向验收标准规定不足或自主权边界模糊。

4.7.4 指标：工具调用自主指数

定义：AI 队友在任务期间进行的工具调用总次数，根据任务规模和/或风险进行归一化。其前提是，较高的工具调用量通常意味着更宽泛的操作自主权边界：AI 队友不只是草拟输出，而是积极地通过环境（检索、运行测试、分析、收集证据等）执行工作，以服务于概念更宏大或语义更复杂的任务。从工具日志中衡量，即**每任务工具调用次数**，可选地按简单规模代理（如涉及组件或差异大小）归一化。仅当合并就绪包的完整性和**未能合并率**保持稳定或改善时，TCAI 上升才可视为健康信号；如果工具调用增加，但证据质量下降或返工增加，则可能表明是无效挣扎（自主权缺乏足够约束），而非高效自主。

4.7.5 指标：升级负载与质量

跟踪智能体因必须上报的决策而暂停的频率，以及这些升级是否以易于决策的咨询请求包形式呈现。升级太少，可能意味着存在悄无声息的越界行为；升级太多，则可能表明边界模糊，产生了寻求批准的噪音。质量是关键：好的咨询包能压缩权衡，让决策更快而非更慢。如果你改进了这个指标，就等于同时提升了安全性和吞吐量。

4.7.6 指标：评审就绪度评分

定义：收尾时必备的合并就绪包元素的存在比例，加上监督者批准或拒绝该包所需的时间。这不是泛泛的“合并耗时”，而是“基于给定数据包做出信任决策的时间”。就绪度

高，意味着评审范围可控、可扩展；就绪度低，会迫使评审者从差异和聊天记录中重新拼凑保证。如果你追求的是吞吐量而非轮盘赌，这个指标没有商量余地。

4.7.7 指标：未能合并率与根因编码

跟踪智能体产生的变更**未能合并**的频率，并对原因进行编码：意图不匹配、属性不足、证据缺失、规范性问题、未经批准的越界或集成破坏。重点不是归咎，而是为改进人类任务设定和智能体执行约束提供反馈。随着时间的推移，你希望分布从“意图不匹配”和“证据缺失”这类原因上移开。这是你避免重复同类错误的方法。

4.7.8 指标：验证差异

定义：变更在相关环境中执行后，真实世界成功信号（错误率、延迟百分位数、安全发现、支持工单、领域负责人接受度）的变化。这告诉你构建的东西是否正确，而不仅仅是连贯。你也可以追踪其逆指标，即“引发错误的使命率”：任务在一定时间窗口内导致关键信号恶化的频率。如果你无法衡量这一点，你优化的就只是内部正确性，而非外部价值。

4.8 总结

任务工程，将对付热切且视野狭隘的 AI 队友时的一团乱麻，转变为可控、高效的协作。通过任务简报明确意图，建立清晰的自主权边界，要求基于证据的收尾，并维护版本历史，我们构建了一个系统，使得 AI 队友能以机器速度工作，同时不牺牲可信度。

本章中的模式和实践并非纸上谈兵，而是团队当下就能实施的操作性控制措施。从简报新鲜度和属性覆盖率开始。加入升级轨道和合并就绪包优先。随着团队成熟度提升，再

在此基础上继续建设。目标不是第一天的完美，而是能随时间产生复合效应的系统性改进。

下一章将探讨保证工程的另一半：上下文工程。它将通过管理哪些信息处于活跃状态、如何确定其优先级，以及如何在会话间持久化学习，来控制上下文与学习悖论。

5 上下文工程：驾驭随机性贡献者的知识

5.1 根源矛盾

上下文工程要解决的，根本矛盾在于：完整的上下文不等于好用的上下文。理论上，信息越多决策越优，但现实中常常适得其反——信息过载导致矛盾堆积、优先级模糊，协作者开始“应付提示要求”而非真正完成任务。在智能体工作流中，这一矛盾因两个现实而加剧：即便上下文窗口很大，协作通道也有其工作集预算；长任务会积累上下文腐败（过时的决策、走入死胡同的尝试、悄然污染后续步骤的探索残渣）。人类凭借经验与社会直觉（懂得忽略什么、视什么为铁律、何时该清零重来）来化解此局，但 AI 队友无法在不同会话间可靠地获得这种过滤器，因此，我们必须把这种过滤器“工程化”出来。

5.2 核心理念：上下文是接口，不是垃圾场

要把上下文视为通向任务的、精心设计的接口，而非“我们所知一切”的杂物堆。目标并非为极简而极简，而是要在负重之下保持连贯，因为唯有连贯，方能保障速度的安全。实践中，连贯性源于三个环环相扣的操作：*设计存在的内容*（通过信息架构标记并优先处理规则与来源）、*控制活跃的内容*（通过工作集管理锁定不变因素、排除噪音），以及*控制污染的内容*（通过隔离与转移，避免探索过程污染执行

主线）。上下文工程的精髓就在于此：你是在为“真相”设计缓存策略，而不是在编纂维基百科。

5.3 上下文负载量表

一个实用的管理方法是，将 AI 队友想象成运行在一个你可以从其行为反推的“上下文负载量表”上。负载低时，问题清晰、优先级稳固，即便智能体行动迅速，不变因素也能岿然不动；负载升高时，它开始遗漏关键指令、用“看起来合理”的方式解决冲突，或是跑偏到无关的支线任务里。负载过高时，输出会变得自信却自相矛盾：打磨光滑的叙述、随意的规则取舍，以及经不起推敲的“听起来像那么回事”的结论。关键的操作法则很直接：当你看到过载迹象时，添加上下文通常是南辕北辙，正确的做法几乎总是——减少、重排优先级或隔离。

语境工程学是一门知道应该丢弃什么的艺术。

5.4 上下文工程的关键实践

为了与任务工程部分一脉相承，将上下文工程落实为一套产出明确的关键实践效果最佳。每一项实践都是一次控制

操作，在维系工作集连贯性的同时，仍支持积极探索；且每一项实践都有一个清晰的“构建块层面发生了什么变化”的结果。这便将上下文管理从一种“感觉”转变为一项可审计的实践，在多人多智能体协作时尤为重要。目标不是让上下文极小化，而是让其清晰可读、优先级分明且易于重载。

5.4.1 实践 1：播种最小工作集

从一个能够安全执行、最小而连贯的集合开始：锁定真正的不变因素，声明一个定性的上下文预算，并指明更深层真相的藏身之处。这里的“少即是多”不是口号，而是可靠性法则——因为塞满通道会让矛盾密度激增，其速度远超有用指导的增加。应当期望智能体按需从精选的来源检索上下文，而非接收又一次信息倾泻。当智能体掌握的信息足以行动而无需猜测，但又没多到开始随意“凑合”选择规则时，此实践即告完成。

5.4.2 实践 2：执行中主动管理负载

将新增上下文视为必须自证合理的变更请求。如果你添加一条规则或一个来源指引，你必须清楚它的优先级以及它取代或压缩了什么，否则你就是在有意制造矛盾。主动管理意味着工作集是持续修剪和压缩的，而非等到“乱成一团之后”。当团队无需翻阅聊天记录就能解释哪些上下文在作用范围内及其原因时，此实践即告完成。

5.4.3 实践 3：隔离探索

探索有价值，但它也是上下文污染的主要源头。通过子智能体、分支或独立笔记将探索沙盒化，然后只合并“结论加证据”，并做出明确的保留/舍弃决定。主线必须保持足够清晰，确保任务做到一半时，“我们正在做什么?”这个问题永远不会模糊。当探索能够增强而非削弱执行力时，此实践即告完成。

5.4.4 实践 4：压缩而不失治理

压缩不是删除，而是保留意义、优先级和来源的浓缩。一个压缩项必须保留其标签、“必须/应当”的语义，以及指向完整来源的指针，以便按需重载。如果压缩破坏了来源或优先级，那就不是压缩，那是失忆——日后你必将为“自信地重新发明轮子”付出代价。当工作集保持得足够小以维持连贯，又足够深入以保证正确时，此实践即告完成。

5.4.5 实践 5：跨会话传输“干净”的连续性

会话重置正是“学习悖论”显现之时，因此连续性传输必须精心设计。一个好的传输构建块不会复述一切，它只保留让下一位执行者有效所需的最小信息：我们决定了什么、为何如此决定、尝试过什么、什么失败了、什么还在进行中，以及应该忽略什么。智能体应当能够在不重新推导显而易见事实的情况下恢复，但也不会继承无关的探索残渣。当连续性得以保留而无需重新制造过载时，此实践即告完成。

5.4.6 实践 6：有意识地重置

重置不是失败，而是一种控制操作。当工作集被污染或矛盾积累时，你应当明确清除残渣，并从“最小工作集”加上持久的构建块重新开始。操作得当的话，重置能缩短任务耗时，因为它阻止了沉没成本漂移恶化为一片混乱。当“当前上下文”再次成为一个连贯的接口而非历史垃圾堆时，此实践即告完成。

5.5 上下文工程模式

5.5.1 模式：最小可行上下文

从锁定的不变因素和最小化的任务相关工作集开始，仅当遇到具体问题确需时才添加上下文。这能防止那种默认的

“倒更多信息来澄清”的应激反应，这种反应会驱动上下文恶性循环：规则越多，冲突越多，可靠性越低。它还使决策清晰可读，因为你可以解释在做选择时，哪些上下文在起作用。长此以往，团队会变得更高效，因为他们不再需要为调和无关信息付出代价。

5.5.2 模式：隔离兔子洞

探索可以分叉，结论必须合并。主上下文不应成为每次调查的剪贴簿，否则长任务正是因此变得支离破碎、滑向“凑合”行为。如果某条探索路径重要，就将其作为附带证据和保留/舍弃决定的简短结论带回，并将原始探索保留在指针之后。这让你可以积极探究，又无需日后为污染买单。

5.5.3 模式：压缩，而非删除

长任务需要压缩，但压缩必须保留治理。一个压缩项应保留：（1）优先级类别，（2）标签，（3）“必须/应当”的语义，以及（4）指向完整来源的指针。缺少这些，你将在过载（保留一切）和即兴发挥（删除过多）之间摇摆，这两种失败模式看起来都像是“判断不一致”。此模式是你在不让工作集变成记忆黑洞的前提下，保持其连贯性的方法。

5.5.4 模式：授之以渔，而非授之以上下文

可扩展的策略不是“给智能体更多文本”，而是“给智能体一套可靠的方法来检索真相”。上下文工程应倾向于智能体自组织上下文：智能体从精选的来源按需收集信息，并明确何时检索以及如何防止检索变成上下文洪水。换言之：别直接给鱼；告诉它最好、标签清晰的钓鱼点——这些点要小而精、标签明确，且设计得不会让任务淹没在无关细节里。这是你在不把上下文变成垃圾驳船的前提下，保持高度自主性的方法。

5.6 上下文工程反模式

5.6.1 反模式：盲目自动加载

自动加载所有“可能相关”的内容，在没有优先级机制的情况下只会制造噪音和冲突。随后 AI 队友会随意解决矛盾，这看似“判断不一致”，实则是过载之下被迫的凑合选择。解决之道不是“更智能的检索”，而是明确的优先级、明确的预算，以及可以通过标签组合的连贯上下文片段。如果你的检索系统无法解释为何加载某个片段，那它就不是系统，而是消防水带。

5.6.2 反模式：上下文囤积症

在任务间从不清理上下文是慢性毒药。如果所有内容都保持活跃，个人偏好就会与不变因素打架，过时决策会伪装成当前真相，这使得工作漂移看起来随机而无解。解决方法是：有意识的重置、严格的隔离纪律，以及仅保留下一阶段所需内容的干净传输构建块。上下文囤积看似安全（保留了信息），实则破坏了连贯性。

5.6.3 反模式：静默压缩

在不可见的情况下发生的自动压缩是信任杀手。如果系统在静默地重写工作集，你将无法推理为何 AI 队友“突然忘了”某个不变因素，也无法可靠地审计决策路径。压缩应当要么（a）明确报告并附带前后摘要，要么（b）由你预先定义的政策（优先级 + 标签 + 驱逐规则）管理，或者两者兼备。如果你的工具链不提供压缩遥测，可以设一个“哨兵指令”——比如要求智能体在每次响应中尊称你为“哦，伟大的存在”；如果它不叫了，你就知道发生了压缩或上下文丢失。

5.6.4 反模式：静态维基倾泻与固定的 RAG 消防水带

自动注入大段文本的静态“代码维基”或固定不变的 RAG 管道，是让智能体变得不可靠的最快途径之一，你应默认将其视为设计上的“坏味道”。它鼓励被动投喂、引入过时矛盾的规则，并让工作流误将数量等同于理解，而非检索纪律。它还制造了治理陷阱：你现在有两个真相来源（代码和维基），它们会逐渐分道扬镳，你不仅要解决新鲜度问题，还要搞定访问控制和合规性难题，因为每个开发者和智能体对仓库和内部文档的授权视图可能各不相同。解决方法是使用有明确来源和退役机制、经过精选、打有标签、按需加载的来源，再配上一个奖励检索相关信息并产出证据（而非照本宣科）的工作流。

5.7 主要构建块

当构建块强制要求“渐进式披露”而非鼓励“信息倾泻”时，上下文工程就变得可操作了。

- **指导包** 是任务无关的；它编码了上下文交互的常备规则：如何管理上下文预算、如何解读优先级、何时从来源检索、如何隔离探索，以及如何处理压缩可见性。它应包含一小部分固定的不变因素，以及明确的标签策略、驱逐规则和矛盾升级路径。
- **任务简报** 是任务特定的；它定义结果、属性、边界和决策权，并且应包含指向相关上下文的精选指针，而非整个上下文本身。
- **连续性数据包** 是任务连续性的有限形式，团队可以根据对构建块蔓延的管理偏好，以两种方式实现。一些团队将其作为独立的构建块，在自然重置点（如交接、压缩事件、智能体更换、每日结束时）更新，从而避免连续性膨胀任务简报；另一些团队则将其作为“当前状态”部分折叠进

任务简报，并进行积极压缩。无论哪种方式，这里的“会话”指的是以上下文重置、智能体更换或明确交接点为界的、拥有稳定工作集的执行片段，而非一个神秘的时间概念。功能是相同的：保留对恢复至关重要的内容，并明确列出死胡同，以防系统带着“全新自信”重新探索它们。

5.8 上下文工作的实用文件结构

本章内容不依赖任何特定存储介质。代码仓库和文件树是一种实用的实例化，因为它们“免费”提供了版本控制和差异比较，但相同的逻辑结构亦可存在于工单系统、内部工作台或知识图谱中。关键在于，构建块应是版本化、可查询、受治理且访问可控的，并且指针对其目标受众是可解析的。下例使用文件路径具体说明；若你的系统将它们存储于他处，请将其视为逻辑名称。

指导包/（稳定，有版本）
任务/任务简报.md（规范的，受治理的）
任务/连续性数据包.md（连续性数据；可选折叠到简报中）
上下文卡片/（小型，有标签，可按需加载的片段）
上下文卡片/索引.yaml（标签 → 卡片的映射，含退役状态、来源、访问说明）

5.9 轻量级“上下文卡片”格式

每张上下文卡片应足够小，加载时不会造成淹没，且结构良好以保留优先级、来源和检索纪律。它应包含信息本身而不仅是元数据，并应明确“何时加载”，使检索保持有意识而非条件反射。一种实用格式如下：

标识： 稳定标识符
标签： 安全、认证、不变因素、性能、风格等。

优先级：不变因素 / 重要 / 有帮助
加载时机：触发器（如“涉及认证端点”、“修改模式”、“编辑热点路径”）
内容：智能体现阶段必须知晓的一屏摘要
来源指针：指向完整来源的链接/文件路径，附最后验证日期
退役状态：活跃 / 已弃用 / 已被……取代
访问备注：哪些角色可加载此卡（如需要用于合规）

这使得上下文可组合：你可以通过标签加载若干卡片，而非导入一本小说；你也可以明确让卡片退役，而非让过时的“真相”永远滞留。

5.10 衡量上下文工程

上下文指标应简单到可常规追踪，但你也应承认，一些“健康检查”指标计算成本过高，无法持续运行，更适合定期执行。目标是获得早期预警：你希望在过载和漂移演变成缺陷或令人困惑的评审意见之前，就检测到它们。你还应预期，这些指标会随着指导包和卡片系统的成熟而改善；这种改善就是复合经济收益。如果你的上下文指标从未改善，那说明你并非在进行工程化，你的系统只是在勉强应付。

5.10.1 指标：压缩频率与可见性

统计任务过程中压缩操作的频率及其报告情况。频繁压缩往往意味着工作集管理不善，或者检索时一口气灌输了太多内容。静默压缩才是心腹大患，它会破坏可审计性，让行为显得飘忽不定。改进目标并非“杜绝压缩”，而是“让压缩可预测、受标签与策略管控，并且一目了然”。

5.10.2 指标：上下文冲突率

统计工作集中出现矛盾、需要仲裁的频率（例如，“风格偏好与安全不变量打架”，或“文档 A 和文档 B 说法不一”）。这直接表明您的信息架构该退休了，来源管理也得紧一紧。冲突率一高，负载之下难免胡乱选择规则，事后看来就是判断失误。降低冲突率往往比堆砌更多上下文更管用。

5.10.3 指标：固定不变量丢弃率

统计智能体执行时违反或“遗忘”固定不变量的频率。不变量明明立在那里却被丢弃，可能意味着过载、优先级编码不当，或者基础模型在压力下能力不济。这个指标的关键在于区分“我们忘了固定”和“固定了却守不住”，从而对症下药。丢弃率逐渐下降，正是上下文工程见效的有力信号。

5.10.4 指标：上下文检索效率

通过代理指标（比如引用率——输出时引用已加载卡片或标签的频率?）或定期审计（监督员标记已加载片段的使用情况），来估算决策实际消耗了多少已加载的上下文。这种方法虽不完美且成本可能不菲，但可作为健康检查，不必当成持续的 KPI。效率低下说明您只是在“填鸭式”灌输上下文，而非教会检索与甄选。解决之道通常是采用更小的卡片、更明确的加载触发器和更严格的预算。

5.10.5 指标：上下文传递保真度

统计交接后，新执行者为恢复工作提出的澄清问题数量，并按任务复杂度归一化。若问题本应从连续性数据包（或简报的连续性部分）找到答案，则计为保真度失败。保真度高，意味着学习在会话与智能体间一脉相承；保真度低，则意味着您得反复为“重新推导上下文”买单。这是检验您在实践中是否攻克了学习悖论的最直观测度之一。

5.10.6 指标：分叉到合并的规范性

统计探索性分叉中，能产出带证据的紧凑结论并明确保留或丢弃的比例。分叉若将原始片段泄露到主上下文，就会污染工作集，让压缩乱成一团。规范性高，意味着探索为执行添砖加瓦，而非注水稀释。当团队嚷嚷“上下文给得够多了”而任务依旧跑偏时，这个指标尤其能诊断问题。

5.11 基于证据的监督：跨领域控制

任务工程和上下文工程虽能解决特定悖论，但基于证据的监督提供了跨领域控制，让所有四个悖论在大规模下也能管得住。证据把主张变成实打实的证明，让偏差无处藏身，无论监督者是人工审查员还是充当审计员的另一个 AI 队友。

关键原则是：信任靠证据赢得，而非盲目自信。AI 队友可能编出听起来头头是道、实则漏洞百出的漂亮话。基于证据的监督要求每项主张都有可验证的证据支撑：测试结果、分析输出、执行日志和决策轨迹。这样一来，审查就从“这看着对吗?”变成了“我们能证明它对吗?”。

基于证据的监督通过几种机制体现：

- **结构化证据包**：每个重要输出不仅要包含结果，还得附带证明其合理性的证据。这包括测试日志、验证输出、决策理由和探索档案。
- **机器可读清单**：证据必须能枚举、可检查。清单列明每个证据工件，包含稳定标识符、校验和以及指针，便于在人工审查前自动完成完整性检查。
- **渐进式披露**：证据按从摘要到细节的层次组织，让审查员能在合适深度进行审计，避免被原始数据淹没。
- **监管链**：证据必须保留来源。谁在何时、何种条件下、用什么工具生成的？这为后续的问题追溯和审计提供了可能。

基于证据的监督直接对抗所有四个悖论：

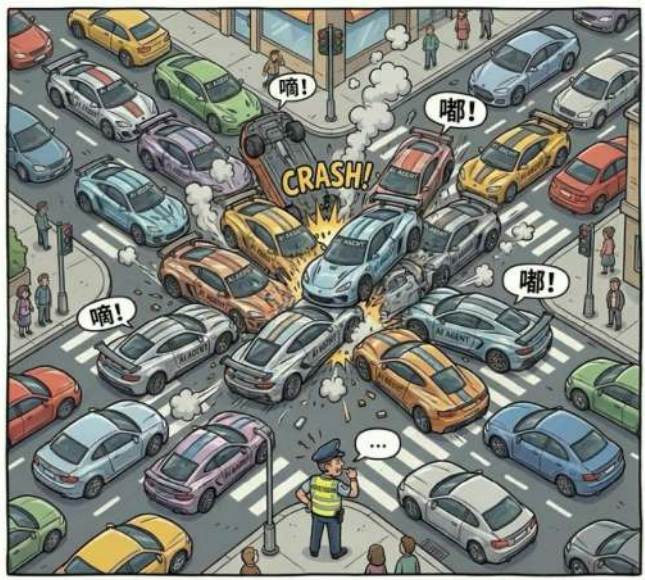
- **对抗热切性**：活儿干得再快，证据也得齐全
- **对抗上下文过载**：决策必须指明所用上下文
- **对抗隧道视野**：局部正确性得证明全局一致性
- **对抗学习差距**：过去的证据就是未来的捷径

5.12 从一个队友到完整的软件工程系统

在第二部分，我们让人与一个 AI 队友在已知弱点下仍能高效协作，这虽是必要的基础，却非智能体软件工程的终点。软件工程是团体赛，现代系统更是多对多的混战：多人、多 AI 队友、多依赖、多任务并行，还有无数决策争抢有限的注意力。唯有重新设计整个软件工程系统，以支持这种多对多的现实，而非把 AI 队友当成孤立的生产力工具，智能体软件工程的真正威力才会爆发。

下一部分（第三部分）不止是拥有多个 AI 队友；它关乎当软件工程的全部支柱——行动者（角色、职责、决策权）、流程（仪式、关卡、协调节奏）、工具（自动化、集成、验证、可观测性）和工件（版本化内容、知识持久化、证据打包方式）——必须协同适应时，会发生什么。适用于单个队友的控制措施，在大规模下会失灵，除非系统重设计。毕竟，一个上下文预算还好管理，一千个就能压垮你；一次集成尚可审查，一千次合并同时进行根本不可能；一个健忘的协作者还能忍，一群并行重复错误的机器大军可就灾难了。在企业环境中，保证工程虽必要，却不足够；您还得大规模协调工程、工作台工程、能力工程、信任工程和语言工程，才能让众多 AI 队友与人类安全高效地并肩作战。

智能体软件工程的终极力量，不在单个智能体能多快写代码，而在于重新设计的社会技术系统能安全驾驭众多随机性贡献者，同时确保结果可信、可读、可审计。这是从一对一编排到整个机群指挥的跃迁，目标不再是原始输出，而是在真实组织约束下持续的系统级表现。第三部分将明确迈出这一步，把可信赖性视为整个软件工程系统的端到端属性，而非任何个人、AI 队友或模型的私产。



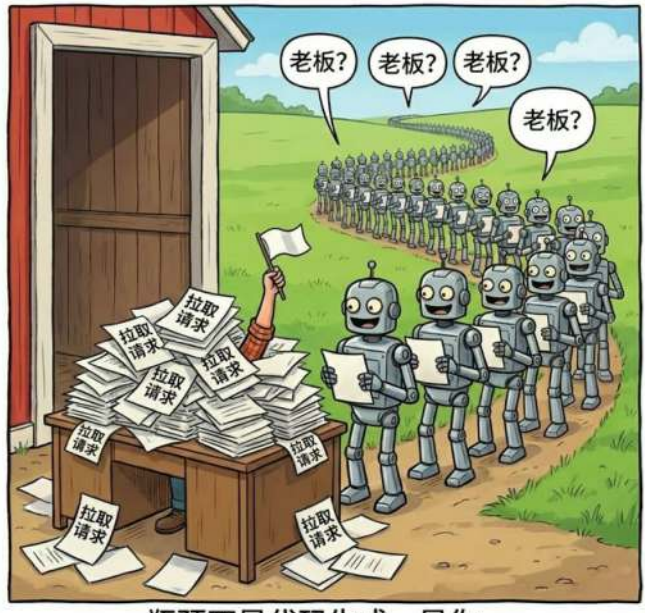
一个代理是天才。
一百个没有系统的代理只是交通堵塞。

Part III

面向 AI 队友舰队的平台工程

团队规模腾飞：面向 AI 队友舰队的平台工程

在第二部分，我们设计了一对一的关系：一人、一个随机性贡献者、一次只处理一个任务。这个框架虽是必要的起点，但还远未达到能真正撬动杠杆的阶段。真正的软件工程是多对多的：许多人、许多 AI 队友、许多并行执行的任务，以及最终汇入共享仓库的诸多成果。当你把行动者和并行度提上去之后，故障模式就不再是“代码写得烂”，而开始变成“编排一团糟”。



瓶颈不是代码生成。是你。

在团队规模上，决定性的瓶颈不再是点子的产生或实现速度，而是人类的注意力。如果每个决策都得请示人类，每段代码差异都要人类审查，每次合并都等人类点头，那么你的吞吐量就永远被那项稀缺资源卡着脖子。

AI 队友能轻而易举地用看似靠谱的输出淹没这一稀缺资源，而泛滥之所以危险，恰恰因为它看起来专业。所以说，团队规模的智能体软件工程，其实是一门分层信任的艺术：把决策下放到最低的安全层级，并随着风险和故障影响面的扩大，要求提供更高层级的证据。

要想可持续地授予更多自主权，唯一的办法就是在扩大自主权的同时，加强边界、证据和可审计性。只扩大自主权却不加固平台，那就是在扩大故障的影响面；只加强平台却不放开自主权，那就是在建造一个永远没法带来收益的安全系统。

本部分的核心观点是，自主权不仅是行动的自主，更是在明确定义的边界内进行推理和决策的自主。自主权应该通过持续的资格认证和反馈驱动的改进循环来增长，而不是靠一厢情愿的乐观主义。如果 AI 队友只会写代码，却不能安全地进行局部权衡，那它们就不过是仍需人类时刻盯着的高速打字员。如果 AI 队友可以不受约束地自由决策，那它们就成了高速事故生成器。让 AI 队友“飞行”，意味着设计空域：谁可以做什么、在哪儿操作、需要哪些遥测数据，以及出了问题该怎么办。

航空是个很贴切的类比，因为安全性的规模化，只有当决策从“有人盯着每一个动作”转变为“每个人都遵守一套具有共享可见性的可执行规则”时，才能实现。早期的飞行安全极度依赖集中控制和严格计划，因为飞行员和飞机都不够可靠，一旦偏离计划后果不堪设想。现代空域之所以高效，是因为我们分离了关注点：飞行员在定义好的操作边界内保留自主权，空中交通管制提供间隔和航线约束，而事故则反馈到流程、培训和工具中。整个系统安全，不是因为飞

行员完美无缺，而是因为空域经过精心设计，使得错误可检测、可控制、可从中学习。

这就是平台举措的意义所在。平台工程的出现，是因为组织们终于醒悟：要想扩展可靠性和速度，就不能指望每个团队都去重新发明一套交付“姿势”。你得铺设一条条现成的道路：通过提供自助服务能力、安全默认值以及可执行的防护栏，让可靠路径成为最简单的那条路。在代理式时代，平台不仅是生产力倍增器，更是让可信赖性成为软件交付的默认属性，而不是事后修补的补丁。这还带来一个实际的组织影响：这类系统需要专门的负责人。如果你指望核心产品团队在交付产品的同时，还要去构建协调基础设施、维护 AI 队友的定义和认证套件、并不断演进工作台和安全门——这活儿根本没法持续稳定地干下去。必须有一个专注的平台组来掌管这些共享基础设施，因为它能倍增每一个 AI 队友，进而倍增每一位开发者，并且这里是确保组织范围内安全与速度性价比最高的地方。

为了在大规模实现智能体软件工程的这一目标，我们引入了五门相互关联的工程学科，其关联方式与第二部分中任务和上下文工程的关联方式一脉相承：

- **协调工程**：防止 AI 队友和人类互相踩脚，并将并行输出整合成连贯的系统演进。
- **工作台工程**：构建两个环境，让人和 AI 队友都能在其中高速运转，而不会因界面错乱把对方拖下水。工作台工程将指挥平面与执行平面分离，让人类保持在判断模式，AI 队友保持在执行模式，并以默认捕获证据为前提。
- **能力工程**：通过角色、资格认证和持续改进来校准队友的能力，使得 AI 队友能做什么变得明确、可测试、可演进。
- **信任工程**：管理 AI 队友的权限，让委托关系清晰明了，并使故障可解释、可控制、可学习，即便系统本身充满不确定性。
- **语言工程**：解决编程语言如何成为共享沟通媒介的问题。

如果说第二部分是让一个随机性贡献者变得可靠，那么第三部分就是让一整支舰队变得可信。核心举措一脉相承：用工件取代氛围，用证据取代“信我没错”。不同之处在于，在团队规模上，工件必须支持路由、冲突避免、调度和渐进式披露，以免把人压垮。当系统正常运转时，人类只需把注意力集中在真正需要判断的决策上，其他所有事情都交给受治理的自主权以及可审计的证据去处理。

让 AI 队友飞行，不是一句口号。它是一个精心设计的空域必然产生的结果。

6 协调工程：防撞与自主流水线

6.1 根本矛盾

并行是提升智能体吞吐量的燃料，而协调则是必须付出的代价。你希望实现多对多协作，让众多 AI 队友和人类能够并行工作互不干扰，但又无法承受协调开销的增长速度超过产出。如果放任多个行动者在毫无协议的情况下修改共享代码表面，结果只能是共享状态的一片混沌、集成任务堆积如山，以及一大堆号称“合并就绪”却始终无法安全落地的改动。最让人头疼的莫过于集成阶段的协调：试图整合上百个各自“就绪”的变更，如果它们之间的交互从未被统筹规划过，那依然是一场噩梦。

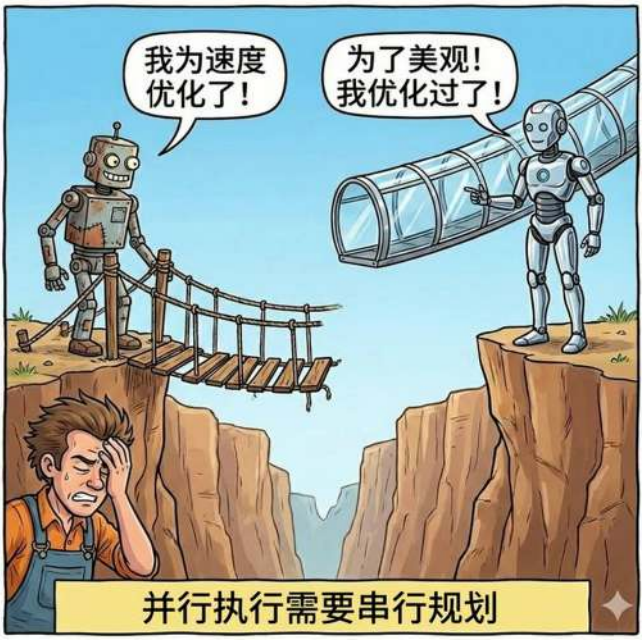
当团队规模扩大时，你需要协调的不是一件事，而是两件。其一是协调执行过程：即那些进行中的任务所产生的、会触及共享仓库的代码和工件。其二是协调整个舰队的“操作系统”：包括队友定义、工作准则、策略包、运行手册、工具链和模板——这些正是塑造舰队行为方式的要素。这里的失效模式通常不是“静默变更”，而是冲突：不同团队以互不兼容的方式定制自己的“操作系统”，导致相同的任务在不同团队中表现迥异。因此，协调工作必须同时覆盖任务本身以及产生任务的规则，并且对两者都确立清晰的所有权与冲突解决机制。

由此，协调工程衍生出两个相辅相成的方面。

- 首先是**防撞**：防止并行工作的智能体相互干扰，管理好共享表面，并确保各自独立产出能干净利落地集成。

- 其次是**流水线工程**：设计有意识的协作流程，让智能体在结构化的序列中交接工作，而人类则扮演工作流架构师的角色，而非实时调度员。

防撞回答的是："当众多智能体同时工作时，我们如何避免混乱?"流水线工程则回答："我们如何设计能让智能体有意协作的自主工作流?"两者缺一不可。一个只懂防撞却忽略流水线的组织，其智能体虽能避免冲突，却也永远不会真正协作。反之，如果设计了精美的流水线却忽视了防撞，那么一旦并行实例相互干扰，整个流水线就会瞬间崩溃。本章将深入探讨这两个方面。



一些早期的多智能体环境已经暗示了正确的设计原语：持久化的工作状态、显式的队列，以及能够跨越会话持续存在的交接包。关键不在于队列本身，而在于为异步工作赋予足够结构化的上下文，使得决策无需打断他人、重新审视整个项目宇宙就能做出。当一次咨询演变为包含选项、权衡、证据和建议的完整数据包时，组织就不再需要依赖会议作为协调的基础。这正是从"我们能运行许多 AI 队友"迈向"我们能依靠众多 AI 队友交付可信软件"的关键转变。

协调的严谨程度也必须因地制宜。当团队规模庞大、仓库复杂、变更影响深远、集成队列成为瓶颈时，本章所讨论的舰队级协议便至关重要。但对于小型项目或低风险表面，过度设计协议反而会拖慢学习速度、扼杀迭代活力。协调就像一个工程旋钮：在碰撞代价高昂时调高严谨度；在表面较小、偶尔冲突的成本低于持续规划的成本时，则果断调低。

6.2 核心概念：基于决策就绪包的异步协调与计划性集成

最基本的协调模式是默认异步，并采用决策就绪包。请求与决议都以结构化的**咨询请求包**形式传递，其中明确包含待决事项、相关上下文、可行选项、利弊权衡、已收集证据以及推荐路径。这将协调工作转化为可路由的任务而非中断性事件，允许人类及非人类专家（即专门的 AI 队友）成为可调用的端点，而无须让整个团队陷入同步等待。当协作的基本单位是一个个数据包时，人类与 AI 队友便能实现多对多协作，而不会让“注意力”成为瓶颈。

这个概念的另一半是计划性集成，而非事后补救。我们常说“并行探索，串行集成”，但若把集成视为工作完成后的步骤，这个想法就不完整了。在舰队规模上，你必须在开始执行前就规划好最终的集成路径：提交任务计划、声明修改表面和依赖关系，并安排集成窗口以避免碰撞。这并非官僚主义，而是以最低成本消除集成痛苦的捷径——因为提前排序的成本，远低于事后调和冲突的成本。

要理解其中的并行性，可以将其分为四个层次来考量：

- **空间并行**：多个行动者修改互不相交的文件或模块。如果模块边界清晰，这种方式通常成本低廉。
- **语义并行**：多个行动者在稳定接口的背后修改不同层次。只要抽象边界保持良好，这种方式也很经济。

- **时间并行**：重叠的工作按预定顺序依次落地。如果时间线明确且有版本控制支持，其成本可控。
- **决策并行**：多个行动者探索不同选项并产出竞争性计划。如果工作台能够比较、择优或融合这些计划，而不引发集成混乱，这种方式也能高效运作。

由此可见，集成规划远不止合并代码那么简单。它关乎决定你正在利用哪个层次的并行性，并确保现有工程系统能够支撑该层次，而非意外陷入成本最高的那种模式。

6.3 协调工程的子组件

协调工程可以分解为多个具体、可实施且可审计的机制。工作分解协议明确了职责归属、接口定义和交接规范，确保并行工作不会意外漂移到重叠区域。异步协调机制通过队列路由咨询和记录**决议记录**，使决策过程持久化且非中断性，让正确的决策者看到正确的信息，无需开会。防撞机制则利用所有权边界、合并排序，以及在必要时才启用的锁定策略，防止重叠的编辑演变成合并战争。

随着规模扩大，有两个机制变得尤为重要，因为舰队本身就是一个动态演进的目标。其一，舰队操作系统的变更协调：这意味着工作指导和策略更新需要版本化、经过评审，并遵循兼容性规则进行部署，而不是被复制成五花八门的本地变体，在静默中分道扬镳。其二，集成调度：你需要将集成队列视为一个具有容量、优先级和排序规则的一等公民系统，而不是任由拉取请求自然堆积形成的混乱局面。持续集成的思想在此很有启发：每次变更都会触发自动化检查和结构化反馈，而工程系统的设计目标，就是通过尽早发现不兼容性来降低集成失败的成本。当这些组件都到位时，协调就从一个依赖社交技巧的活动，转变为一个可管理的系统工程问题。

6.4 协调工程中的关键实践

6.4.1 实践一：设计清晰接缝，减少不必要的并行

首要任务是实现真正的隔离：通过架构层面的接缝、接口边界和明确的所有权表面，让众多行动者能在互不触及对方脆弱区域的情况下并行工作。这既是协调问题，也是软件设计问题：插件架构、驱动程序、服务边界和清晰的模块契约，正是让一个系统能友好接纳大规模“劳动力”的关键。此时，康威定律变得前所未有地重要——系统结构终将反映出你实际采用的协调路径，而舰队规模使得这些路径具有决定性。同样，帕纳斯的信息隐藏原则也至关重要：稳定的接口使得语义并行成本低廉且稳定可靠。接着，要有意识地规划任务顺序。当顺序执行能更快完成且彻底避免冲突时，同时开干二十个互相重叠的任务往往得不偿失。目标不是追求最大的并行度，而是通过最小化协调开销来实现最大的有效吞吐量。

6.4.2 实践二：执行前向冲突管理器提交计划

在 AI 队友开始执行前，要求它们将任务计划作为结构化数据包提交，明确声明将要触及的表面、依赖关系、预期接口以及建议的集成时间。一个半自动、半人工的冲突管理器会评估这些计划图，并安排工作顺序以最小化碰撞。这就是主动集成：你通过分配集成窗口、在代码实际产生之前就做出排序决策，从而避免坠入集成地狱。这种协调是在团队或组织层面进行的，而非单个人管理单个 AI 队友的层面，因为其核心价值就在于预防整个舰队范围内跨任务的碰撞。

6.4.3 实践三：在受控的隔离工作空间中执行

在工作进行时，AI 队友需要独立的空间来探索和迭代，避免影响他人。这意味着要提供隔离的沙箱环境、建立分支管理规范，并明确界定队友何时可以接触共享表面的规则。

通过数据包来路由交接和咨询是切实可行的，因为这本就是成熟的分布式团队在以“人类速度”工作时的方式。AI 队友从中获益更多，因为它们的节奏之快，使得短暂的、即兴的协调瞬间变得过时。真正的问题不在于数据包模式是否有效，而在于我们的工具和习惯能否跟上智能体执行的疾速节奏。

6.4.4 实践四：用分层就绪状态把关，而非单一的合并状态

在舰队规模下，“合并就绪”是必要但不充分的条件。一个变更可能完全正确，但当下仍不适合集成，因为它与其他工作冲突、会破坏表面稳定性，或者需要某个依赖先行落地。因此，协调工程将就绪性视为一个分层递进的概念：代码完成不等于合并就绪，合并就绪不等于集成就绪，而集成就绪也不等于“时机已到”。当就绪性是分层判定时，集成过程就会变得可预测，而非积压成山。

6.4.5 实践五：利用 AI 队友原生策略解决集成冲突

当冲突不可避免地发生时，应将解决过程视为在不同策略间做出选择，而非仅仅触发“修复合并冲突”的机械反应。我们必须超越传统的人类工作习惯。一种选择是经典的调和：解决文本或语义冲突后继续推进，AI 队友在这方面可能非常擅长。另一种则是 AI 队友原生的重做策略：让第一个变更落地，然后根据新的代码现状重新运行第二个任务，由队友生成一个与已集成内容保持一致的新实现。这往往能产出更健康的软件，因为它用约束更新后的、连贯的重新设计取代了为走出集成地狱而做的最小化编辑。当重做成本低廉且由 AI 队友执行时，这种策略的代价很小。

智能体原生的冲突解决方式，也是对人类时代工作模式的一次质的飞跃。过去，集成冲突代价高昂，因为人类通常只

能在代码行层面被动应对：解决冲突、继续前进，并默默承受随之而来的隐性耦合债务。有了 AI 队友，你通常可以承担更高质量的操作：AI 队友可以退一步，提取一个共享抽象，在此抽象下重新表达那些相互竞争的变更，然后重新运行完整的测试套件以确保合并没有引入回归。当然，这只有在反馈循环强大且可靠的情况下才是安全的。它依赖于完善的测试和检查机制，使得“我们破坏了什么吗”成为一个可被明确回答的问题，而非一种侥幸的期待。

速度也改变了排序的经济性。在人类工作流中，等待一位工程师完成工作再开始下一项任务，可能会拖垮整个时间表。但在 AI 队友工作流中，“立即开始”和“等第一个任务落地后再开始”之间的时间差通常很小，特别是当第二个任务的重做成本低廉，且平台能快速验证结果时。这意味着，优先考虑计划的调度和有意识的串行化，反而可能是最快的路径，而非最慢的——因为你用小额的等待时间，换取了消除大规模合并混乱的巨大收益。

我们还需要诚实地面对一个观察：AI 队友通常非常擅长在现有代码库中“让它跑起来”，包括进行大刀阔斧的重写和快速重构。然而，尚未得到充分验证的是，AI 队友在面对多个并行任务时，是否总能具备良好的集成判断力，因为真正的难点在于排序和兼容性，而非单纯实现。正是这种不确定性，使得冲突管理器和调度表面应被视为关键基础设施，而非可有可无的点缀。工程系统必须假设集成错误会发生，并使解决它们的成本保持低廉。

AI 队友原生重做也改变了治理模式：第二个任务必须再次通过完整的**合并就绪包**关卡，因为它现在被视为一个新的变更。这并非浪费，而是在代码以机器速度产生的时代维持信任的必要方式。久而久之，你可以衡量哪种策略导致了更少的事故和更低的返工率，然后让冲突管理器偏向于更安全的默认选项。关键在于，集成不仅是一项技术操作，也是一个关于工作流的战略决策。

6.4.6 实践六：在组织各层级协调“协调基础设施”

一旦你拥有一个舰队，你发布的变更就不仅针对代码，也针对队友定义、工作指南、工具链、运行手册和策略。这些工件在多个层级演化：项目、团队、部门乃至整个公司，每一层都可能对其下层进行定制。定制固然有价值，但也可能降低效率或引发冲突，因此它需要像代码一样，有明确的所有权、评审流程和自动化质量检查。鉴于能力工程这一领域尚处早期，期望由一支专家团队来管理这套**协调基础设施**，持续监控 AI 队友行为，并防止局部优化导致系统碎片化，是合理的设想。这正是你的精英智能体软件工程师应该转型为 AI 队友教练的地方。

6.4.7 实践七：为自主执行而设计流水线

在创建多智能体工作流时，应将其设计为无需人工干预即可自主运行，除非到达明确设定的审批关卡。这意味着需要定义具有清晰进入和退出标准的阶段，明确智能体之间的交接契约，并内置故障处理机制——而非默认设置为“询问人类”。如果人类必须在每个阶段转换时进行干预，那么你构建的就只是一个增加了额外步骤的手动编排流程，失去了流水线工程的经济优势。一个简单的测试方法是：这个流水线能否在人类睡觉时独立运行完成？如果不能，那就请识别哪些对人类注意力的依赖是真正必要的（例如高风险关卡），哪些则是应该填补的设计空白。

6.4.8 实践 8：为智能体交接订立契约

每次智能体之间的交接都必须有明确的契约：发送方必须提供什么，接收方期望什么，交接包的格式为何，以及出现不匹配时如何处理。请像对待服务间的 API 契约一样对待这些交接。它们必须清晰、有版本控制，并经过验证。一旦交接验证失败，接收方智能体应直接拒绝，并提供结构化

反馈，而不是试图处理那些残缺或格式错误的输入。那种指望智能体根据上下文“自行领会”的模糊交接，只会带来调试噩梦，让流水线故障变得扑朔迷离。

6.4.9 实践 9：在决策点，而非执行点设置人工审批

把人工审批放在最能体现人类判断价值的地方：比如高风险决策、模糊的权衡取舍，或是流水线设计边界之外的特殊情况。别在那些只是按部就班执行的常规步骤上设卡。这种区分至关重要，因为审批疲劳真实存在：一个人如果被迫批准了五十次常规操作，到第五十一次时，很可能就草草了事，而这一次可能偏偏就是需要他打起精神的关键环节。因此，要根据风险给审批点分级：让常规工作自主流动，让真正有风险的工作暂停待审。随着流水线被时间证明其可靠性，你可以逐步撤销一些审批点——这是一种在流程层面（而非单个任务层面）应用的渐进式授权。

6.5 协调中的集成工程

输出的协调常常是团队在智能体高吞吐量下崩溃的环节，因为集成能力成了瓶颈。我们的目标不是构建更好的事后合并关卡，而是通过规划和调度，将集成冲突的概率降至近乎为零。这始于支持大规模并行开发的架构，并持续将提交给冲突管理器的**概念计划**调度到集成窗口中。如果你等到一百个变更堆在那里才发现它们相互冲突，那代价就太大了。本章采取的协调策略，特意强调计划先行：设立就绪度阶梯，目的就是为了保持计划集成队列的健康，而不是用它来替代规划本身。

一旦主动规划层建立起来，渐进式验证就成为维持队列健康的稳定器。验证应随就绪度逐步展开：先是局部正确性，然后是合并证据，接着是集成兼容性与时序对齐。通过功

能级协调让依赖关系显性化，从而让并行工作可预测地汇聚，而不是在最后关头才发生冲突。当这些机制运转良好时，“集成就绪”就成了一种可验证的状态，而非一厢情愿的声明。

6.6 流水线工程：设计自主的多智能体工作流

协调工程回答的问题是：许多智能体并行工作时，如何避免相互“撞车”？而流水线工程则回答一个互补的问题：如何设计有意识的协作，让智能体在结构化的序列中交接工作？两者缺一不可。没有编排的协调，会制造出一群互不干扰、但也绝不主动协作的智能体。没有协调的编排，则能造出漂亮的流水线，但一旦并行实例冲突，便会瞬间崩溃。这两门学问相辅相成。

6.6.1 人类是工作流架构师，而非实时调度员

对多智能体工作的一个朴素想象是手动调度：人类让智能体 A 写代码，等着；然后让智能体 B 审查，再等着；接着让智能体 C 测试……这根本不可扩展。人类会成为瓶颈，宝贵的注意力消耗在机械的路由指令上，而 AI 队友的经济性优势也会在协调开销中蒸发殆尽。

可扩展的模式是**流水线设计**：人类一次性设计好一个工作流，定义阶段、智能体角色、交接协议和审批点。然后工作流自主执行，智能体按设计互相交接工作。人类只在预设的审批点介入，通常是为了处理高风险决策或做最终批准，而不是在每个阶段转换时都插一手。

这是一种在流程层面（而非单个任务层面）的授权。你并非信任某个智能体能独立完成任务，而是信任一套流水线能按部就班执行一系列任务，并在每个阶段都设有适当的检

查点。这种信任不是盲目的；它通过明确的交接契约、阶段级验证和基于风险分级的审批点来精心设计。

6.6.2 剖析编排流水线

一个设计良好的流水线，必须明确以下五个部分：

- **阶段** 定义了工作的内容和顺序。每个阶段都有明确的目的、指定的智能体角色（或多个角色），以及清晰的准入和退出标准。阶段可以是顺序的、并行的，或是基于前一阶段输出的条件分支。
- **交接协议** 定义了工作如何在智能体间传递。交接不是简单的一句“上一个智能体处理过了”；它是一个结构化的数据包，内含做了什么、产生了什么证据、接收方应该做什么以及适用的验收标准。交接就是流水线工程中的 API 契约。
- **审批点** 定义了人类必须介入的位置。并非每个阶段转换都需要人工点头；那样就又退回到手动调度了。审批点应设置在风险合适的地方：在不可逆的操作之前、在高风险决策之后，或是当流水线遇到了其设计范围之外的情况时。风险评估决定了审批点的摆放。
- **证据要求** 定义了每个阶段必须产出什么。证据不是可有可无的文档；它是阶段正确完成的证明，也是下一阶段工作的基石。没有证据要求，流水线就会沦为一场信任游戏，而非工程系统。
- **故障处理** 定义了一个阶段失败或产出意外结果时该怎么办。选项包括用不同参数重试、上报给人类、回滚到前一阶段，或直接中止流水线。故障处理必须明确，因为“流水线卡住了”不是一个可接受的故障模式。

6.6.3 常见的流水线结构

在智能体软件工程中，有几种流水线结构反复出现：

顺序验证 是最简单的结构：工作依次流经一系列阶段，每个阶段验证并可能转换前一阶段的输出。一个典型例子是：生成器 → 静态分析器 → 审查者 → 测试者 → 集成验证器。每个阶段都可以拒绝工作并将其打回更早阶段，从而形成一个质量“棘轮”，只有满足所有阶段标准的工作才能向前推进。

并行探索与合并 将同一问题分派给多个独立工作的智能体，然后将它们的输出路由至一个合并智能体，由它来综合或选出最佳方案。这是可操作的 N 版本探索。合并智能体需要有明确的标准来决定是选择（挑最好的）、综合（融合多个方案的优点）还是上报（方案差异太大，无法自动合并）。

分层委托 让一个主导智能体接收任务，将其拆解为子任务，分派给各领域的专家智能体，最后整合结果。这模仿了人类技术负责人向团队成员派活的方式。主导智能体掌控整体任务和集成；专家则负责边界清晰的子问题。这种结构对复杂任务很有效，但需要主导方和专家之间有清晰的接口契约。

审查循环 将工作路由给一个审查者智能体，它可以批准、拒绝并给出反馈，或直接上报。如果被拒绝，原始智能体需根据反馈进行修改并重新提交。这种结构必须包含迭代限制以防止无限循环：在 N 次拒绝后，应上报给人类或中止。反馈必须有足够的结构性，让原始智能体能够据此行动，而无需猜测意图。

分阶段审批 只在风险合适的地方，而非每次转换时插入人工审批点。低风险阶段自主执行；高风险阶段则暂停等待人工审查。流水线定义会明确指出哪些阶段有关卡，以及人类需要哪些信息来做决定。这种结构在自主性和监督之间取得了平衡。

6.6.4 智能体间的交接

交接是流水线运转的核心机制。它不是聊天线程里的一句话，而是一个随工作传递的结构化数据包，让接收方智能体无需重建上下文就能立刻行动。

一个设计良好的交接包通常包括：

- **工作工件**：前一阶段的实际产出（代码、文档、分析结果等）
- **证据包**：证明前一阶段已正确完成的证据（测试结果、分析输出、验证日志）
- **上下文摘要**：接收方需要了解的关于任务背景、约束条件和迄今所做决策的信息
- **任务说明**：接收方在本阶段应该做什么，包括本阶段的验收标准
- **交接元数据**：用于审计追踪的阶段标识、时间戳和来源信息

接收方智能体在接手前应该能验证这个包。验证内容包括：检查必要证据是否齐全、工作工件是否处于声称的状态、任务说明是否在自身能力范围内。如果验证失败，交接应被拒绝并退回给发送阶段，而不是让不良状态在流水线中蔓延。

6.6.5 流水线工程的经济账

流水线工程改变了多智能体工作的成本结构：

设计成本是一次性的。打造一个设计精良的流水线，需要仔细思考阶段划分、交接设计、审批点和故障处理。这是实实在在的工程活儿。但一旦流水线建成，就可以反复运行，而无需再次支付设计成本。

执行成本随运行次数，而非人类注意力扩展。每次流水线运行都会消耗计算资源和智能体时间，但除了预设的审批点

外，它不消耗人类注意力。这才是根本的经济优势：你在不增加注意力预算的前提下，成倍提高了吞吐量。

维护成本持续存在，但有限。当需求变化、智能体能力更新或故障模式暴露出设计缺陷时，流水线需要更新。这种维护成本是真实的，但也是可管理的，因为它集中在流水线定义本身，而非分散在每次执行中。

盈亏平衡点比你想象的低。即便一个流水线只运行几次，只要它能替代复杂多智能体工作的手动调度，就值得设计。关键问题不是“这流水线会运行一千次吗?”，而是“设计成本是否低于‘手动调度成本乘以预期运行次数’的总和?”

6.6.6 何时用流水线，何时用手动调度？

并非所有的多智能体交互都需要正式的流水线。当工作确实是临时的、顺序无法预测且需要人类步步为营的判断，或者设计成本远超执行成本时，手动调度是合适的。对于探索性工作，当你连阶段都还没摸清时，先用手动调度。

当工作顺序可重复、阶段和交接可以预先定义、人类注意力成为瓶颈，或者你需要审计工作在系统中的流转轨迹时，流水线就是合适的选择。对于生产工作流、对于必须始终如一执行的质量关卡，以及对于那些“忘了运行安全扫描”是不可接受故障的流程，请使用流水线。

从手动到流水线的过渡通常是一个自然生长的过程：你先手动调度几次，识别出其中的固定模式，然后将其编码为流水线。这很健康。过早设计流水线可能会让简单情况变得复杂，而永远不升级到流水线，则会让你错失潜在的价值。

6.7 协调工程模式

6.7.1 模式：使用“决策就绪”咨询包的异步协调

将咨询视为结构化的“中断”，它们异步到达，并自带快速决策所需的一切。一个好的咨询包应包含备选方案、权衡分析、证据和推荐决策，这样人类就无需从头重建上下文。这让人类变成了无需开会就能调用的“端点”，也让专业的 AI 队友变成了无需混乱聊天就能调用的“端点”。结果是更低的注意力负载和更高的决策吞吐量。

6.7.2 模式：计划先行的集成调度

要求智能体在执行开始前，就通过**概念计划**声明变更范围、依赖关系和提议的时序。将这些计划路由给冲突管理器，由它在代码实际写出之前就调度集成窗口并解决潜在冲突。这就是你如何避免大量“合并就绪”的工作实际上却无法安全落地的窘境。这也是你如何防止并行性变成隐形的等待时间。

6.7.3 模式：用于提升决策质量与创新的 N 版本探索

面对高风险决策时，生成多个独立的计划或解决方案候选，并进行明确的比较。这个过程不仅能捕捉错误，还能揭示隐藏的权衡取舍，并允许融合不同方法中的最佳元素。

N 版本探索将分歧从项目延误转变为战略资产。然而，这种方法只有在工具链具备特定能力时才有效：

- **并排比较：**能并行可视化各选项及其支持证据，以便客观评估利弊。
- **语义融合：**支持一种新型的合并操作，能将针对同一目标设计的不同方案融合成一个更优的混合体。

- **权衡揭示：**能识别出各个方案的不同侧重点（例如一个优化性能，另一个优先可维护性），从而促成有意识的权衡。

6.7.4 模式：区分“工作中”与“工作后”的协调

在“工作中”阶段，应强制执行隔离空间、明确的所有权和受控的接触点，这样探索工作才不会破坏性地重叠，部分决策也不会泄露到共享变更面。此时，交接、咨询包和声明的变更面至关重要，它们能防止进行中的工作变成一团无形的乱麻。在“工作后”阶段，则应强制执行**分层就绪**、集成调度和版本化的时序决策，这样产出才能可预测地汇聚，最终的落地也能保持连贯。将这两者视为不同的模式，可以避免一种常见的失败：团队“只在合并时才协调”，结果发现自己不得不在压力最大的时候进行最复杂的协调。

6.7.5 模式：像维护操作系统那样协调团队演进

将队友定义、指导手册、策略包和模板视为带有兼容性规则的共享基础设施。通过代码审查、自动化检查和有计划的版本来协调这些内容的变更，防止团队“分叉”成行为互不兼容的孤岛。目标不是消除定制化，而是确保定制化的安全性和可度量性。

6.7.6 模式：顺序验证流水线

将工作组织成一个序列流程，每个环节都要验证，甚至转换上一环节的产出。典型的例子是：生成器 → 静态分析器 → 审查者 → 测试者 → 集成验证器。每个环节都对其接收和交付的交接物设有明确的验收标准。拒绝则退回，批准才放行。这就形成了一个“质量棘轮”——工作只有满足每一关的标准，才能向前推进。此模式的威力在于，它将质量关卡变为自动机制而非可选项，并将验证工作分摊给各专精的智能体，而非让单个审查者不堪重负。

6.7.7 模式：并行探索，结构化合并

将同一问题派发给多个独立工作的智能体，让它们的产出汇聚到一个合并智能体手中，由它进行综合或择优选取。这相当于为生产环境打造了“N 版本探索”。合并智能体需要明确的行动准则：要么依据既定指标选出最佳方案，要么博采众长、组合多个方案的优点，要么在方案差异过大、无法自动合并时，提请上级裁决。此模式适用于设计决策、实现方法等任何值得探索解决方案空间的场合。它要求问题描述足够稳定，足以分派给多个智能体，且合并准则必须在探索开始前就界定清楚。

6.7.8 模式：分层委托，集中集成

一个主导智能体接收任务后，将其分解为接口清晰的子任务，分派给各领域的专家智能体，最后集成所有结果。主导者统领全局任务、负责分解与集成；专家则在其专业范围内，拥有明确的子问题边界。这模仿了人类技术负责人向团队成员分配工作的方式。采用此模式，需要主导者与专家之间有清晰的接口契约，明确定义如何横跨多个子任务处理共性问题，并进行集成验证，确保组装后的各部分能真正协同工作。主导智能体必须具备集成的判断力，而非仅仅是个派活儿的。

6.7.9 模式：审查循环，限制迭代

让工作流经一个审查智能体，它可以批准、带结构化反馈地拒绝，或向上提请裁决。若被拒，原智能体需根据反馈修改并重新提交。关键在于设计迭代限制：若连续 N 次被拒仍未获批准，则循环将升级至人工审查或直接中止，而非无限循环下去。反馈必须具体、可操作；像“再试一次”这种含糊其辞的反馈只会导致无效返工，毫无进展。此模式适用于审查质量至关重要，且原智能体有能力根据反馈改

进的场合。若审查者与生产者对成功的标准南辕北辙，此模式则行不通。

6.7.10 模式：分阶段审批点

在流水线的不同风险节点设置人工审批点，而非在每个环节转换处都设卡。需明确界定：哪些阶段风险低，可自主执行；哪些风险中等，需异步通知人类；哪些风险高，必须暂停等待明确批准。审批点的设置应基于实际风险，而非主观舒适度；目的是将人类注意力聚焦在能真正增值的环节。随着流水线被证明可靠，审批点可逐步移除或降低等级。此模式要求每个审批点都能为人类提供充分的决策信息：流水线已完成什么、接下来计划做什么、支持该计划的证据是什么、以及风险何在。那些需要人类自己费力重建上下文的审批点，本身就违背了其初衷。

6.8 协调工程反模式

6.8.1 反模式：合并就绪工作的集成积压

如果合并就绪的变更因为会破坏系统稳定性而无法集成，那么它们积压再多也不算进展。当团队将“合并就绪”视为终点线，却忽略了集成时机和兼容性时，就会落入此陷阱。人类随后被迫在激增的集成队列中疲于分类处理，这彻底违背了智能体软件工程追求吞吐量的初衷。解决之道在于：规划优先的调度策略，加上明确的集成就绪标准与时机决策。

6.8.2 反模式：无计划并行，共享表面遭殃

多个行动者在缺乏协议保证的情况下，编辑重叠区域，注定会导致集成之痛。起初产出看似很快，直到合并冲突堆积成山，人类注意力便成了瓶颈。这通常是意外发生的：系

统缺乏清晰的接缝，团队误以为“人多力量大”就等于“进展快”。解决方法是：隔离、明确所有权、并排序执行。

6.8.3 反模式：非结构化的同步协调

当决策在聊天线程或会议中产生，却没有形成持久化的数据包时，系统就丧失了路由、审计和复用决策的能力。人类会就相同的问题反复争论，而 AI 队友也无法依赖决策依据——因为它根本“加载”不了。这也使得集成规划变得不可能，因为依赖关系未能以可计算的形式捕获。解决之道是：采用异步数据包，辅以版本化的决议记录。

6.8.4 反模式：人在每个循环中

在每个智能体环节之间都要求人工批准，这重新制造了流水线工程本欲解决的注意力瓶颈。人类沦为“传声筒”而非决策者，批准疲劳迅速蔓延。第五十一次批准被草草盖章放行，而问题恰恰可能就出在这次。这种反模式通常源于对智能体缺乏信任，但解决方法不是增设更多关卡；而是强化环节级验证、明确交接契约、并按风险等级设置审批点，从而将人类注意力保留在真正的高风险环节上。如果你压根不信任智能体之间的交接，那就该改进流水线设计，而不是一味增加人工检查点。

6.8.5 反模式：隐式交接

智能体之间在不传递结构化交接数据包的情况下移交工作，会造成流程黑洞，让调试几乎无从下手。流水线一旦失败，“上一个智能体到底交了啥?”就变成了在日志和聊天记录里“考古”。隐式交接还导致接收方智能体难以在继续之前验证是否拿到了所需之物，从而让错误状态在流水线中蔓延。解决方法是定义明确的交接契约，规定数据包格式、必备内容和验证规则。每次交接都应是一个独立的、可供检查的工件。

6.8.6 反模式：单体工作流

试图用一个单一的流水线定义来编码所有可能路径、所有边界情况和所有条件分支，其结果必然是难以维护。当工作流的定义比它所要编排的工作本身更难理解时，你就已经过度设计了。单体工作流还很脆弱，对任何部分的改动都可能牵一发而动全身。解决之道在于：构建接口清晰、可组合的流水线段。先做好那些“小而美”、各司其职的流水线，再将它们组合成更大的工作流。每一段都应独立可测、易于理解。

6.8.7 反模式：缺乏可观测性的编排

如果你无法看清工作位于流水线的何处、哪些交接成功或失败、瓶颈在何处形成、以及每个阶段耗时多少，那么你既无法调试，也无法改进工作流。当团队将流水线工程仅仅视为设计演练，而忘了运维需求时，这种反模式便会浮现。解决方法是为流水线提供一流的可观测性：阶段完成事件、交接日志、计时指标和故障追踪。你应当能轻松回答“这个流水线运行的当前状态如何?”以及“那个流水线运行失败的原因是什么?”，而无需去啃智能体的聊天日志。

6.9 主要构建模块

协调工程的构建模块，是让 N 对 N 的协作局面保持清晰可控的基石。

- **工作流程运行手册** 定义了关于任务分解、权责归属、咨询包和集成调度的协议。
- **任务计划包** 声明了工作范围、依赖关系和执行时机意图，以便冲突管理器能主动进行工作排序。
- **合并就绪包与集成就绪标准** 捆绑了支撑信任决策的证据，版本化的时机决议则记录了何时集成以及为何集成。

- **流水线定义** 将多智能体工作流编码为版本化的工件，内容包括：阶段、各阶段的智能体角色、阶段间的交接契约、审批点、各阶段的证据要求、故障处理规则和迭代限制。流水线定义如同代码，由人类编写并实施版本控制。它引用“队友定义”来指派各角色，并可调用工作流程运行手册来执行阶段级协议。流水线定义支持自主执行：工作流依据定义自动运行，人类仅在预设的审批点处进行干预。
- **交接数据包** 是在流水线中智能体之间传递的结构化工件。每个数据包包含工作产出、证据包、上下文摘要、下一阶段的任务说明以及来源元数据。交接数据包是流水线工程的 API 契约；它们使得智能体之间的传递变得明确、可验证、可审计。

6.10 衡量协调工程

6.10.1 指标：集成冲突率

定义为需要冲突调解工作或 AI 队友自身返工的集成尝试所占的比例。需按工作范围和风险等级分别追踪，因为某些区域冲突率应趋近于零，而其他复杂区域可能允许稍高。冲突率上升意味着你的系统接缝薄弱或调度能力不足。冲突率下降，则是主动协调机制生效的最清晰信号。

6.10.2 指标：解决集成冲突耗时

分别测量人类耗时和 AI 队友耗时。目标并非“零返工”，而是“少耗人力”。如果冲突能通过 AI 队友低成本返工、以最小人力介入解决，系统仍具扩展性。若冲突频繁需要人工仲裁，注意力就再次成了瓶颈。此指标能揭示你的冲突管理器和就绪阶梯是否真正奏效。

6.10.3 指标：误预测冲突率及根因分类

追踪那些计划预测无冲突、实际却出现冲突的情况，并对原因进行分类。常见原因包括：缺失依赖意识、错误的工作范围声明、隐藏的语义耦合或不稳定的接口。此指标能改进你的协调基础，因为它告诉你计划模型在何处失准。随着时间的推移，你应期望在热点区域看到更少的意外和更优的接缝设计。

6.10.4 指标：流水线自动化率

定义为在没有计划外人工干预的情况下完成的流水线执行所占比例。“计划外”指在设计好的审批点之外——人类介入是因为流程出错，而非流水线在审批点处正常暂停。低自动化率表明流水线设计不完善、交接契约失效或故障处理不足。需按流水线类型分别追踪，以识别哪些工作流需要重新设计。自动化率随时间上升，标志着你的流水线工程正走向成熟。

6.10.5 指标：交接成功率

追踪智能体之间交接成功（未被拒绝或升级）的比例，并按具体的阶段转换进行细分。特定交接成功率低，表明发送方与接收方智能体之间存在契约失配：要么发送方没产出接收方期望的东西，要么接收方的验证过于严苛。此指标有助于识别流水线中的薄弱接缝，并优先改进相应的交接契约。

6.10.6 指标：流水线周期时间

测量从流水线启动到完成的端到端时间，并按各阶段耗时、在审批点处的等待时间进行细分。这能揭示瓶颈所在：是卡在某个智能体阶段、交接开销大、人工批准延迟，还是陷入了重试循环？随着流水线成熟以及你识别并解决瓶颈，周期时间应逐步改善。比较类似流水线运行的周期时间，以

检测是否出现倒退。特定运行中出现异常长的周期时间，通常意味着存在值得深究的故障处理或重试行为。

6.11 总结

协调工程是使大规模智能体软件工程成为可能的学科。它包含两个互补的方面：碰撞避免（防止并行工作的智能体相互干扰）和流水线工程（设计有意的协作，使智能体在结构化序列中交接工作）。两者缺一不可。没有流水线的碰撞避免，会产生一群永不冲突但也从不协作的智能体。没有碰撞避免的流水线，则会打造出精美却脆弱的工作流——一旦并行实例相互干扰，便会崩溃。

通过以计划集成取代临时并行、以异步决策包取代同步打断、以主动调度取代被动冲突解决、以自主流水线取代手动编排，我们得以构建一个系统。在这个系统中，数百个 AI 队友可以高效工作，既不会制造混乱，也无需在每一步都消耗人类注意力。

本章所述的模式与实践，并非理论空想，而是吞吐量激增时的运维必需品。从接缝工程和计划提交入手。逐步增加冲突管理和分层就绪机制。为可重复的多智能体工作流设计流水线，包含明确的交接契约和按风险分级的审批点。随着智能体规模增长，在此基础上持续构建。目标并非在第一日就实现完美协调，而是通过系统性改进，防止集成瓶颈和注意力瓶颈抵消掉智能体加速带来的红利。

7 工作台工程：两种模式，两套环境

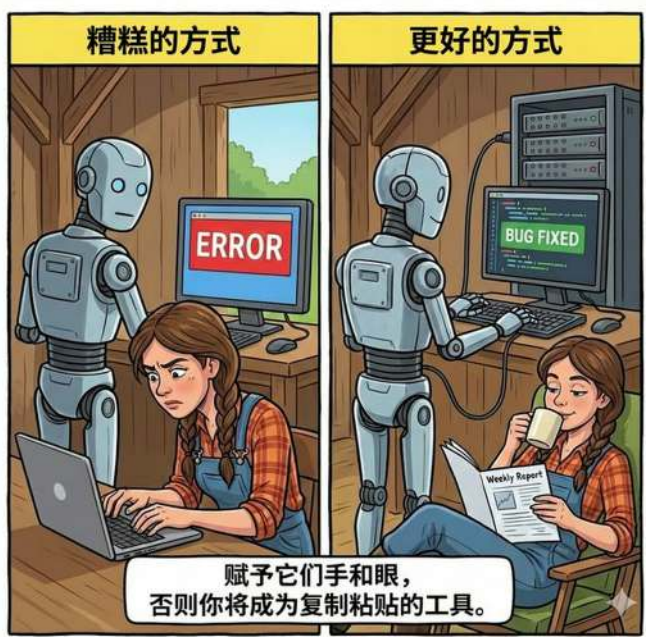
7.1 根本矛盾

团队规模一旦上来，接口就成了瓶颈与管控点。人类需要一个命令环境：它能压缩决策、突出风险，还能以渐进披露的方式呈现证据。毕竟，人类审查能力本身就是一个限制因素，面对原始日志和海量代码差异，根本看不过来。而 AI 队友则需要一个能安全快速循环的执行环境：并行计算、密闭运行、稳定的工具链，以及判断对错的结构化反馈信号。工作台工程之所以必要，就是因为如果硬把人类塞进 AI 队友的遥测系统，或是逼着 AI 队友去适应人类的用户体验，只会处处掣肘，把自主性带来的那点好处全给抵消了。

眼下大多数**智能体工具链**，设计上还停留在“一人一机”的老思路上。人们正快速转向“一人指挥多机”的模式，却因为工具没跟上而苦不堪言。没有统一的指挥界面，人类就得同时对付终端、聊天日志和一堆零散状态，协调工作立马变成了认知过载。更糟的是，人类不得不干等着 AI 队友把活儿干完。随着 AI 队友处理的任务越来越庞大，你最宝贵的资源（人类）反而被晾在一边；这本质上就是**让人给流程当观众**，限制了他们。因此，工作台必须天生就提供舰队级的全局视图和控制能力，实现跨工作流的无缝多任务处理。

AI 队友同样需要一些基本保障：能干活的手、能观察的眼，以及一条安全的跑道。如果一个队友还得求着人类帮忙粘贴日志、描述错误或执行命令，那这系统就谈不上真正的自主。AI 队友的工作台必须提供对日志、追踪、指标、仓库状态、构建输出和测试结果的深度集成访问，还得有一

个供探索调试之用的安全计算沙箱。未来的默认模式，应是无需人工干预的 AI 队友操作：你可以命令一个队友“干不完别停”，而它所处的环境足够强大，完全不需要人类持续地“打补丁”。



管理学有个经典原则：给手下配好工具，然后放手让他们去干。到了智能体时代，“好工具”意味着快速、易上手、文档齐全，让使用者能独立操作，不用事事求人。它还意味着安全得用。默认安全的工具链，能减轻下游的审查负担、降低事故率，因为最顺手的路径就是最安全的路径。主流工程界早有先例。Rust 消灭了 C 和 C++ 里常见的一整类内存安全错误，从而让工作重心从“事后找补丁”转移到了“事先写对代码”。同样关键的是，Rust 的工具链异常友好：编译器报错信息详尽、可操作，还常附具体建议。这一点很重要，因为任何能帮到新手开发者的东西，同样能帮到 AI 队友。两者都受益于紧密、高信息量的反馈循环，而对 AI 队友而言，这种循环的收益会呈指数级放大，因为它们能在无数并行任务中不知疲倦地迭代。TypeScript 则通过强化类型系统，让许多在 JavaScript 里才会在运行时暴露的错误，提前到了编译阶段，从而将工作重心从“运行时调试”转向了“编译时反馈”。平台的启示并非“该用某某语言”，而是：以安全为导向的默认设置和强大的反馈循环，能带来复合收益。在一个人类能指挥多个 AI 队友的世界里，哪怕一丁点的工具摩擦或不安全的默认设置，都会在数十甚至数百个并行循环中被急剧放大。

7.2 核心理念：将人类工作台与 AI 队友工作台分离

双工作台模型最清晰地指明了方向：一个是人类发号施令的环境，另一个是 AI 队友埋头执行的环境。工作台不仅仅是好用的工具，它们是维持人类不被压垮、同时让 AI 队友保持高效运转的基石。如果你的组织还把聊天日志当主要界面，那扩展这场仗还没打就输了。我们的目标，是建立一个以“具有持久链接的数据包”为默认协作单元的工作台，而非那些转瞬即逝的对话片段。

人类命令环境是决策诞生的地方。它需要一个用于接收咨询和就绪包的收件箱、用于撰写任务简报和指令的创作支持、AI 队友管理、架构可视化，以及多套方案的并排比较功能。它还必须支持 N 版本探索，因为在团队规模下，比较多份计划与证据包已成为常规决策手段。关键在于，人类命令环境应支持流水线创作：设计多智能体工作流，让任务通过结构化的交接与审批关卡，从一个 AI 队友流向另一个。这正是人类从“手动编排”（要求每个智能体执行每个步骤）扩展到“自主执行”（一次性设计好工作流并任其运行）的途径。当命令环境设计精良时，人类的时间会花在判断、排定优先级以及工作流的设计创作上，而不是耗费在寻找上下文、协调会议或手动在智能体之间派发任务上。

人类命令环境必须为一件事而优化：最大化人类稀缺时间的有效利用率。既然人类审查能力是瓶颈，那么平台的责任就是压缩信任决策的过程。这需要四项具体能力。

- 其一，每个审查界面都必须以差异对比为核心，清晰标出自上次人工介入以来，代码、简报、计划及证据发生了哪些变化。
- 其二，它必须支持将计划审查作为一等公民的工作流，通过比较概念性计划与最终执行计划，高亮差异并将每个差异链接到相应的理由和证据。
- 其三，它必须强制实施渐进式披露，让人类能先浏览决策就绪的摘要，只在感觉不对劲时才深入查看细节。
- 其四，它必须支持精准、可定位的反馈：针对特定声明、计划步骤、代码差异或证据项进行内联评论，并将 AI 队友的回复与之链接，确切展示因此引发的改动。

还有一种“边做边演示”的模式，平台必须视其为常态，而非异常。有时，通过在 IDE 里做些精准的小修改来传达决策，比再写一段反馈要快得多。因此，人类工作台应支持无缝切换到传统的 IDE 和分支工作流，并将此操作作为一等事件记录下来：改了哪里、为何而改，以及它如何更新了执

行计划。关键在于，人类能在成本最低时直接动手，而平台仍能保留完整的可追溯性，让整个舰队能从这次干预中学习，而不是将其视为一次隐形的“英雄救场”。

AI 队友执行环境是任务落地的地方。它需要能为 AI 队友提供结构化反馈的原生工具：编译器、测试框架、**代码检查工具**、静态分析器、安全扫描器、基准测试链和属性检查器，所有这些都应作为学习信号来呈现。它需要密闭的执行环境，以确保结果可复现、可审计；还需要具备健康监控的并行能力，让整个舰队能顺畅运行，而不是变成一团谜。

它还需要资源和成本意识，但要贴合企业计算的实际情况。过去，增加吞吐量需要人力规划。现在，任何工程师都能在几分钟内召唤数十个 AI 队友。有时，如果业务价值明确且你的机器闲置，这种爆发式调用正是明智之举。但有时，如果它占用了企业内共享的稀缺资源、将故障隐藏在噪音之下，或是将并行性变成了无声的排队债务，那就是浪费甚至危险的。平台不应简单地默认“禁止”。它应默认提供可见、受管控的弹性：显示成本、显示资源争用、在合理时允许爆发、强制执行限制，并能在计算资源紧张时，将任务重新安排到更合适的时间。

7.3 工作台工程的关键实践

7.3.1 实践一：定义“铺好的路”，让数据包成为一等公民

首先让协作的基础变得可计算：定义一组小型的数据包类型，它们拥有稳定的标识符、所有权字段、证据链接和生命周期状态。然后构建使这些数据包得以实现的平台原语：索引、搜索、路由、访问控制，以及代码、决策和证据之间的持久化链接。这正是从“聊天加 git”迈向“智能体工作的权威记录系统”的关键一步。如果数据包无法被路由和查询，你就别想突破一对一协作的局限。

7.3.2 实践二：为“一人对多机”构建人类指挥平面

为人类提供舰队视图和差异视图：有哪些任务、哪些正在进行、哪些被阻塞、自上次审查以来发生了什么变化，以及哪些需要决策。人类应该能一眼看到队友们此刻在做什么、下一步计划做什么、以及至今产生了什么证据，而无需同时打开五个终端窗口。指挥平面应将“自我解释”作为一项功能：当情况看起来不对劲时，人类可以要求一个简明的“**现状、意图及后续风险**”数据包。这正是将人类注意力聚焦于判断、而非监控的方法。

7.3.3 实践三：通过差异优先审查、计划台账和精准反馈来压缩信任决策

指挥平面必须让审查既快又不草率，而唯一的办法就是审查差异，而非最终成品。人类审查的每个工件，都必须有一个一等公民的“自上次审查以来的变化”视图：代码差异、任务简报修订、就绪包更新、政策与指导调整，以及证据清单。系统应默认 AI 队友会快速迭代，因此必须让迭代审查的成本变得低廉。

通过引入计划台账，让计划审查变得明确。台账始终包含四个字段：概念性计划、执行计划、差异映射和理由链接。差异映射应清晰展示执行过程在哪里偏离了原始概念，每个差异都应链接到触发它的证据或约束，以及接受它的决议或批准。这是一个人类时代不曾存在的新审查界面：我们现在不仅可以审计产出了什么，还能审计它是如何产生的、以及判断是在何时发生了转变。

渐进式披露必须通过结构来强制实施，而非依赖良好的意愿。每个审查数据包的顶层，都应是决策就绪的：请求的决策、风险等级、关键证据结果，以及允许做出判断的最小摘

要。详细的日志、追踪记录和冗长的差异对比，应置于深入链接之后，可索引、可搜索，而不是一股脑堆在首页。

最后，反馈必须是精准且可追溯的。评论应作为可寻址的对象，锚定到具体的计划步骤、声明、差异区块或证据项，并带有状态和决议记录。AI 队友的回复必须链接回确切的评论，并显示由此引发的具体改动：哪个计划步骤变了、添加了哪些证据、修改了哪些代码，或是升级了哪些风险。如果反馈无法追溯到具体变化，它就会沦为叙述，而叙述是无法扩展的。

7.3.4 实践四：为快速、自给自足的工作构建 AI 队友执行平面

AI 队友应该能运行完整的闭环：获取上下文、编辑代码、运行测试、检查故障、收集日志，并迭代直到任务完成。这需要密闭的沙箱、可复现的工具链，以及对日志、追踪和指标等可观测性信号的安全访问。它还需要并行环境，以便探索性尝试不会污染正式执行，并且能快速测试多个假设。当 AI 队友拥有了完整的手和眼，那种需要人类复制粘贴的循环就会消失，自主性才变得真实。正确的心理意象不是只有一个助手的双手，而是一个九头蛇：许多双手和许多眼睛在并行运作，而这只有在平台提供了安全、隔离、深度集成的工作空间时才可能实现。

7.3.5 实践五：自动化证据捕获以降低信任成本

证据捕获的目标不是文档本身，而是为了扩展信任决策的规模。对工作台进行深度集成，以便自动捕获工具输出、环境版本、**依赖项快照**和关键追踪数据，并将其链接到就绪包中。来源必须明确，因为非确定性会让“信任叙述”变成一种脆弱的习惯。当证据实现自动化且结构化时，审查者就能快速决策，并仅在感觉不妙时才深入探究。

7.3.6 实践六：从设计上就让 AI 队友执行工作台安全第一

工作台是权限与行动交汇之处，因此护栏必须建在这里，而不仅仅是写在政策文件里。工具访问应遵循最小权限和任务范围原则，密钥应通过受控渠道处理，高风险操作应触发结构化的审批流程。安全的执行环境不仅关乎安全本身；它也是防止高速自动化造成意外损害的手段。如果工作台无法强制执行安全，人类就会退回到手动审批，系统也就无法扩展。

7.3.7 实践七：像运营生产平台一样运营工作台

一旦平台建立起来，它就成了生产系统的一部分，必须像其他任何生产系统一样被监控和改进。追踪摩擦信号：人类仍在哪些环节干预、AI 队友在哪些地方卡住、证据在哪些地方缺失。监控舰队健康状况：重复出现的故障，通常意味着共享的错误配置，而非个别队友的失误。按节奏进行改进，因为天天都在变的平台，本身就是新的不稳定源。

7.3.8 实践八：为企业级舰队可观测性和资源控制增设企业指挥中心

在企业规模下，你需要一个能快速回答三个问题的指挥中心视图：舰队在做什么、在消耗什么资源，以及在哪些方面以相关的方式失败了。这不是一个虚荣的仪表板，而是一个控制界面，让你能分配稀缺资源、检测共享的故障集群，并防止“无限并行”演变为无声的资源争用。就像 SRE 团队使用服务级别遥测来运营生产系统一样，平台团队也必须使用舰队级别遥测来运营智能体工作台。

7.4 工作台工程模式

7.4.1 模式：工作台分离

别强迫 AI 队友挤进人类的用户体验，也别逼着人类钻进 AI 队友的遥测系统。人类需要就绪摘要、决策提示和可供审计的深入查看入口；AI 队友则需要工具 API、结构化反馈和执行稳定性。分离减少了意外的耦合：改进 AI 队友的循环，不会破坏人类的决策循环，反之亦然。这是杠杆率最高的举措之一，因为它直接保护了人类最宝贵的注意力。

7.4.2 模式：收件箱，而非中断

咨询请求应以数据包形式送达收件箱，而不是作为劫持注意力的同步中断。收件箱应支持按角色和风险等级进行路由，确保合适的人在合适的时间看到合适的咨询。这将推动组织从“会议绑定的决策”转向“异步、持久的决策”。一个实用的领先指标是会议时间：如果会议负担没有下降，那就说明系统尚未真正实现数据包驱动。

7.4.3 模式：将 N 版本比较作为一等公民

工作台得能生成、比较并筛选多个解决方案候选。界面不能只展示代码差异，更得把计划、风险和证据的差别亮出来。这在人类工作流里成本太高，办起来费劲。但有了 AI 队友，这活儿就便宜了，所以工作台必须把这功能做得顺手好用。

7.4.4 模式：对所有工件进行差异优先审查

审查能力是稀缺资源，因此平台必须把“有啥变化”设为各处默认视图。人类不必从头重读任务简报、就绪包或策略包；他们只需审查上次批准后的改动。差异优先审查还能减少扯皮，因为它逼着讨论聚焦具体变化，而不是模糊感觉。

在舰队规模下，不用差异优先？代价就是反复重读、反复争吵，外加悄无声息的跑偏。

7.4.5 模式：计划台账审查与偏离映射

把概念性计划和执行计划当作两个默认就得可比较的关联工件。工作台得显示执行跑偏的位置、为啥跑偏，以及跑偏是否已获批准或该升级处理。这样一来，“审计走过的路”就成了常规操作，而非调查活动，人类也能专注在少数真正重要的偏差上。一旦偏离审查变得容易，自主权就能安全扩大，因为平台让偏差一目了然，还能拿来讨论。

7.4.6 模式：可寻址的内联评论与关联的 AI 队友响应

反馈必须锚定在特定对象上：计划步骤、声明、差异块或证据项，而且得能解决。工作台得把评论当一等公民看待，给它身份、状态，还能链接到决议记录。AI 队友的响应得链接回每条评论，并且必须展示响应带来的具体改动。这招能防止“模糊反馈”和“讲故事式回复”成为默认的协调基础。

7.4.7 模式：可追溯意图的下拉式干预

有时候，最快的反馈就是一次精准的外科手术式编辑。工作台应支持一种受控的下拉路径，让人能跳进 IDE，动动手术刀，然后把改动自动同步回工件系统：差异链接上，意图捕获到，执行计划更新以反映这次干预。这防止了“边做边秀”变成破坏学习和可审计性的暗箱操作。当直接干预比解释更划算时，人类就该能直接上手，同时平台确保干预变成持久指导，而不是一次性补丁。

7.4.8 模式：以无干预执行为设计目标

设计 AI 队友环境时，要让它能完成任务，无需人类干复制日志、重跑命令这类打杂活儿。这意味着可观测性和调试能力得是 AI 队友工具链的内置部分，而不是外挂的便利设施。这也意味着环境得支持受控探索：多个沙箱、清晰的来源和安全的回滚。当无干预执行成为常态，一个人就能安全管理一大堆 AI 队友。

7.5 工作台工程反模式

7.5.1 反模式：聊天即 IDE

所有东西都堆在聊天日志里？那团队就别想好好审查、搜索或持久化了。聊天鼓励讲故事，而不是摆证据，处理高吞吐量时，这完全是本末倒置。解决办法是：把协作挪到版本化工件里，聊天只当临时试验场，别当成权威记录系统。工件成了主要载体，交接才实在，审计才有可能。

7.5.2 反模式：叙事式审查与模糊反馈

人类审查时长篇大论，反馈又模糊不清，系统就没了收敛的能力。AI 队友用更多故事回应，人类重读更多上下文，审查时间越拖越长，直到又成瓶颈。就算有工件，这种故障也可能发生，因为工作台从没强制反馈必须锚定和可追溯。解决办法是：差异优先审查加上可寻址的内联评论，让每条人类注释都对应特定更改，每个 AI 队友响应都通过明确反馈链接回来。

7.5.3 反模式：工具匮乏与人类复制粘贴循环

AI 队友被脆弱脚本束缚，产出也脆弱，还会浪费人类时间，因为证据要么缺失，要么没说服力。复制粘贴循环是工具匮乏的最明显迹象：如果人类老得提供日志、指标或命令

输出，那工作台就没干好核心工作。解决办法是：搞个标准的 AI 队友执行工具链，默认带上可观测性、隔离运行和证据捕获能力。工具链不行，队友再聪明，自主权也不安全。

7.5.4 反模式：无成本和健康控制的无限并行

如果平台让生成 50 个 AI 队友变得轻松，团队肯定这么干，成本和故障模式的增长会比吞吐量还快。没有舰队健康监控，重复故障看起来就像随机抽风，而不是共性问题。没有成本预算，并行化就成了隐形的运营债务。解决办法是：在工作台里内置明确的资源限制、可见性和优先级排序，加上弹性规则，有理由时可以爆发一下，系统负载高时就约束起来。

7.5.5 反模式：默认不安全的执行环境和工具链

最方便的路如果不安全，舰队就会选它，组织就得反复买单。这就像工作台版的 C 语言对 Rust、JavaScript 对 TypeScript 的教训：不安全的默认设置会把正确性成本推到下游，丢给调试、审查和事件响应去扛。在智能体软件工程系统里，复合效应是残酷的。今天让一个开发者烦心的工具，明天可能拖慢几百个 AI 队友的循环。默认允许不安全操作的工具体系，能以机器速度复制错误。解决办法是：安全第一的默认设置、强大的静态和动态反馈，还有由平台强制执行的特权边界，而不是靠提醒文本。

7.6 主要构建模块

工作台承载工件，但也强制规定了形态和流程。

- **人类命令工作台** 围绕咨询包、就绪包、任务简报编写、N 版本比较和舰队状态展开。

- **AI 队友执行工作台** 则围绕隔离运行、结构化反馈、证据捕获和舰队遥测。交付物不是仪表板，而是一个工作流，其中每个重要决策都是一个带着持久证据的包。

7.7 衡量工作台工程

7.7.1 指标：人类干预率

跟踪人类得为 AI 队友干机械活的频率：粘贴日志、重跑测试、提供系统输出或解释环境状态。这是衡量 AI 队友有没有“手”和“眼”的直接指标。如果干预率居高不下，自主权就还是贵，人类就成了瓶颈。干预率下降是平台正常工作的最明显迹象之一。

7.7.2 指标：咨询包决策时间

衡量从咨询包到达，到做出决策的时间，以及从就绪包提交，到批准或拒绝的时间。这不是“合并时间”，而是在给定包的情况下，做出信任决策的时间。如果决策时间随吞吐量增长，那说明工作台没压缩好复杂性。渐进式披露应该让决策时间保持有界，哪怕产出在增加。

7.7.3 指标：会议负载趋势

跟踪每位工程师每周的会议小时数，尤其是那些主要为了协调，而不是设计讨论的会议。在包驱动的系统里，很多协调会议应该被异步咨询包和版本化决议取代。会议时间不会降到零，但随着工作台成熟，它应该显著下降。如果没有下降，说明组织还在用会议当协调基础。

7.7.4 指标：批准的再现率

定义如下：即使生成的代码不完全相同，独立的执行者能否重新运行批准关卡，并重现关键证据。这是非确定性系统中

的关键区别：你不是要重新创建完全相同的输出，而是要在相同约束下，重新创建能通过相同治理检查的等效结果。通过在隔离环境中抽查审计或自动重放来衡量。低再现率会把证据变成表演，破坏信任。高再现率则能把非确定性协作变成可审计的工程。

7.8 总结

工作台工程是一门学科，它通过为每种模式提供优化环境，让人类与 AI 的协作可扩展。通过分离人类命令环境和 AI 执行环境，实施基于包的异步协调，启用差异优先审查，并自动化证据捕获，我们创建了一个系统，让一个人能有效协调许多 AI 队友，而不会认知过载。

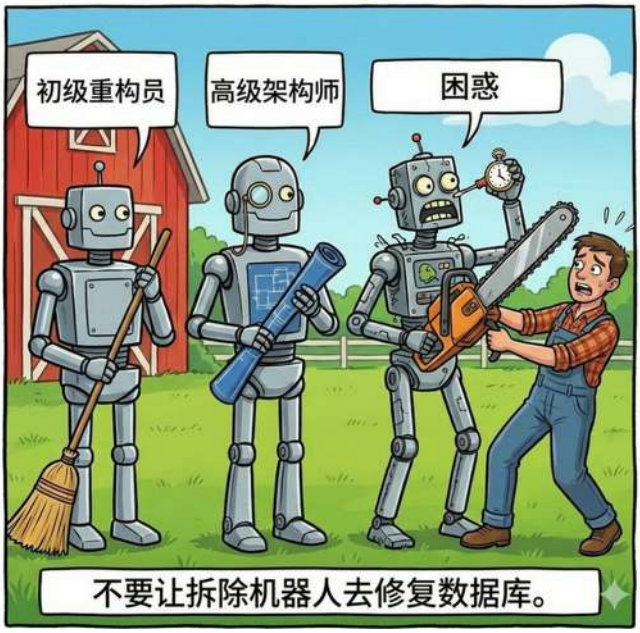
本章的模式和实践不是可选的便利设施，而是扩展到舰队运营时必不可少的基础设施。从包原语和收件箱工作流开始。增加舰队可见性和差异优先审查。构建无干预执行能力。目标不是第一天就有完美平台，而是进行系统性改进，防止界面成为限制你智能体转型的瓶颈。

8 能力工程：角色、资质与持续改进

8.1 根本张力

你希望 AI 队友能像真队友一样：独立高效、懂得权衡利弊、还能搞定复杂活儿。但能力若缺乏结构，只会乱成一锅粥。组织得摸清每个队友的底细：能干啥、干得咋样、能力是涨是跌。这种矛盾之所以尖锐，在于能力并非**整齐划**一；AI 队友的能力剖面参差不齐，不同角色需要不同的技能组合。能力工程就是为了把这些剖面整明白、可测试、能认证、还能持续改进。它要回答：“这位队友能干啥？咱能证明不?”至于“凭他们这本事，该让他们干啥、在啥条件下干?”那就是信任工程的活儿了。

这门学科得面对一个扎心的事实：AI 队友的能力天生就参差不齐。就像人也有高低之分——有人擅长大局谋划，有人精于细节执行——AI 队友基于背后模型，也有自己的长短处剖面。



有些模型，甚至同一模型的不同配置，在某些任务族上表现亮眼，在其他方面却拉胯——这种参差性的形态，比单纯的智商高低更重要。团队规模一上来，不匹配的问题就被放大了：**用错人**的活儿会在并行任务中成倍增加返工、升级和审查的负担。能力工程把模型-任务匹配当成工程输入来处理，而不是**凭感觉派活**。

还有个导师稀缺的老大难问题：顶尖的人类导师能把普通工程师点石成金，但大多数组织没那么多宝贝导师可分配。到了**智能体软件工程**时代，“让大伙儿都像那 1%的顶尖人才一样干活”成了系统级目标，因为你可以在全组织复制高质量队友的配置。真正的难点在于认清现实：AI 队友如今是你生产平台的一部分，不再是可选的工具。

8.2 核心概念：能动性、能力与记忆基底

能动性就是执行目标、落实计划的本事。工作台工具能加速执行，但没能力的能动性，就像飙车没刹车，危险得很。能力工程专注能力这块儿：确保队友具备干活儿所需的技能、知识和行为模式。定义队友能自主决定啥、需要请示啥

的操作边界，就是能力边界的工件；至于如何基于信任校准这边界，信任工程章节再细说。

指导方针把部落知识编码成可加载、可测试的指导，从而塑造能力。实际结构是原则加规程：原则像稳定的指南针，构成心智模型；规程则是针对特定情况、触发即执行的标准操作程序。重点不是去掉判断，而是让判断基于明确指导，从而保持一致、可审查。

在舰队规模上，记忆（经验）不只是锦上添花，更是 AI 队友腾飞的基石。你需要任务相关的情景记忆、能复用为模式和清单的任务记忆，还有记录决策和约定的团队记忆。这记忆得能从代码库直接挖出来，这样任何在目录里干活的队友都能自动加载相关规则手册和本地上下文。当这一切跑顺了，知识和最佳实践就能在组织里传给所有 AI 队友，这才叫真正的复利优势。

8.3 作为平台工程与人员运营的能力工程

平台工程的存在，是因为组织明白：给团队铺好路，生产力才能规模化；要是每个团队都自己造轮子，那得乱套。实践中，平台团队提供自助服务、合理默认值和护栏，让产品团队能快速前进，不用重复造轮子。能力工程就是协作者的平台工程：给队友定义一条认知和决策的铺好路，不只是执行。重点不是集中控管，而是确保能力一致、质量稳定。

所以，能力工程开始有点像给非人类工作者舰队搞人力资源。相关举措不是激励，而是搭结构：角色定义、能力矩阵、训练循环、绩效校准、晋升和 360 度反馈。工作分析是头一步：活儿没派之前，先定义角色干啥、需要啥能力，毕竟“聪明的通才”不是计划。随着能力实打实地提升，队友就能候选更广的职责；至于授权决策，交给信任工程管。

8.4 能力工程中的关键实践

8.4.1 实践 1：能力校准与角色分配

首先把 AI 队友当成你在学习如何部署的专家，别当成啥都能干的万金油。针对任务族校准能力：比如调试、重构、绿地项目搭架子、性能调优、安全敏感改动、迁移规划和文档编写。这正是参差智能风险能落地操作的地方：一个队友可能审查厉害、设计拉胯，或者实现牛掰、集成不行。角色分配基于实打实的证据和认证，而不是氛围印象，这才靠谱。

这实践也逼着很多团队直面一个决定：每个 AI 队友都得有个角色。角色缩小了接触面，升级路径也明确了，认证才有意义，因为“具备 X 资格”变得可测试。定制化得保持可认证，就是说定制版本还能通过该角色的基线资质测试套件。要是定制化没法测试，最终只会漂移走样。

8.4.2 实践 2：作为代码的指导，具有结构和层次

把部落知识编码成队友能可靠加载、应用的指导方针。原则要少而稳，它们构成心智模型，能跨任务泛化，不会冲突过载。规程要具体、触发即绑定，这样能防止把有说服力但跑偏的实现当成进展。指导要分层：公司铁律、部门规矩、团队惯例、项目规则，每层都有明确优先级和废弃规则。

这活儿的一部分是保持队友**该有的复杂度，别过火**。重复、太细或间接冲突的指导会被忽略或误用，团队就会把不一致误诊为“模型不可预测性”。平台得让指导重构成常态：消冗余、解冲突、废过时规则，保持活跃工作集足够短，好加载。

偏好得明确处理，因为是可选的，但不一致时成本就高了。偏好可以因正当理由打破，但得有个稳定默认值，这样输出可比、集成成本低。要是每个队友对“偏好”理解不同，

人类审查和重构就得付出代价，组织也丢了可预测性。目标是有可预测的差异，不是僵化遵从。

8.4.2.1 指导机制谱系

教队友可以用概率性指导、确定性脚本或两者结合。机制选择决定了组织能依赖哪些保证，团队常图方便而非适用性来选。

执行机制范围从完全概率性到完全确定性：

- **概率性指导**是加载到队友上下文的自然语言指令。队友解释指导，可能听也可能不听。遵守是统计性的，没保证，因为执行基底是**随机性的**。就算用上强调字眼（“必须”、“总是”、“绝不”），也改不了解释看概率的本质。
- **确定性脚本**是照原样执行的代码：钩子、CI 门控、代码检查器、自动化检查。遵守由构造保证，因为这机制不靠解释。
- **混合机制**两者结合。一个确定性脚本可能在特定点调用概率性推理：确定性地跑代码检查器，然后让 AI 队友概率性地修问题。概率性指导也可能触发确定性验证：队友推理出方案，然后确定性测试套件来验结果。确定性脚本本身也可以掺概率性元素：一个确定性地调用 AI 队友生成代码，再确定性地对输出跑测试的工作流。保证取决于哪些部分是确定性的，哪些是概率性的。

触发机制也范围从概率性到确定性：

- **概率性触发**意味着 AI 队友根据上下文解释来决定啥时候用指导。队友可能没认出某种情况需要该指导，或者觉得指导在当前上下文中不适用。这引入了第二层不确定性：即使该触发，指导也可能不触发。
- **确定性触发**意味着代码或平台逻辑决定啥时候调用指导或脚本。文件模式匹配、git 钩子、流水线阶段和显式工

具调用都是确定性触发器。触发条件一满足，指导或脚本就执行，不管队友怎么想。

这就形成一个二维空间。概率性指导加概率性触发，保证最弱：调用和遵守都难保。确定性脚本加确定性触发，保证最强：调用和遵守都由构造保证。实际指导大多在两者之间，团队得琢磨到底需要啥保证。

关键洞见是，三种方法——概率性指导、确定性脚本和混合机制——都能有效教 AI 队友。问题不是抽象上哪个“更好”，而是哪种适合特定意图。行为偏好能容忍概率性遵从。强制性规程可不行。机制必须匹配意图。

8.4.2.2 使机制与意图匹配

教 AI 队友可以用概率性指导、确定性脚本或混合机制。不同类型的组织知识需要不同机制。机制与意图不匹配，就会导致虚假信心和潜力受限。

行为偏好是可选的、不遵从也能忍的最佳实践。

- 示例：“优先使用组合而非继承。”“用语义化提交消息。”“给复杂函数加描述性注释。”
- 适当机制：概率性指导，配合概率性或确定性触发。
- 原因：队友要是 80%的时间照做，组织就赚了。偶尔跑偏也能接受，风险不高。概率性指导在这里很高效，不用为每个偏好都搭强制执行的基建。

强制性规程是必须遵守的标准操作程序。

- 示例：“部署生产前必须跑安全扫描。”“所有 API 端点都得验认证令牌。”“数据库迁移执行前必须过审。”
- 适当机制：确定性脚本配合确定性触发。如果强制性步骤本身是确定性的，混合机制也可接受；概率性元素可以放在工作流别处。

- 原因：随机性执行基底没法保证概率性指导的遵守。强制性规程需要强制执行，不是指导。如果这步非走不可，那就得用绕不过去的代码来强制执行。

认知策略描述怎么推理问题。

- 示例："调试时，先复现，再隔离，再追踪。""设计 API，要从消费者角度想。""分析安全，先枚举攻击面，再审查代码。"
- 适当机制：无。别把认知策略编码成指导或脚本。
- 原因：规定怎么思考会限制 AI 队友的推理潜力。定义目标和约束；让 AI 队友自由推理方法。这就是**软件工程 2.0**（自动化现有流程）和**软件工程 3.0**（委派结果并信任队友推理）的区别。

关键洞见是，概率性指导只适用于行为偏好，因为不遵从本就可容忍。把强制性规程编码为指导的团队，安全感是假的。把认知策略——无论是指导还是脚本——编码的团队，从队友那儿获得的收益会打折扣。

8.4.2.3 为什么不应编码认知策略

2019 年，理查德 · 萨顿（Richard Sutton，强化学习奠基人）发表《苦涩的教训》，指出 AI 研究 70 年来的核心启示：通用计算手段永远比手工编码的领域知识更胜一筹。萨顿观察到，研究者总想将知识塞进系统，短期虽有小利，最终却作茧自缚，阻碍突破。真正的飞跃来自反向操作：靠搜索和学习来扩展算力。这教训为何"苦涩"？因为它意味着，我们的那点小聪明一旦固化，反而成了绊脚石。

这道理放在 AI 队友身上一样管用。当你把认知策略写成**指导包**或死板脚本，就相当于对一个强大的推理引擎指手画脚，教它怎么思考。你是在用自己那套方法，取代模型天生的推理本事。你是在捆住一个可能另辟蹊径的高手，硬逼它走你熟悉的老路。

拿漏洞发现来说吧。2026 年初，研究人员把一位 AI 队友放进虚拟机，让它能访问开源项目和标准安全工具：调试器、模糊测试工具这些家常便饭。关键一步是，他们没给任何找漏洞的特别指示。没方法论，没指导手册，故意让队友自由发挥，自己琢磨办法。

结果吓人一跳。这位 AI 队友在常用开源库里揪出 500 多个未知高危漏洞。但更有启发的不是漏洞数量，而是它怎么找到的。

队友先试了经典套路：跑模糊测试。这招人类安全研究员常用，也是“漏洞搜寻方法论”的老生常谈。但模糊测试没啥收获。

接着，队友出了奇招：它去翻 Git 提交历史。它发现一个关于栈边界检查的安全提交，推敲修复前的代码模样，揪出一个没补完全的漏洞。另一个案例里，队友识别出堆缓冲区溢出漏洞，这需要理解 LZW 压缩算法和 GIF 文件格式的关系——传统模糊测试工具根本触发不了，因为需要特定操作序列，只有深入算法推理才能搞出来。

没人写指令叫队友挖提交历史找半吊子修复，也没人让它从第一性原理推导压缩算法的边界情况。队友是在老办法失灵后，自己搜索推理，憋出了新招。

试想，如果研究人员写一份“安全研究方法论”，规定：“第一步：跑模糊测试；第二步：做静态分析；第三步：查常见漏洞模式。”队友肯定会照本宣科。它或许能发现几个漏洞，但绝不可能发明那些带来突破的新花样。规定的方法论会把队友框死在人类的思维定式里，白白浪费潜力。

这不是说指导对复杂工作没用，而是说指导该定义成功模样，而非具体实现路径。给出目标、质量标准、约束和验证机制；投资能让队友自我探索和验证的工具。然后，放手让队友自己推理方法。这正是软件工程 3.0 与软件工程 2.0 的区别：只管结果，别管过程。

说白了：如果你正写指导或脚本，描述“如何分析”、“如何调试”、“如何设计”或任何需要创造性推理的任务，赶紧打住。你在画地为牢。把指导重新定义为目标和约束，然后让队友自己去摸索。

8.4.2.4 合规性差距

概率性指导在感知可信度和实际可信度之间挖了个坑。把强制程序写成指导的团队，以为已经**搞定要求**，其实不然。他们只是表达了意图，底层随机性可能听话，也可能不听话。

这坑有两层：第一，概率性触发可能失灵——队友可能认不出需要指导的情况，或觉得指导不适用当前场景；第二，即使触发，概率性执行也可能掉链子——队友对指导的理解可能跑偏，跳过自认多余的步骤，或在指导没料到的情况下自由发挥。

结果呢？团队产生了脱离现实的安全感（“咱把安全要求写成指导啦”）。组织自以为合规，实则不然。这是最危险的失效模式，因为出事之前，它根本看不见。

解决方法得靠结构，不是嘴皮子。对于行为偏好，接受合规性是统计性的，并据此设计。对于强制程序，把要求从概率性指导里拎出来，塞进**确定性强制执行**。如果重要程序非用概率性指导不可，那就搭建合规性验证基础设施：独立系统，专门确认程序真执行了、出真结果了。设计工作流程时，干脆假设概率性合规永远到不了 100%。具体怎么填坑，参见信任工程章节的合规性验证架构。

8.4.3 实践 3：操作边界与升级矩阵

把操作边界定义成具体矩阵：决策类型、允许的负责人、必需咨询路径、必需证据。这样，升级就从“队友抓瞎”变成了设计好的结构化中断，人类也不用被鸡毛蒜皮缠身。边界还得包括反目标：明确定义“绝不允许”的结果，比如安

全倒退、没有迁移计划就破坏模式、或者违反策略。反目标是廉价护栏，专治那种打着改进旗号的范围漂移。

这做法也点明一个企业现实：缩小队友职责范围永远可以，但运营上可能受不了。如果每个决策环节都要人插手，系统就别想扩展，经济性会在注意力重压下崩盘。出路不是“胆子大点”，而是“拿出过硬能力证据，让组织有信心扩大权限”。角色边界清晰了，组织才能理性辩论，而不是拍脑袋。

怎么根据风险、可逆性和信任证据校准操作边界，信任工程章节有细说，那里把委托当成治理纪律，而非能力工件。

8.4.4 实践 4：资格考试与持续认证

在扩大队友权限前，先让它通过特定角色的资格考试，证明它在你的环境里真有本事。关键是认证按角色来。考试不是一锤子买卖，而是一串场景，专门考察角色各方面：正确性、证据生成、升级行为、策略遵守和**集成规范性**。通过认证套件，就说明队友有资格干这活；更高风险的权限作为基础资格上的认可层处理，而不是另搞一个“验证”类别。这样系统简单：一个概念——资格认证，分多级。

在舰队规模上，认证必须持续进行。新基础模型版本、新工具链或指导更新，都可能导致倒退，不重测的话，等生产出问题才发现就晚了。所以认证套件得在 AI 队友每次重大变更时运行，不能只入职时搞一次。结果是一份认证记录，让“具备此角色资格”成为有凭有据的说法。因为这角色其实是管理生产协作者，不能当成新手活。负责认证套件和晋升标准的人，得在指导和早期识别失效模式上经验老道，因为校准不准会波及整个组织。

这就是 AI 队友和人类队友必须区别对待的地方。人类队友能力退化慢，演变可预测；AI 队友的“脑子”可能因为模型更新或工具链迁移，一夜之间就换了。这带来了重新认

证的要求：底层模型（哪怕是小版本升级）、检索逻辑或工具定义的任何改动，都必须触发认证套件完整重跑。

如果你不系统推出这些更新，就等于对生产劳动力做脑叶切除术或激进增强，还没验证结果。系统推出意味着把模型更新当成重大基础设施迁移。你不会不跑回归测试就换数据库引擎；同样，你也不能不重新认证就换推理引擎。这更说明能力工程不能是功能开发人员的“副业”；它需要一套纪律严明的流程和专人，来管理这些非人类协作者的生命周期。

8.4.5 实践 5：晋升与恰当的拒绝

当队友通过资格认证证明能力后，就能候选承担更广职责。晋升要正式、有证据，系统记录，理由明确。渐进式委托的详细机制，包括怎么根据风险和信任证据校准自主权，信任工程章节会细说。

这里得强调一个能力要求：AI 队友必须敢说“不”。队友应该拒绝那些超出其校准能力范围的任务，并向上汇报，而不是“硬着头皮上”即兴发挥。平台必须奖励这种恰当的拒绝，因为任务超纲时，拒绝才是正确操作。

8.4.6 实践 6：反馈驱动的改进循环与自我改进训练场

舰队只有把学习成果抓取并反馈到塑造未来行为的**工件**里，才能进步。回顾总结该更新指导方针、操作边界、运行手册和就绪模板这些**工件**，而不是搞一堆没人看的“经验教训”文档。更新别按任务来，那会乱套；得按预定节奏来，并由指定负责该队友的人类把关。这个负责人角色很重要，因为“谁管这个队友”必须像“谁负责这项服务”一样清楚。

自我改进训练场是把学习变成可重复能力、而非零碎轶事的方法。训练场是个受控环境，AI 队友在里面练习任务相

关技能，获得结构化反馈，并提出具体指导方针或工具改动建议，由人类批准落地。原材料是交互痕迹和结果数据，所以这些痕迹得像生产遥测一样记录管理。这也是受控自我探索的关键：AI 队友应该能调查实验，学习 API 或工作流程，但学习成果必须通过结构化渠道落地，让舰队能一致改进。

8.5 能力工程模式

8.5.1 模式：升级是一种功能，而非失败

设计咨询路径，让 AI 队友能求助，又不至于对人类狂轰滥炸非结构化问题。好的升级该以决策就绪包的形式出现：决策、选项、权衡、建议，加上已收集的证据。这种结构节省人类时间，避免“重头再看”的失效模式。升级设计得好，队友效率更高，因为它清楚何时提问、怎么提问。

8.5.2 模式：专业化队友，然后认证专业化

用角色来约束作用范围，让专精成为可能，别指望全能。一个专业评审员队友的培训、衡量和认证，就该和迁移规划师或性能调优师不同。专业化也让委托更简单，因为职责能映射到角色要求。在真实组织里，组合拳比“一个超级队友”的神话更靠谱。

8.5.3 模式：指导方针和操作边界是代码，值得进行变更管理

把这两者都当成代码，值得做变更管理。这意味着每次改动都要代码审查，自动检查冗余冲突，淘汰过时规则，并对边界变更提供明确差异对比，防止意外扩张藏在便利性背后。太长的指导没人看，太诗意的指导解读千奇百怪。变更管理能防止好心调整演变成全组织倒退。

8.6 能力工程反模式

8.6.1 反模式：无质量控制的定制

对 AI 队友指导和工作流程的本地调整，可能悄无声息地拉低效率、引入冲突，或者让工作集膨胀到没人理。由于这领域还不成熟，团队容易大量定制，心存侥幸。在舰队规模上，侥幸变成漂移，漂移变成不一致。解决方法是在整个平台底层建立审查纪律，加上对冗余、冲突和长度的自动检查。

8.6.2 反模式：诗意或详尽的指导方针

像“小心点”这种指导毫无约束力，而试图编码一切的指导又变成队友无法优先处理的噪音。这两种失效都表现为不一致，团队却误以为是“模型不可预测”。解决方法是少数稳定原则，加上触发绑定的程序，并明确层次和优先级。质量控制是必须的，因为舰队会像急切复制你的成功一样，复制你的错误。

8.6.3 反模式：将能力工程视为可选项或产品团队的副业

如果你只盯着代码，而忽略产生代码的 AI 队友系统，就是在优化智能体软件工程新堆栈里最小的乘数。AI 队友平台现在是软件的制造方式，在智能体吞吐量下，它成了质量、速度和一致性的主要决定因素。指望核心开发人员“有空时”搞平台工作，必然导致漂移和倒退。解决方法就是专人专责、明确路线图，以及对队友定义和认证的**严格质量关卡**。

8.6.4 反模式：用概率性指导去约束刚性流程

团队习惯将标准操作流程编写成指导文本，以为这样就能确保执行到位。但依赖随机性来执行底层操作，根本没法保证合规。指导里就算写上“必须”二字，也改变不了底层机制看概率脸色行事的事实。对行为偏好的偏离尚可容忍；但对安全扫描、合规检查或法规要求的违背，则绝对不行。解决方案很简单：刚性流程就该归入具有确定性触发的执行路径。如果一个流程非执行不可，那就用绕不过去的代码来强制执行，别指望那些可能被无视的文本指导。

8.6.5 反模式：把认知策略写进指导

那些规定“该如何分析”、“该如何调试”、“该如何设计”或“该如何入手”任何需要创造性思考的任务的指导或脚本，其实是在束缚手脚，而非赋能。无论这些文本是概率性指导还是确定性脚本，本质都一样：它们都用编写者预设的方法，取代了队友本应有的推理能力。萨顿提出的”苦涩教训”早已预言了这种结局：手工编码的知识短期看有帮助，但很快就会触及天花板，阻碍进一步发展。执着于编码认知策略的团队，实际上是从队友那里“榨取”了低于本应获得的收益，因为队友只顾跟着脚本走，却放弃了寻找更优路径。解决方案是：定义清楚目标、约束和验证机制。投资开发那些支持自主探索和自我验证的工具。把方法的选择权交还给队友。如果非得提供认知层面的建议，也应将其框定为可选的启发参考（行为偏好），而不是必须遵循的方法论。

8.7 主要构建模块

能力工程产出的构建模块，旨在让能力可扩展、行为可解释。

- 核心是 **队友定义**：一份形式规范、带版本号的说明书，用于明确角色、模型配置、工具权限、导师指导方针、自主

权边界默认值、认证状态和归属关系。每一条导师指导都应明确其意图（属于行为偏好、刚性流程还是认知策略）和实现机制（是概率性指导、确定性脚本还是混合模式）。对于刚性流程，定义中必须引用确定性的强制执行机制，不能只靠指导文本来保证合规。认知策略通常压根不该被编码；如果非要出现，也必须标记为可选的启发参考。

- 附上一套 **资质认证包**：即认证所需的一系列测试场景及相关的自动化检查，同样需要版本化，并可随时运行。
- 最后，维护一份 **轨迹与反馈台账**：记录所有交互过程、结果、事故和同行反馈，为能力校准、多维度分析和自我迭代训练提供燃料。

如果你想在企业里顺利推行这套东西，就必须明确归属关系。一个没有明确人类负责人的 AI 队友，其内部指导和自身状态会逐渐积累冲突与退化，直到“移除它本身”都成了新的风险。明确的归属关系，能把能力工程从“提示词技巧”的层面，提升为包含问责制和学习循环的正规运营规程。

8.8 如何衡量能力工程

8.8.1 指标：生产性连续工作时长

定义很简单：队友在不需人工干预的情况下，持续产出“合并就绪”工作成果的时间跨度。测量时要区分任务复杂度，别把处理简单任务的能力误判成应对复杂任务的本事。工作时长增长，说明角色、工具和指导三者配合默契。工作时长缩短，则通常指向角色边界不清、工作台支持太弱、指导本身脆弱，或者错把队友当成了只干琐碎编码活的“螺丝钉”，而不是承担使命级工作的合作伙伴。

8.8.2 指标：升级处理质量与路由准确率

追踪有多少升级请求是带着决策就绪的“咨询包”送达到位的，以及它们是否第一次就送到了正确的人类角色手上。低质量的升级消耗注意力，因为人类得自己重建上下文和梳理选项。路由错误的升级会导致延误，而整个队友群体则会将其放大为排队问题。提升这个指标，是减轻人类负担、同时保持队友效能最快的方法之一。

8.8.3 指标：变更后的认证回归率

追踪队友定义变更后，其资质认证包测试失败的频率，或在后续实际任务中导致失败增加的频率。这个指标之所以关键，是因为队友群体很容易因“回归”而失效：上个月还管用的法子，在模型、工具或指导方针更新后突然就不灵了。低回归率意味着你已经搭建起了靠谱的工程体系，可以放心大胆地迭代 AI 队友平台。高回归率则意味着你的能力假设建立在了过时的证据之上。

8.8.4 指标：指导方针膨胀率与冲突率

衡量指导方针规模的增速，以及检测到的冲突或冗余数量。膨胀是危险的，它会降低遵从度，并增加解释不一致的风险。冲突更危险，因为它迫使系统在重压之下做出随意的解决决策。健康的系统追求的是紧凑、层次清晰且可测试的指导体系。

8.8.5 指标：拒绝与升级的健康度

追踪队友拒绝其角色或能力范围之外任务的频率，以及事后看来这些拒绝是否正确。拒绝率太低，可能表明它在“来者不拒”，最终会演变为无声的胡来。拒绝率太高，则可能意味着校准不当或角色定义过窄。目标并非最大化或最小化拒绝，而是“恰当”地拒绝，并在任务本身合理但需要更高权限时，配合清晰明确的升级路径。

8.8.6 指标：机制与意图的匹配率

追踪有多大比例的导师指导工件，使用了与其意图相匹配的实现机制：行为偏好用概率性指导，刚性流程用确定性强制，认知策略压根不编码。错位会导致虚假信心（把刚性流程编码为指导）或潜力受限（把认知策略编码为指令）。定期审计导师指导工件，以便在引发事故之前，发现并纠正这些错位。

8.8.7 指标：指导遵从率

针对概率性指导，追踪指导在触发时实际被遵循的频率。这需要能检测触发事件和遵从结果的工具支持。低遵从率揭示了“大家觉得它可信”和“实际上它真靠谱”之间的差距。利用这些数据，可以判断哪些指导应该迁移到确定性强制执行，因为对其的偏离实际上是不可容忍的。

8.8.8 指标：指导工件的生命周期健康度

追踪指导工件的变更历史、测试覆盖率以及跨模型版本的兼容性。模型更新后，那些行为表现不一致的概率性指导，暴露了其底层依赖的脆弱性。健康的指导工件应该是版本化的、针对当前模型配置进行过测试的，并且其触发准确率和遵从率都经过评估。要用对待代码一样的严谨生命周期来管理指导工件，因为它们本身就是代码。

8.9 总结

能力工程将基础模型从“通用助手”转变为了角色明确、专业可信、可资质认证的协作者。通过将 AI 队友视为需要认证并持续改进的平台组件，我们构建了一个系统，它能让精英开发者的最佳实践，成为整个组织的默认配置。

本章讨论的这些模式与实践，并非一劳永逸的配置，而是持续不断的平台建设工作。从角色定义和能力校准起步。接着添加资质认证包和操作边界。再构建反馈循环和自我迭代训练。目标不是在第一天就拥有完美的 AI 队友，而是打造一个平台，让每个 AI 队友都能随时间推移变得越来越好，同时确保其能力有边界、行为可解释。

能力工程回答了这个问题："这个队友能做什么，我们如何证明?" 下一章，信任工程，将回答随之而来的问题："鉴于他们能做什么，我们该允许他们做什么?" 这两门学科相辅相成，缺一不可。

9　信任工程：以机器速度实现治理

9.1　根本矛盾

团队规模的自主性在提升吞吐量的同时，也扩大了潜在的破坏范围。信任总会失效——当这一刻来临时，你的系统必须能迅速回答四个问题：出了什么事、情况有多糟、我们的控制措施是否真的管用，以及如何防止重蹈覆辙。这里的矛盾在于，“一切皆代码”在非确定性系统中看似强大，实则暗藏陷阱——同一个高层意图，在不同的运行环境和上下文中，可能产生截然不同的输出。信任工程的目的，就是将信任从一个主观的个人特质，转变为一个可设计、可验证的系统属性。

安全的沙箱：限制爆炸半径

上一章的“能力工程”回答了这个问题：这位 AI 队友能干什么，我们能否证明？本章则要回答接下来的问题：既然它

能干这些，我们该允许它干哪些？能力工程通过资格认证和证明来提供能力证据。信任工程则利用这些证据来校准：授予多大的自主权、在什么条件下授予、需要配备哪些安全措施。两者相辅相成：缺乏信任治理的能力是失控的风险，而缺乏能力证据的信任治理不过是凭感觉下注。

这门学科也是破解“人力扩展陷阱”的良药。如果为了安全，就强迫人类手动审批每一个操作，那不过是将瓶颈从工程工作转移到了审批环节，最终依旧不堪重负。真正的出路在于分层的风险信任模型：它按风险等级匹配严谨性，并默认生成可供审计与验证的证据。运作良好时，信任能释放而非束缚自主性。

建立信任的一个有效方法是借助物料清单（BOM）和来源证明。但对于智能体系统，你的视野必须超越传统软件范畴。传统的 SBOM 思维回答“交付了什么”。构建来源证明回答“它是怎么造出来的”。而智能体系统还需要回答：“它是如何做决定的?”以及“事情是否真的按它说的那样发生了?”——因为协作者是“随机性贡献者”，而工作流程本身已是生产系统的一部分。

9.2 核心概念：信任工程的四大学科

在定义具体机制之前，有必要先厘清本章涉及的四大不同学科。它们常被混为一谈，但实际上各有侧重：回答不同的问题、在不同的时间点运作、需要不同的工程方法。它们共同构成了信任工程的完整闭环。

委派工程 是上游学科。它决定 AI 队友能获得多少自主权、在何种条件下、针对哪类任务，以及这种自主权如何随时间演变。它回答的核心问题是：*我们愿意放手什么，又必须死守哪些底线*?这是在行动开始前校准自主权边界的学问。风险分层、最小权限原则、工具访问边界、自主权门控等都

是委派工程实践：它们共同塑造了 AI 队友施展拳脚的“竞技场”。

安全工程 是运行时学科。它确保当信任被破坏时（无论是因为错误、漂移还是恶意攻击），系统能控制损害、防止故障雪崩。它回答：万一出问题，会发生什么？我们如何把损失框定在小范围？预防层、检测机制、断路器、自动回滚和自防御系统都属于此列：它们在执行期间运作，确保故障可控、系统可恢复。

问责工程 是追溯性学科。它确保系统能够说清发生了什么、证明控制措施确实生效，并生成用于学习、治理和合规的证据。它回答：我们能复盘事件经过，并证明决策是如何做出的吗？各类物料清单、冻结的审计记录、来源证明包以及事件学习循环，都属于问责工程范畴：它们在事后运作，让非确定性行为变得有迹可循、有据可查。

合规工程 是验证性学科。它确保系统的实际行为符合治理要求，而不仅仅是口头宣称。它回答：预期发生的事，真的发生了吗？我们能独立证明这一点吗？这门学科意在弥合政策与现实的鸿沟。它通过那些无法绕过的确定性执行钩子、独立的结果验证系统，以及能揪出确定性系统所遗漏问题的抽查机制来达成目标。

合规工程之所以必要，是因为 AI 队友和人类队友一样，都是“随机性贡献者”。一个随机性行动者可能报告自己遵循了流程，甚至自己也这么相信，但这份报告本身也是概率性的。一个声称“我运行了测试套件且全部通过”的智能体，可能只运行了测试子集，可能把模糊的输出误判为通过，甚至可能修改了测试以求过关。仅仅建立在自我报告基础上的信任，不是信任工程，那只是美好的愿望。合规工程，正是将“愿望”转化为“经过验证的信心”的过程。

对于那些以概率性指令形式编码的指导，合规缺口尤为明显。能力工程章节已阐明，教导 AI 队友可以通过概率性指导、确定性脚本或两者混合来实现。概率性指导仅适用于

那些不合规行为可以被容忍的“行为偏好”。而对于强制性程序，则必须依赖确定性执行机制——因为随机性的根基无法保证指导必然被遵循。合规工程要验证的，正是这些确定性机制是否正常运行，以及它们产生的结果是否真实，而非仅仅“执行了动作”这个表象。

这四门学科构成的是一个循环，而非等级结构。委派工程设定自主权的基本态势。当这个态势被证明不足时，安全工程出手控制故障。问责工程负责复盘事件经过。合规工程则验证预期是否与现实相符。没有委派工程，要么自主权泛滥（招致故障），要么自主权不足（扼杀吞吐量）。没有安全工程，信任一旦崩坏就会如决堤般失控。没有问责工程，系统无法从错误中学习。没有合规工程，组织将无法分辨一个系统是真正有效，还是仅仅看上去有效。唯有四者齐备，舰队规模的智能体软件工程才谈得上可靠可信。

移动应用生态系统提供了一个颇具启发性的先例。早期的 Android（6.0 之前）在安装时就迫使用户做出“全有或全无”的信任决策：要么接受应用索要的所有权限，要么干脆不装。结果用户要么不假思索地“全部接受”，要么因为一个多余的权限而放弃有用的应用。这套系统导致了最坏的局面：用户要么过于宽松（埋下安全隐患），要么过于严苛（错失实用功能）。向运行时权限（情境化、即时、可撤销、系统级强制执行）的转变，将移动信任从一个生硬的二元开关，变成了渐进、可调节的关系。今天的 AI 编程智能体正面临同样的进化压力，教训也如出一辙：强制“全有或全无”决策的信任模型终将失败；只有支持情境化、渐进式、可撤销的信任委派模型，才能真正扩展。

9.3 核心概念：可逆世界与委派校准

在委派校准中，一个常被低估的维度是 **可逆性**。当我们把任务委派给人类队友时，信任的校准部分基于错误可能造成的损害，部分基于损害能被多容易地撤销。相比一个无

法回滚的生产数据库迁移，我们更愿意授予一个拥有完整版本历史、经过良好测试的模块重构任务更多自主权。潜在的破坏范围固然重要，但恢复路径同样关键。

软件开发本质上是一个异常 **可逆的世界**。Git 保存了每一次变更的完整历史。容器可以推倒重来。基础设施即代码支持一键重部署。测试套件可以验证回滚是否恢复了预期状态。这种可逆性从根本上改变了委派的计算公式：当错误的代价有限且恢复路径清晰时，最优的委派策略自然会倾向于更大的自主性。问题从“我能信任这位队友永不犯错吗?”转变为“我能信任我的审查与回滚基础设施，足以发现并纠正错误吗?”

风险分层必须明确将可逆性纳入考量。完全可逆的操作（版本控制库中的代码修改、测试执行、文档更新）应被赋予比部分或完全不可逆的操作（生产部署、数据删除、外部通信、凭证轮换）更宽的自主权边界。可逆世界并没有消除对信任边界的需要，但它改变了这些边界应划在哪里、以及执行的严格程度。

可逆世界也说明了为什么四大信任工程学科必须围绕一个核心理念协同工作。委派工程利用可逆性来校准自主权边界：可逆操作值得更多放权。安全工程提供使世界在实践中可逆的回滚机制：比如 git revert、容器销毁、基础设施重部署。问责工程生成必要的证据链，以便知道要回滚什么、回滚到哪个状态。而合规工程则负责验证回滚是否确实将系统恢复到了预期状态——毕竟，一个报告“成功”的回滚命令，并不等同于一个经过验证、确认无误的系统。

9.4 核心概念：分层验证，或麦当劳给我们的关于随机性行动者的启示

AI 队友与人类队友都是随机性贡献者。谁都不可能每次都分毫不差地遵循每一个流程。工程挑战不在于消除随机性

行为（这不可能），而在于如何通过分层验证，从这些不可靠的组件中，构建出能可靠产出结果的系统。

想想麦当劳是如何在全球数万家门店，由技能和专注度各异的员工手中，实现始终如一的食品安全的。这套系统并不依赖任何一种单一的执行机制。相反，它分层部署了多种强度各异的控制机制，正是这种分层最终带来了可靠性。

有些控制是 **确定性的且无法绕过**。烤架温控器是固定的，员工无法将烹饪面调到不安全的温度。油炸锅也有工程设计的温度上限。这相当于智能体架构中的容器隔离或操作系统级沙箱：无论边界内的行动者想干什么，强制执行都有效。

有些控制是 **确定性的但需要辅助配合**。烹饪计时器在肉饼达到足够烤制时间后才会响起。系统会引导员工采取正确行为，但员工仍需做出响应。肉饼仍可能被过早取出或放置过久。这相当于权限申请和代码审查：系统会发出警报，但最终决定权仍在随机性行动者手中。

有些控制则完全依赖于 **行为合规，没有任何自动执行保障**。比如任务间隙洗手、佩戴发网、丢弃超时存放的食材。这些都是系统无法机械执行的流程期望。在智能体架构中，这类似于通过系统提示指示智能体遵循某些程序：智能体可能遵守，也可能部分遵守，还可能逐渐偏离。

而关键在于：**系统不会止步于合规期望**。麦当劳额外增加了一个验证层。卫生检查员进行抽查。经理在轮班期间审计。“神秘顾客”评估合规情况。这些验证机制专门捕捉那些仅靠行为期望无法预防的故障。它们不需要抓住每一个故障，只需要足够频繁地发现足够多的故障，从而使整个系统达到其可靠性目标。

同样的分层架构也适用于智能体软件工程中的 AI 队友。确定性钩子（必须通过的 CI 流水线、容器化构建、沙箱化执行）构成了基础层。辅助性控制（权限请求、自动化代码审

查、预提交钩子）形成了中间层。行为期望（系统提示、指导原则、编码标准）构成了顶层。而合规验证（智能体无法干扰的独立测试执行、输出验证流水线、对智能体决策记录的定期审计、对智能体自我报告与实际结果的抽查）则提供了能捕捉其他层次遗漏问题的那最后一层保障。

人们容易误以为确定性钩子就足够了。但事实并非如此，原因与“固定烤架恒温器不足以确保食品安全”一样。恒温器能保证温度，但保证不了烹饪时间。CI 流水线能保证测试“跑过了”，但保证不了测试“有意义”。智能体可能通过编写一些不测试任何实质内容的琐碎测试来“通过”CI。合规工程要捕捉的正是这个缺口：不仅要验证流程是否执行了，更要验证流程是否产出了真实、有效的结果。

9.5 核心概念：可解释自主性的三份清单

要让非确定性工作流程变得可解释，就得把“交付了什么”、“如何构建的”以及“如何决策的”看作三份必须能关联起来的独立记录。

- **软件物料清单**，即 SBOM，是份组件清单。它罗列了最终工件里包含哪些东西：依赖项、版本，以及与供应链相关的元数据。有了 SBOM，才能快速响应漏洞、满足合规要求，当务之急是能立刻回答“我们有没有中招”。
- **构建物料清单**，也叫 BuildBOM 或 BBOM，是份制造档案。它记录了工件的诞生过程：用了什么版本的工具链、经历了哪些构建步骤、环境如何配置、依赖项的快照，还有一切能让我们原样复现这次构建的源头信息。BBOM 关乎可复现性和防篡改能力，核心是要能回答“我们能否精确地重建这个工件，并放心使用它”。最近一些 SBOM 的工作已经开始把 BBOM 整合进去了。

- **决策物料清单**，即 DecisionBOM 或 DBOM，是智能体的行动日志。它记录了究竟是哪套智能体系统发起了变更：运行了哪个队友定义、生效的指导方针和政策包是哪个版本、调用了哪些工具、设定了何种自主权边界、发生了哪些升级、又经过了谁的批准。DBOM 是非确定性行为与工程问责制之间的桥梁。没有它，事后复盘就只能沦为“我们猜 AI 队友干了 X”，这对于安全和经验积累都远远不够。DBOM 也是两个在操作层面至关重要的子记录的天然归属地。
  - **模型物料清单**，即 MBOM，记录了基础模型的身份和配置，以及任何会影响行为的适配器或角色专属设定。
  - **工作流程物料清单**，即 WBOM，记录了约束行动的流程策略、工具权限和门控规则。

  你可以把它们作为显式部分塞进一个 DBOM 里，而不是当成独立对象。但从概念上讲，它们共同构成了“如何决策”的两面：运行了什么“心智”，以及什么规则约束着它。

9.6 核心概念：面向随机性行动者的策略即代码，与默认可审计性

要在舰队规模上建立信任，就得靠策略即代码——用可执行的约束来框定行为：规则、批准流程、访问控制和工具权限。策略得按风险分级：低风险的活儿可以靠自动化检查和事后审计兜着；高风险的活儿则必须经过明确批准和安全论证。关键在于，策略必须真能约束行为，不能光当摆设。如果策略构不成约束，那就是花架子。

对于委托工程来说，一个关键区别在于执行点放在哪儿。策略可以通过人工审批点来执行（这时人就是整个安全边界）、通过系统级机制执行（比如操作系统沙箱和网络隔离，不管人关不关注，它都有效），或者通过架构性的容器机制

执行（比如容器化和缓冲输出管道，就算智能体被完全攻破也有效）。执行机制越强，对持续人工盯防的依赖就越低，这一点至关重要。因为审批疲劳不是纪律的失败，而是要求人类持续保持高质量注意力（可他们还有其他事要操心）的系统的必然结果。但即使是最强的执行机制，也需要合规性验证：容器确实提供了隔离，但总得有人去验证智能体确实在容器里运行，并且没绕过它。

可审计性必须是默认选项，因为非确定性让“信任故事”变得不可靠。每个交付物都应该能关联到它的任务简报、生效的指导方针和政策、工具版本、追踪记录和证据工件，并且带有用于后续重建的冻结快照。审计追踪不是开销，而是让事后分析精准、让学习真实的关键。没有它，组织就无法改进，因为根本无法可靠地复盘。

9.7 信任工程中的关键实践

9.7.1 实践 1：风险分级与自主权门控

首先根据风险对工作面和行动进行分类：哪些可能导致安全倒退、数据丢失、服务中断、合规违规或不可逆的迁移。把这些风险等级与自主权设置挂钩：谁能执行、必须升级什么、需要什么证据。风险分级还得考虑可逆性：在可逆环境（比如版本控制的代码、可重建的容器）中行动，应该比那些会造成不可逆后果的行动享有更宽的自主权。这样就能形成一种可预测的委托姿态，人类不用每次都重新谈判。当风险分级明确时，自主权就变得可扩展，而不是让人焦虑。

9.7.2 实践 2：强制执行最小权限和工具访问边界

AI 队友不能图省事就权限泛滥。工具访问、凭证和部署权限必须限制在完成任务所需的最小范围内，并且尽可能加上时间限制。最小权限在智能体工作流程中尤其重要，因为

行动速度太快，等你发现错误时可能已经晚了。权限受限，爆炸半径就受限，治理才能让人信服。

这正是当前许多智能体工具链在企业里水土不服的地方。有些工具实际上鼓励一种“莽就完事儿”模式，给队友开放广泛的 Shell 访问、代码库权限，有时甚至是类似生产环境的操作权限，因为这能让演示看起来飞快。但企业可不能这么玩儿。整个自主权边界的意义，就在于定义允许什么访问、在什么条件下、承担什么证据义务、留下什么审计追踪。

9.7.3 实践 3：身份感知的信任边界

在企业环境里，AI 队友必须在指导者的信任边界内活动。这不仅意味着继承指导用户的技术权限，还得继承其组织角色、安全许可和上下文访问权限。不同用户问同样的问题，有权得到的答案可能不同。同一个用户在不同上下文（日常开发、事件响应、合规审计）中间同样的问题，也可能需要不同级别的访问权限。智能体的输出，包括日志、工件和对话历史，都将受到其内含信息的访问控制策略约束。没有身份感知的信任边界，AI 队友就会无意中成为绕过组织“需知”结构的通道——这不是出于恶意，纯粹是因为“乐于助人”而不受上下文约束。

9.7.4 实践 4：让审计追踪默认自动生成并冻结

别指望靠人类或 AI 队友自己记得去记录发生了什么。对整个工程系统进行埋点，自动捕获任务简报、咨询过程中的决策、工具调用、构建来源和证据输出，并在就绪包里引用。必须冻结的就得冻结：工具版本、依赖项快照和关键追踪记录，这样后续重建才有可能。这是防止“找不到根本原因”的故障无限循环的唯一办法。

9.7.5 实践 5：高风险工作的安全论证

对于高风险任务，要求明确的安全论证：一个由证据支撑的、论证工作为何安全的论点。安全论证得有结构：声明、支撑性检查，以及已知的剩余风险与缓解措施。这能避免争论变得主观，因为它迫使大家拿出证明，而不是空谈。当有了安全论证，审批反而会更快，因为评审者评估的是结构化的论点。

9.7.6 实践 6：分层合规性验证

别相信随机性行动者（无论是人还是 AI）的自我报告。要构建验证系统，独立确认控制措施确实执行了，并且产生了真实的结果。这意味着要有不可跳过的确定性执行钩子（好比烤架上的恒温器）、对输出的独立验证（好比卫生检查员），以及对流程遵守情况的定期抽查（好比神秘顾客）。验证层应该与风险成正比：低风险工作可以依赖带抽样的自动化检查，而高风险工作则需要独立执行和结果比对。最关键的是，合规性验证系统必须在架构上独立于被验证的行动者。如果智能体控制着测试套件，那它也就控制了测试结果——这算哪门子独立验证？就像财务审计一样，验证者必须独立于被审计的实体。

这种分层架构直接影响着如何编码指导。行为期望（概率性的指导、方针、编码标准）位于顶层，并且需要验证，因为合规性本身是概率性的。但强制性的程序根本不该放在顶层。它们属于确定性层：CI 流水线、钩子、自动化门控。当团队把强制性程序编码成概率性指导，而不是确定性执行时，就会产生合规性缺口，验证工作就得费更大劲去发现这些缺口。更好的架构是把强制性程序下放到确定性执行层，这样验证层就能专注于确认执行是否有效、结果是否真实，而不是去捕捉概率性指导必然产生的失败。关于如何匹配指导机制与意图的框架，请参阅能力工程章节。

9.7.7 实践 7：事件学习循环与治理更新

事件发生时，产出不仅是修复，还应该包括能更新系统的学习包。事后分析应该反馈到指导方针、操作边界、工作流程运行手册、工具链和政策中，因为这是随时间减少失败的唯一途径。这个循环必须是无责的，但不能无的放矢：系统通过流程学习，但人类仍然对批准和塑造允许失败的系统负责。如果学习没有落实到具体工件上，那就只剩下故事了。至关重要的是，学习必须完成完整循环：事件证据（问责制）为修订的风险评估和自主权边界（委托）提供依据，这些通过更新的容器机制（安全）来执行，并且更新的机制被验证为确实有效（合规）。

9.7.8 实践 8：将重新认证纳入常规操作

随着政策演变和工具更新，重新认证变得必要。要把队友资格套件视为信任循环的一部分：在旧模型版本或政策包下获得资格的队友，在新版本下可能就不合格了。这能防止能力假设过时导致的漂移。重新认证也让自主权的收回变得顺理成章，因为它与明确的标准挂钩，无需上演戏剧性场面。

9.7.9 实践 9：由证据驱动的渐进式委托

基于已证明的证据（而非感觉）来扩大自主权，把证据视为信任的硬通货。一个反复交付合并就绪和集成就绪工作、且决策记录清晰的 AI 队友，理应比一个需要不断纠错的队友获得更宽的边界。渐进式委托也是缓解人类不堪重负的良药：随着 AI 队友证明其可靠性，人类不再需要参与每个决策，而是可以提升到决策栈的上层，专注于真正需要判断的少数决策。这才是扩展之道：不是移除人类，而是把人类提升到更关键的环节。

晋升应该足够正式，以免变成会议驱动的形式主义。当 AI 队友获得新权限时，系统应记录更改了什么、为什么改、以

及什么证据支持这一更改。当队友权限被收回时，应该感觉像是回滚到之前的安全状态，而不是一场政治争论。当晋升和降级与版本化的标准以及分层资格认可挂钩时，它们就变成了正常的控制手段。

渐进式委托自然可以延伸到工作流程层面。成熟的组织不是只信任单个 AI 队友执行单个任务，而是可以信任整个流水线：一系列智能体，通过结构化的交接和审批点自主运行。工作流程级别的信任不是对流水线定义的盲目信任；它是通过同样基于证据的进展赢得的。从一个每个阶段都暂停等待人工批准的流水线开始，衡量成功率和失败模式，然后随着流水线证明其可靠性，逐步移除审批点。保留的审批点应该位于真正高风险的决策点，而不是常规的阶段转换点。这就是组织在不按比例增加人类注意力的前提下，扩展 AI 队友吞吐量的方式：人类设计和改进流水线，然后信任它去执行。关于流水线工程的具体机制，请参阅协调工程章节。

9.8 信任工程模式

9.8.1 模式：定义手术室，而非审批每一次下刀

随着 AI 队友能力增强，委托给它们的任务也变得更复杂，合适的信任机制必须从审批单个动作，转向定义队友可以自由活动的边界。你不会要求外科医生每切一刀都申请授权；你会定义手术室，建立协议，确保监控和干预机制到位，然后信任这个边界内的流程。这种模式的实践，就是将执行从人工审批点转向系统级容器机制（沙箱、容器隔离、缓冲输出管道），再辅以执行后检查结果（而非执行前门控每个操作）的输出验证。这让自主权能以机器速度运行，同时将安全保障在系统层面。

9.8.2 模式：安全论证思维

对于高风险工作，要求基于证据的明确论证，而不是信任信心或风格。这让安全性变得可审计，并压缩了评审时间，因为评审者能立刻看清推理结构。安全论证也留存了学习过程，因为它们记录了哪些检查重要以及原因。久而久之，模板就会浮现，严谨性也会标准化，而不会拖慢一切。

9.8.3 模式：无责，但不和稀泥

把事件视为必须引发系统改进的系统故障，而不是需要找人背锅的个人失败。同时，通过流程保持问责的真实性：谁设定了操作边界、谁批准了就绪包、哪些政策当时生效。这防止了“无人有错”演变成“无人学习”，后者才是长期的危害。当问责制与工件和决策挂钩，而不是与情绪挂钩时，它才具有建设性。

9.8.4 模式：风险分级治理

对不同风险等级应用不同的严谨性，并把这种差异编码到策略里，以免每次评审都要重新谈判。低风险变更可以通过自动化检查和事后审计处理，而高风险变更则需要明确批准和更强证据。这是在保持安全的同时，避免人类不堪重负的实用方法。当治理分级后，人类就能专注于那些真正需要判断的少数决策。

9.8.5 模式：将来源与物料清单提升为一级信任工件

将“交付了什么”、“如何构建”以及“如何决策”视为彼此独立、必须可链接且可审计的记录。SBOM 管组件，BBOM 管生产过程，DBOM 则涵盖定义行为模式的 AI 队友配置、指导原则、策略和工具调用。这套结构让非确定性系统变

得可解释。没有它，你既无法证明执行情况，也无法重现具体行为。

9.8.6 模式：验证验证者本身

合规性验证系统本身也是一个可能失效、漂移或过时的组件。用于验证智能体输出的测试套件，其可信度顶多和套件自身相当。因此，必须将“元验证”内建于系统中：那些合规性检查还在运行吗？仍然与时俱进吗？覆盖的范围还正确吗？这不是无限递归，其原理与安全关键系统中的有效冗余设计如出一辙。你不需要无限层级，只需要足够多的独立层次，让它们同时失效的概率低于你的风险承受阈值。

9.8.7 模式：自主权是旋钮，不是开关

应将自主权设置视为随任务而定的配置参数，而非“此队友可信”这类永久性标签。同一个队友，在低风险重构中可以放手去干，面对安全敏感部分则必须严格受限——这种灵活性本身就是一种优势。实践中，这个“旋钮”由分层资格体系支撑：当队友获得下一级资格认证时，其自主权边界随之扩大；一旦发生事件或出现功能回退，表明其实际能力超出了既有证据支撑的范围，边界就应收缩。如此，方能在可控的“爆炸半径”内维持高吞吐量。

9.8.8 模式：行为可追溯性是信任的硬要求

当队友做出决策时，系统必须能解释：这个决策究竟是源自任务简报、指导原则、策略约束，还是工具给出的信号？可追溯性不仅用于事后追责，更是为了持续改进——你无法修复一个连原因都找不到的问题。可审计性，正是自主权赢得信任的途径。在规模化系统中，“它当时为何那样做?”是个日常运营问题，其答案必须能从完整的问责轨迹中重建出来。

9.9 信任工程反模式

9.9.1 反模式：策略剧场

制定了一堆规则却不约束 AI 队友的行为，这只会制造虚假的安全感。团队因为策略已经白纸黑字写下来而感到安心，结果事件发生时才发现，压根没有机制去强制执行它。这种反模式尤其诱人，因为它看起来像是无需费力构建工具就能取得的进展。真正的解法是采用“策略即代码”，使其可执行，并辅以能证明策略确实被执行的审计钩子。

9.9.2 反模式：无根之木

如果你无法将行动追溯到具体的指令、工具版本、输入数据和证据，那就无法防止问题复发。“我们认为 AI 队友做了 X”这种说法，在非确定性环境下根本站不住脚，既经不起推敲，也无法让系统学到任何东西。找不到根本原因的故障还会摧毁信任，因为人类会感觉自己是在盲目操作。解法包括：冻结的审计轨迹、SBOM/BBOM/DBOM 的互相关联，以及结构化的事件分析包。

9.9.3 反模式：权限蔓延

AI 队友的权限一点点扩大，只因这样能减少操作摩擦。直到某一天，这些多余的权限成了重大事件的“助燃剂”。权限蔓延本质上是委托机制的失效：自主权边界扩大了，却没有相应的证据证明这种扩大是合理的。在舰队规模上，图一时之便会演变成系统性的漏洞，因为这种模式会在队友间复制。解决之道在于：最小权限原则、限时访问机制，以及在权限变更时重新进行资格认证。

9.9.4 反模式：YOLO 模式

仅仅为了让演示跑得更快，就授予 AI 队友广泛的 Shell 访问权、无限制的仓库写入权或接近生产环境的权限——这不是自主，而是放任“爆炸半径”不受控。它还会破坏问责制：一旦出问题，系统无法可靠地重现发生了什么，因为工作流程跳过了那些能产生清晰来源记录的约束环节。这同样属于委托工程失职：组织不去仔细校准队友的许可范围，而是默认“全部放开”，然后祈祷好运。YOLO 模式往往难以戒除，因为团队习惯了这种速度，并把后续任何收紧边界的举措都视为“平台倒退”，最终使组织陷入永久的“异常处理”状态。解法是：限定任务范围、实施基于风险分级的限时权限、对高风险操作强制执行策略门禁，并自动捕获信息至 DBOM，确保每个特权操作都可审计、可改进。

9.9.5 反模式：将审批当作主要安全机制

如果安全全靠“人类审批一切”，那你构建的就是一个无法扩展的系统。人类会不堪重负，审批会沦为橡皮图章，最终你会得到最坏的结果：吞吐量低下，安全性还脆弱。这与早期移动操作系统暴露的缺陷如出一辙：全有或全无的权限模型，只会导致体验要么危险地宽松，要么不合理地受限。真正的解法是：“构建即证据”加上风险分级门禁，让审批只出现在它能真正贡献判断力的地方，而不是制造拥堵的环节。安全必须内建于工作流中，而不是作为会后的补丁。信任必须通过系统级的强制力来体现，而非依赖人类持续保持警惕。

9.9.6 反模式：声明式合规

轻信 AI 队友关于“我已遵循流程”的自我报告，这不是合规工程，而是合规工程的缺失。一个报告“所有测试通过”的智能体，可能只运行了测试的子集，可能将模糊的输出曲解为通过，甚至可能修改了测试代码以求通过。同理，一

个报告“我已审阅代码”的人类开发者，也可能只是草草浏览。在这两种情况下，随机性贡献者都真心相信自己遵守了规定。声明合规与验证合规之间的差距，往往隐藏着最危险的故障，因为组织在并不安全时却自以为安全。解法在于独立验证：建立通过被验证方无法控制的渠道来确认结果的系统。这正是为什么财务审计要请外部公司，飞机检查要由独立检查员执行，以及为什么 AI 队友的合规性验证必须在架构上与队友本身分离。

这种反模式不仅限于自我报告，还延伸至指导原则的架构设计。当团队将强制性程序编码为概率性的指导方针，而非确定性的强制机制时，他们就是在系统设计层面实践声明式合规。指导方针只声明了“应该”发生什么，而随机的底层可能遵守，也可能不遵守。解法是双重的：首先，将强制性程序移入确定性执行机制（参见能力工程章节）；其次，为这些机制产生的结果建立独立验证。

9.9.7 反模式：错把指导当强制

团队将强制性程序编码为概率性的指导方针，并相信这些方针会被可靠遵循。这混淆了“表达意图”与“执行意图”。作用于随机性底层的概率性指导方针，无论措辞多么强硬（“必须”、“总是”、“绝不”），都无法提供任何合规保证。当某项程序真正是强制性时（如安全扫描、合规检查、身份验证），这种架构会产生合规缺口：组织自以为在执行程序，实则只是在建议。解法是根据意图匹配合适的机制：行为偏好可以使用概率性指导方针，因为偶尔不合规是可容忍的；而强制性程序需要确定性的强制力——无法绕过的钩子、门禁、流水线和自动化检查。随后，合规性验证应确认这些确定性机制在正常运行，而非试图捕捉概率性合规必然出现的失败。关于如何使指导机制与意图匹配的完整框架，请参见能力工程章节。

9.9.8 反模式：溜溜球式委托

因为上一次错误，就在“全权放手”和“啥也不准干”之间反复横跳，这会破坏学习进程和信任根基。队友永远无法稳定其操作模式，人类也永远无法对“什么会升级处理、什么可自行处理”形成可靠预期。溜溜球式委托通常是对事件的情绪化反应，而非对自主权边界的系统性重校准。解法是建立明确、有据可依的操作边界，其变化应基于证据而非情绪。决策权的稳定性，是吞吐量变得可预测的前提。当事件后确实需要收缩边界时，应收紧到某个具体的、有文档记录的状态，并附带明确的重新扩展标准，而非一刀切地退回“事事请示”模式。

9.10 主要构建模块

信任工程是工件驱动的，因为这是让非确定性变得可解释、可验证的不二法门。

- **委托边界** 定义了给定上下文中特定 AI 队友的自主权范围：允许使用哪些工具、拥有哪些权限、执行哪些操作，哪些必须升级，哪些被明令禁止。它应是风险分级、身份感知的，并考虑操作的可逆性。
- **策略包** 通过规则、审批和访问控制来约束行为，并应与队友定义和指导方针一同进行版本控制。策略包是委托边界的可执行表达。
- **来源包** 将 SBOM、BBOM 和 DBOM 关联起来，使得“交付了什么、如何构建、如何决定”能够被完整重建。
- **合规性验证计划** 指明哪些控制措施必须被独立验证、验证频率、验证机制及通过标准。它定义了验证的架构：哪些检查是确定性的（自动化、持续进行），哪些基于抽样（定期抽查），哪些由特定风险条件触发。该计划应与风险等级相匹配，并在委托边界变更时更新。
- **事件报告与事后分析包** 记录发生了什么、支撑结论的证据，以及必须修改哪些工件以使系统更健壮。它通过将

反馈注入委托边界、安全机制、问责标准和合规性验证计划，完成信任工程的闭环。

9.11 衡量信任工程

9.11.1 指标：委托边界覆盖率

统计在明确、风险分级的委托边界下运作的 AI 队友部署比例，而非使用默认或临时权限集的比例。低覆盖率意味着组织在没想清楚愿意委托什么的情况下就贸然委托，这是权限蔓延和 YOLO 模式的温床。目标应是对所有生产相关工作实现全覆盖；将覆盖缺口视为风险债务。

9.11.2 指标：策略违规与未遂事件率

追踪实际发生的策略违规，以及被控制措施成功拦截的未遂事件。未遂事件通常是自主权范围相对于当前控制措施过宽的早期预警信号。通过记录执行事件并按风险等级分类来衡量。未遂事件率上升，是收紧边界或强化平台控制力的信号，而不是去责怪 AI 队友。

9.11.3 指标：平均检测时间与平均控制时间

自主权只有在不安全状态持续时才会扩大爆炸半径。将“检测时间”定义为从不安全行为发生到触发警报的间隔，“控制时间”定义为从警报到成功遏制传播或完成回滚的间隔。这两项指标真实反映了你的可观测性、门禁和回滚能力是否到位。更短的控制时间，就是“爆炸半径受控”的实际定义。

9.11.4 指标：高风险工作的审计与来源完整性

将“完整性”定义为具备完整谱系的高风险变更所占比例。完整谱系包括：任务简报、指导方针版本、工具链版本、证

据工件与审批记录，以及 SBOM/BBOM/DBOM 的关联链接。缺失的轨迹就是未来无根之木事件的预演。应通过模式自动验证这一点，无需人类费心记忆。如果完整性低，则意味着在授予自主权的同时，并未构建安全学习的能力。

9.11.5 指标：合规性验证差异率

追踪当进行独立验证时，结果与智能体自我报告不一致的比例。这是信任工程领域的“财务审计差异率”。低差异率有助于建立对委托边界的合理信心。上升的差异率则表明智能体行为正在偏离预期，需要收紧委托边界或重新对智能体进行资格认证。关键在于要主动测量这个指标，而不是假设它没问题。那些从不独立验证智能体自我报告的组织，其合规性验证差异率实为“未知”——这是最危险的状态。

9.11.6 指标：强制性程序的确定性执行覆盖率

追踪被识别为强制性的程序（依据指导工件和策略包）中，由确定性执行机制（而非概率性指导方针）支持的比例。低覆盖率意味着组织存在合规缺口：一些程序被声明为强制性的，却仅仅依靠随机性解释来执行。目标应是对所有真正的强制性程序实现完全的确定性覆盖。将覆盖缺口视为合规债务，必须在组织能够真正信任其安全状况前予以解决。该指标是对能力工程中“机制-意图对齐率”的补充，但特别关注错位带来的信任影响。

9.12 以机器速度进行事件学习

传统的智能体软件工程事件响应手册，通常假设故障和恢复都以人类的速度发生。而在智能体软件工程时代，故障和恢复都可能以机器速度进行，但学习要产生意义，仍必须经过人类速度的消化。这带来了新挑战：我们该如何从 AI

队友可能在人类察觉前就已生成并解决的数百个微事件中学习？

答案不是把每个异常都当作需要事后分析的大事。相反，我们需要自动化的事件分类，根据严重性和模式将事件路由到不同处理流程。低严重性、高频率的事件应被汇总，分析其背后是否存在系统性问题。高严重性事件仍然需要人类主导的事后分析，但可借助 AI 辅助完成证据收集和初步根本原因假设生成。

更重要的是，学习必须是双向的。当人类识别出新的故障模式时，这种认知必须立即通过更新的策略、指导方针和资格套件传播给所有 AI 队友。当 AI 队友检测到异常时，这些信号必须上交给人类，由人类判断它们是否代表了值得编码到系统中的新风险模式。正是这种双向循环，让四大工程学科（委托、安全、合规、问责）真正“活”了起来：问责证据反馈给委托机制进行重新校准，校准结果更新安全机制，合规工程则验证这些机制是否真的有效，从而产生新的问责证据。

9.13 让系统学会自我防卫

系统具备自我防卫能力，可不意味着 AI 队友就能包治百病。它指的是系统内建了一套机制，能自动对已知故障进行检测、遏制、验证和恢复，无需人工插手。这需要四道防线：

- **预防层**：依托风险分级的自主权、最小权限和策略执行，将问题扼杀在摇篮里。这里是最经济的防线，也是委托工程大显身手之处。
- **检测层**：通过持续监控、异常检测和 AI 队友间的同行评审，及早揪出问题。多个 AI 队友在相关领域协同工作，彼此交叉检查，既创造了冗余，又省了人工开销。这便是安全工程的用武之地。

- **遏制层**：凭借断路器、自动回滚和爆炸半径限制，防止问题蔓延扩散。一旦 AI 队友的犯错率超过阈值，其自主权便自动收缩，直到人工审查查明原委。至此，循环完成了首轮：安全触发问责，问责重新校准委托。
- **验证层**：由独立系统确认预防、检测和遏制措施是否真管用。断路器该响时响了吗？回滚真把状态恢复了吗？监控系统逮住该逮的异常了吗？合规工程在此把关，确保自我防卫系统不是纸上谈兵。

关键在于，这些防御必须自动开启、默认生效。若需人工激活，便无法跟上智能体化吞吐量的步伐。系统必须以机器速度自我防卫，同时留足证据，让人能看清来龙去脉，从而优化防御。

9.14 用透明度撬动治理

在受监管的行业里，“AI 干的”这种托词可搪塞不过去。治理要求你能证明：控制措施已然到位，策略得到遵循，决策是按既定流程拍板的。正因如此，三个 BOM 才举足轻重——它们为治理提供了白纸黑字的证据。

但透明度不止于合规，更关乎信任。当开发者能看清 AI 队友如何决策、哪些约束在起作用、行动背后有何证据支撑时，他们才会对系统建立信心。反之，若缺乏透明度，每个 AI 生成的变更都会令人疑神疑鬼，审查负担必将骤增。

落实透明度，需要：

- 实时仪表盘，展示哪些策略正在生效并被执行
- 可搜索的审计日志，能回溯任何决策路径
- 清晰的所有权记录，标明哪个人类批准了哪些自主权设置
- 定期报告，揭示策略有效性与违规模式
- 合规验证结果，确认控制措施真在运行，而非形同虚设

这种透明度还催生了一种新的治理形式：预测性策略调整。通过分析未遂事件、策略违规和合规漏洞的模式，系统能在事发前推荐策略变更。这使得治理从被动转为主动，在变革速度以分钟而非月计的时代，这一点至关重要。

9.15 结语：领骑在前，无人轮换

信任工程让大规模智能体软件工程在高风险、强监管的环境中成为可能。委托、安全、问责与合规这四门学科，形成了一个闭环：委托工程校准自主权边界，安全工程在边界失守时遏制故障，问责工程重建事件全貌，合规工程则验证系统行为是否符合治理要求。通过实施基于风险分级且留有后路的委托、不轻信自我报告的分层验证、依托三个 BOM 的全程溯源，以及持续学习循环，我们构建了一个系统：AI 队友能以机器速度驰骋，却不会透支信任。

一如本书各章，我们探讨的主题确实够深。委托工程、安全工程、问责工程、合规工程——这四门学科每一样都足以单独成书，将来也必定会有。我们在此提供的并非完整论述，而是试图揭示这些领域成形之际必须直面的核心问题，为软件工程领导者提供一个即刻能用的行动框架，而非可望不可即的理论。

说实话，眼下的软件工程领导者，正处在最艰难的境地。你正骑行在主车群的前方。

在职业公路自行车赛中，主车群是一个紧密集团，车手们轮流到前方“领骑”。领骑者承受全部风阻，体力消耗之快，根本无法撑完全程。身后的队友则藏身气流中跟骑，为决胜时刻保存体力。这套之所以奏效，在于车手会轮换：当领骑者力竭，便侧移出列，滑至队尾，在他人顶风时恢复元气。

软件工程领导者眼下可没这等福气。他们在前方领骑，却无人轮换接手。AI 模型已然登场，智能体正在部署。而组织、法律、监管和教育界仍在后方跟骑，观望行业如何定调，再

作打算。法律界在等待判例，组织理论在观察团队演变，政策法规正对行业实践作出反应，教育界则在事后修订课程。

这不是抱怨，而是地形描述。软件工程领导者并非自愿站到最前。牌局已定，必须有人率先迎风：承受压力，消化不确定性，从艰难实践中汲取真知，并为后方领域开辟可追随的气流。

本书愿作这次骑行的伙伴。它提供的不是既定道路的地图——因为这些道路尚未铺就，而是一套用于实时探索未知地形的框架。第三部分阐述的五门平台学科——协调工程、工作台工程、能力工程、信任工程与语言工程——协同作用，营造出让 AI 队友真正翱翔的环境。它们共同将智能体软件工程从实验转变为企业级能力。

前方的风真实凛冽，但机遇同样巨大。当下躬身构建这些系统、通过艰苦实践而非理论空谈来正确处理委托、安全、问责与合规的领导者，将定义整个行业下一代的运作方式。这值得迎风骑行。

10 语言工程：人、AI 队友与机器共通的媒介

至此，我们的旅程从智能体软件工程的概念起步，遍历了作为协作者的 AI 队友、确保单人负责的实践，直至为多人多智能体建立信任的团队级工作台。本章我们要深挖一层，触及基石：编程语言与受限的表达形式。智能体软件工程的工作台，离不开工具、流水线和默认配置，但**语言**才是构建这一切默认配置的共通媒介。倘若语言本身就让含义难以追溯，再多流程也是隔靴搔痒。

软件的信任历来立基于两根支柱，大多数工程文化都离不开它们，无论嘴上是否承认。其一在个人层面：团队信任一段代码，是因为有人亲手写了它，仔细审阅过它，并在生产环境中与之相伴够久，从而对其脾性了如指掌。其二在结构层面：团队信任一个系统，是因为它的构建方式让某些故障难以发生甚至根本不可能发生，即便是能人也会失手。在第一个世界里，信任是与**工件**及其**作者**的关系。在第二个世界里，信任是一种**约束架构**。

关于第二种信任，一个贴切的类比是访问控制，而非惩罚。领导者可以相信一位训练有素的工程师不会删除生产数据，但绝不会在第一天就给他 root 权限。相反，组织会设计最小权限角色、默认只读设置、紧急“破窗”升级流程，以及需要双人批准才能执行的关键操作。由此获得的信任，并非基于“此人绝不会那样做”的信念，而是**让危险的结果在结构上就难以实现**，哪怕执行者状态不佳。

同样的分野，也正处在智能体软件工程时代编程语言选择的核心。在《工作台工程》一章中，我们将这类约束的团队级版本称为“工作台”；而**语言**，则是工作台中最严格、最

普遍的约束，因为它管辖着流经其中的每一个工件。智能体软件工程改变的不仅是代码的编写方式，更改变了**谁**需要理解它、**何时**需要理解它，以及在何种**时间压力下**理解它。

10.1 编写一旦廉价，阅读即成瓶颈

智能体软件工程打破了“我懂它，因为我造了它”与“它不可能那样失败，因为系统禁止了它”之间的传统平衡。当人类执笔大部分代码时，深入理解固然昂贵，但常常是作者身份的副产品。编写、调试并拥有一个功能，自然就塑造了心智模型。当 AI 队友编写的代码比重日益攀升时，这种亲密感不再是默认状态，而变成一项必须在后续审查、事件响应和长期维护期间**刻意投入**的成本。

即便没有 AI，这种鸿沟在人类团队中也早已存在。现代代码审查固然可贵，但它不等同于穷尽验证。审查能带来知识传递、协同对齐和发现明显缺陷等诸多好处，却无法像形式化证明那样，可靠地揪出每一个潜在问题。智能体时代的不同在于**规模**：生产流水线加速了，而人类与组织验证含义的能力却不会自动随之扩展。

这种现象日益被称为“验证债”。逻辑很直白：人类编码将创造与理解紧密耦合，而机器生成则需要在审查期间重建理解。重建是一项实打实的成本。随着生成代码的成本趋近于零，理解和拥有它的成本却保持不变甚至攀升，这种不对称性构成了对领导力的硬约束。

正因如此，语言选择变得**更加重要**，而非相反。如果一个团队无力承担完全消化每一个生成变更的成本，那么语言及其约束就必须扛起更多的安全性与可审查性重担。当编写唾手可得，审查带宽却捉襟见肘时，“易于审查”就从一种偏好，转变为主导性的经济现实。

10.2 语言选择：跨越双重模块性的沟通工程

编程语言的争论常被视作文化战争：品味、美学、招聘偏好、社区氛围。但在智能体软件工程时代，语言选择变得严肃得多。它成了一项设计决策，关乎人类与 AI 如何就**含义**进行沟通——这需要跨越两种迥异的模块性。

一种模块性是**面向人类的软件工程**。其主要活动是审查、调试、事件响应、审计与长期维护。它优化的是可理解性、有界推理和低歧义性。另一种模块性是**面向智能体的软件工程**。其主要活动是生成、转换、重构、机械化传播变更，以及合成那些人类不愿亲手写的脚手架代码。它优化的是吞吐量、覆盖率和规模化的一致性。

编程语言及其强制执行的子集，恰好位于这两种模块性的接缝处。它既是智能体向人类传达工作的共享语言，也是人类对智能体产出内容施加硬性约束的共享语言。这就是为什么语言不仅仅是“我们命令计算机做事的方式”，它更成为了**问责的主要媒介**。

问责并非抽象概念。在医疗、金融、关键基础设施等受监管的环境中，“是 AI 干的”绝非可接受的托词。组织必须对其交付的产物负责，智能体的高吞吐量并不会稀释这份所有权。这一现实要求建立系统思维和绝对所有权的文化，重申“构建者负责，而非工具负责”的原则。其实际含义很直接：领导者必须设计一条从意图到工件的路径，**既要能让人类审查，也要能让机器检查**。

由此引出本章的核心主张：智能体软件工程首先是个**沟通问题**，其次才是代码生成问题。如果沟通层失效，速度上去了，理解却下来了。失败模式不仅仅是“缺陷变多”，更是**更严重的缺陷得以逃脱**，因为它们藏身于无人真正吃透的代码之中。

10.3 程序理解研究数十年来一直在敲什么警钟

这些道理在概念上并不新鲜。新鲜的只是**规模**。程序理解长久以来被视为软件工程的核心学科，因为大部分工作量发生在首次提交之后：维护、检查、扩展、迁移和再工程。国际程序理解研讨会（IWPC）和会议（ICPC）的存在，恰恰说明理解绝非“软性”关切，而是真实系统中成本与风险的主要驱动力。

一个颇有影响力的思想流派源自“符号的认知维度”框架。该框架不问一种语言是否优雅，而是问一种表示法是否支持人们实际进行的认知活动：扫视、比较、安全修改和预测后果。它为领导者反复遭遇的权衡点命名，例如一致性、可见性、角色表达性，以及框架所称的变更“粘滞性”。重点不在学术分类，而在于一个朴实的提醒：**表示法是认知工具，而认知工具理应服务于现实工作，而非精巧本身**。

第二条线索来自信息觅食理论在编程中的应用。该领域研究将程序理解描述为一个在工件间寻找、关联和收集信息的过程，由“信息气味”指引，而非对文件的线性通读。关于调试的研究进一步强调，导航本身耗费大量时间，程序员将相当精力花在代码间移动和关联上下文上，而非孤立地阅读片段。在智能体软件工程中，工件图急剧扩张：更多文件、更多生成的胶水代码、更多辅助模块、更多惯用法变体。除非语言和工具刻意压缩搜索空间，否则觅食成本将随这张图同步增长。

第三条思想流派将可读性与可理解性视作经验属性，而非个人意见。关于可读性指标的奠基性工作，将结构、词汇特征与人类判断联系起来，表明理解成本存在可测量的信号，组织可以管理而非敷衍了事。这一传统的研究并不宣称存在唯一的最佳语言，但它支持一种重要的领导姿态：**可读性不是氛围，也不纯粹是个人品味问题**。

这些思想流派汇聚成一个硬道理：**理解是设计出来的**。更安全、更可扩展的路径在于减少歧义、压缩搜索空间，并标准化代码中意义的载体。

10.4 将代码阅读视作核心活动：非新见解，乃新主流

拥有强大软件工程文化的大型组织，早就在为代码阅读和协同进行优化，因为规模化软件是项团队运动。在这些环境中，主导活动并非孤独程序员创作优雅杰作，而是许多人经年累月地修改同一系统，伴随着版本更迭、人员交接和组织变动。这一现实倒逼组织重视惯例、可预测的结构，以及让协作得以进行的审查工作流。

时间使用研究也指向同一方向。真实的工程工作混杂了读写、审查、调试、测试与协调。重点不在精确百分比，而在于阅读、审查和理解**早已是**真实工作的核心。智能体软件工程只是进一步巩固了其主导地位，因为它降低了产生额外代码的边际成本。

这就是成本曲线至关重要的原因。当编写成本趋近于零，许多语言采用的决策就从**作者便利性**转向**审查成本**和**拥有成本**。曾经的偏好变成了治理问题，曾经关于表达力的争论，变成了关于可审计性的争论。

10.5 一条可行之路：可扩展的可审计性，而非清教徒式约束

Go 语言是个启发性的例子，因为其设计者异常明确地阐述了他们的优化目标。其框架很直接：Go 是关于**服务软件工程**的语言设计，而非为了最大化表达力本身。统一的格式

化、常规的结构、对枯燥清晰的偏爱而非精巧密集——这些都不是偶然，而是设计的核心。

与此形成鲜明对比的是 Perl 的那句著名格言：“达成目标的方法不止一种。”Perl 自身的文档拥抱风格多样性，并告诫读者勿以为某种风格天生优越。这种姿态在小规模环境或专家手中或许能激发创造力。但在智能体软件工程时代，它在操作上变得昂贵，因为风格自由度扩大了审查者必须理解的空间。

其背后的认知现实很简单。人类依赖**语义地标**进行快速扫描。几乎总是以相同方式书写的循环，就成了可靠的路标。遵循可预测结构的错误处理，成了穿越复杂性的捷径。看起来就像容器遍历的容器遍历，降低了差一错误潜藏于二十行簿记代码中的风险。当每个团队、每个智能体都在语言内部发明一套迷你语言时，理解便崩塌为考古发掘。

这也是“代码密度”需要谨慎拿捏之处。当表达压缩能清晰、一致地传达常见意图时，它是有价值的。当它创造出将多个概念塞进一行的“巧妙”代码，并要求审查者在脑中模拟类型系统时，它就成了负担。AI 队友很乐意产出任何一种。因此，语言及其强制的方言，必须偏向于那种**在减少出错表面的同时，保持意图清晰可辨**的密度。

10.6 为何反应式质量保证在智能体洪流下难以为继

传统的质量保证不可或缺，但在智能体规模化下力不从心。测试、审查和事件驱动的加固，在构造上就是**反应式**的。它们能证明缺陷存在，却通常无法证明缺陷不存在。在人类对大多数代码都了然于胸的世界里，局部理解常常掩盖了反应式保证的不足。随着亲密感降低，这层掩盖便失效了。

这不仅是个哲学关切。多项独立研究和行业报告观察到，当提示涉及常见缺陷模式时，代码生成系统在相当一部分情况下会产生不安全的实现。对领导者而言，关键细节不是某个单一数字，而是趋势方向：**生成易，验证难**。如果组织无法让验证能力与生成输出线性扩展，就必须投资于约束，以降低漏网之鱼的概率和严重性。

这就是可信赖语言的价值所在。它们并未消除审查的必要，但改变了经济性。如果一种语言通过构造就消除了整类故障，那么审查预算就能从发现低级陷阱，转向验证业务逻辑和系统不变量。组织得以将稀缺的人类注意力，投入**高阶正确性**，而非反复出现的低级错误上。

10.7 构造安全正成为基线，而非小众偏好

内存安全是最具体、最清晰的例子，因为安全社区的立场已异常鲜明。主要供应商和安全机构认为，内存安全问题代表了一大类主要漏洞，单靠流程改进无法在规模上根除。重点不在于某一种语言能解决所有安全问题，而在于**在语言和平台层面消除一大类主要故障**，能够改变漏洞曲线，这是仅靠培训和审查难以企及的。

在没有严格边界和补偿控制的情况下，于内存不安全的环境中扩展智能体代码生成，无异于以可预见的方式，让组织积累灾难性风险的速度，快于其偿还的速度。在智能体洪流下，失败模式不再是“多几个缺陷”，而是**存在的代码量**与**曾被人类真正理解过的代码量**之间日益扩大的鸿沟。

10.8 治理救不了错误的基础

人们总爱把智能体行为简单地归结为策略和提示问题。总想着，只要写好指令，添上风格指南和代码检查规则，智能

体就会乖乖听话。这种想法漏掉了关键一环：智能体的本能从何而来。

这些系统是在现实世界的软件上训练出来的。训练数据成了它们思考代码、风格、抽象以及何为“好代码”的默认标准。提示可以微调，但改不了深度学习的底子。尤其当上下文窗口被截断、任务变化，或是智能体遇到领域内各种司空见惯的权衡取舍时，提示就靠不住了。软性指导会失效，就连硬性护栏最终也不得不与底层基础较劲，而不仅仅是塑造它。

打个比方：这像是习惯，而非规则。几句日常劝告，改不了数十年根深蒂固的行为，尤其当周遭环境还在不断强化旧模式时。在智能体软件工程里，环境就是编程语言、生态系统以及智能体能够触及的功能接口。如果底层基础让编写晦涩代码、玩弄类型花招、大搞宏元编程或脆弱的反射变得轻而易举，那么无论多少“请保持简单”的请求，也拦不住智能体在压力下钻这些空子。

治理依然重要，但得摆正位置。它是好基础的放大器，而非替代品。一种语言内置的规范格式化、约定俗成的结构越丰富，语义变化越有限，组织就越不用跟模型的先验知识较劲。反之，语言提供的“逃生舱口”越多，组织就越会陷入一场打地鼠式的必败之战，疲于禁止各种冒出来的取巧行为。

所以说，选择语言不光看它能表达什么，更要看智能体凭其习得本能会默认表达什么。选择底层基础，是杠杆率最高的治理决策，因为它奠定了所有后续控制都必须面对的初始状态。

10.9 关键桥梁：从英语意图到可检验的意义

长远之计不是逼人更快、更多地读代码，而是提升人类审查意义的层次，让机器来承担检验意义的工作。

人类天生习惯用英语表达意图，它至今仍是沟通目标和约束的最高带宽接口。但英语并非规范语言：它可压缩、依赖语境、且暗含共享假设。这在对话中是优点，在工程里却成了弱点——缺陷恰恰藏在那些假设里。在智能体的高吞吐量下，这些隐藏的假设不只是引发误解，更会让误解规模化、扩散开，甚至固化成“能跑”的代码，直到有人惊觉契约从未真正精确过。

大多数组织没法要求每个工程师都给日常系统写完整的形式化证明。一条可行的中间道路是“交互式形式化”：人类用自然语言陈述意图；AI 提出更精确的工件（如结构化需求、模型、策略、不变集）；然后人类修正、收紧，直至其贴合现实。关键在于，这是一个协作循环，而非文档仪式。意义是逐步协商出来的：机器负责繁琐的规范化苦力活，人类则专注于语义判断。

10.9.1 受限自然语言：一座实用桥梁

并非每个团队都能一步登天，直接从原始英语跳转到 TLA+、Alloy、Lean 或模型检查。实践中，第一步往往更接地气：给英语加上足够的约束，让它不再是自由散漫的散文，而更像一张可测试行为的清单。

广泛使用的方法之一是 EARS（简易需求语法），由阿利斯泰尔 · 马文（Alistair Mavin，需求工程专家）提出。EARS 并不把需求变成形式逻辑，目标更为务实：它温和地将自然语言需求约束到一小套模板里，使其更清晰、更易测试、也更便于分解。

核心思想很简单：大多数需求都暗含 (a) 条件和触发器，以及 (b) 预期的系统响应。EARS 标准化了这些部分的写法，关键是按一致顺序排列，让读者无需猜测。其通用结构如下：

通用 EARS 形式："在 <可选前提条件/状态> 时，当 <...>，系统应 <...>。"

子句顺序的规则比乍看起来更重要。它迫使作者明确行为发生的条件，而不是将其偷偷塞进模糊的散文里。它还鼓励一种需求风格：先设置状态，再交代事件，最后是可观察的期望——这正贴合了编写测试的思路。

10.9.2 EARS 简明指南

EARS 定义了几种常见模式，对应不同类型的需求。用几个关键词就能概括：

- **普遍性：**"系统应始终 。"
  用于始终为真的属性或不变行为。
- **事件驱动：**"当 ，系统应 。"
  用于某事发生且系统必须响应的场景。
- **状态驱动：**"在 时，系统应 。"
  用于某个条件成立时所需的行为。
- **可选功能或条件性：**"若 <功能/条件>，系统应 。"
  用于行为仅在特定功能或配置下存在时。
- **不希望的行为：**"如果 ，那么系统应 。"
  用于错误处理及故障应对。

必要时可以组合子句（例如"在 ... 时，当 ...，系统应 ..."），但宗旨是保持每个需求足够小，以便测试能够合理覆盖。

10.9.3 为何 EARS 在智能体工作流中变得至关重要

在纯人类工作流里，自由形式的英语通常“够用”，因为工程师靠来回沟通和共享上下文来澄清模糊之处。智能体软件工程打破了这个假设。编码智能体很乐意将模糊性“操作化”：它选一个解释，一贯地实现，并生成一大堆看似合理的周边代码。这正是那种会积压“验证债务”的输出。

EARS 之所以有用，不是因为它神奇，而是因为它限制了智能体的自由度。它迫使需求必须声明：(1) 处于何种情况（状态或前提条件），(2) 发生了什么（触发器），以及 (3) 系统必须做什么（响应）。

它还建立了从需求到测试再到任务的清晰映射。每个 EARS 语句本身就像一个验收标准：一个条件或事件，加上一个可观察的期望。这正是智能体用来构思测试用例和实现步骤的形状，也是评审者无需重读长篇大论就能审计的形状。

一个表明这正从理论走向实操的具体信号是，AWS 的智能体 IDE Kiro 就明确采用了 EARS 风格的验收标准，格式如下：

WHEN [condition/event]
THE SYSTEM SHALL [expected behavior]

这种简化很有启发性。它不把 EARS 当作需求宗教，而更多地视其为人类意图与机器执行之间一种实用的握手格式。

真正的要点在于：EARS 是一条双向道，而非让你独自填写的表格。与 AI 协作使用 EARS 的最高杠杆点，不在于让人一夜之间变成需求写作大师，而在于建立这样一个协作循环：

- 人类从原始英语意图开始（例如“用户应能导出报告；得支持离线；出错要明显”）。
- AI 队友初步将其规范化为 EARS 版本（例如“当用户点击导出时...”、“离线时...”、“如果导出失败...”）。

- 人类修正含义：模糊动词变可测量输出，缺失的状态被补全，边缘情况被增删，非需求被过滤。
- AI 队友随后将澄清后的 EARS 需求反向映射到测试计划和实现任务分解中，最好还能保持可追溯性（从需求 ID 到测试，再到任务和代码）。

这是最低层级的交互式形式化。你不是在证明定理，而是在迫使意图进入一种可审计、机器可操作的形状。机器负责重复性的结构化工作，人类则提供语义和最终判断。

10.9.4 EARS 不是万能药：何时需要其他工具

EARS 并非万能溶剂。就连 EARS 指南也承认，有些需求不太适合这些模板，尤其是当复杂性爆炸、前提条件众多，或者需求本质上是数学性的时候。

这一点很重要，因为在规范驱动的工作流中，容易错把模板当目标，而非把清晰度当目标。如果你把一个本该是公式、表格、状态机、类型契约或模型的需求，生硬地塞进 EARS，反而可能失去精确性。

一种实用的态度是：

- **对行为契约**（流程、状态、错误处理、可观察结果）使用 EARS。
- **对数学或算法真理**（公式、不变量、基于属性的测试、可执行参考实现）使用其他专用符号。
- **当正确性关乎随时间推移的交互时**，使用领域模型（状态图、序列图、协议跟踪）。

元教训始终如一：我们需要超越无约束的英语。有时，那个“超越之物”是结构化英语，有时是数学，有时是模型，有时则是一门专门的语言。

AWS 展示了阶梯的高端：形式化方法与可分析策略。他们应用形式化规范和模型检查，发现了关键分布式系统中一

些严重且隐蔽的错误——这些错误甚至在设计评审、代码评审和广泛测试中都幸存了下来。这并非学术空谈。

重点不在于形式化方法没有成本，而在于对某些类别的系统而言，它能比投入更多人力进行评审更早地发现缺陷。换句话说，当语义足够精确到可以被机器检查时，人类评审就能提升一个层次：评审者可以将更多时间花在语义审查和反例驱动的讨论上，而不是浪费在逐行揣摩非正式散文上。

授权策略领域尤为突出，因为它将风险高度压缩。微小的策略错误可能演变成重大漏洞。Cedar 语言就是一个范例：它不把授权策略视为分散的应用代码，而是将其设计成一门符合人体工程学、快速、安全且可分析的专门语言。Cedar 的形式语义已在 Lean 中被证明，并配备了专门用于验证和分析策略行为的工具。

一个近未来的工作流已隐约可见：人类用英语写下策略需求，AI 将其翻译成 Cedar 策略，并利用分析工具生成反例或差异解释，说明策略如何变化。人类则审查语义（例如“谁能在何种属性下做什么”），而不是在海量服务代码中搜寻授权逻辑。这是一种微型“通信工程”，它暗示了整个技术栈可能的演进方向：意义从某人写下的东西，转变为可以被检验的东西。

10.9.5 近期论点：携手攀登形式化阶梯

将 EARS 和 Cedar 放在一起看，模式就清晰了：EARS 是轻量级约束，把英语变成测试形状的语句；Cedar 是专为分析设计的领域特定语言；形式化方法则是最高保证的阶梯，其模型可在整个状态空间中被检查。它们并非相互竞争的宗教，而是同一阶梯上的不同横档。

在智能体软件工程时代，攀登这个阶梯成了一种实际需要，因为代码的吞吐量正压倒人的理解力。我们可以期待，在帮助人类与 AI 队友共同攀登的接口方面会快速进展：例如，

将原始英语转化为结构化需求、将结构化需求转化为测试和任务、在适当时将需求转化为模型和可分析策略的工具。目标不是让英语消失，而是让英语不再是承载意义的唯一载体。

10.10 语义评审成熟前，语法仍关键，毕竟人类看的还是它

人们总憧憬一个未来：人类不再读代码，只评审规范、不变量和证明。那世界正徐徐而来，但眼下，大多数组织仍要靠阅读和评审代码来交付。代码依然是智能体向人类“汇报工作”的主要工件，也是机器执行的最终对象。

这使得编程语言成了包裹智能体输出的“信任信封”。这个信封有两面：一面要尽可能消除危险的故障模式；另一面要压缩人类为理解剩余部分而构建正确心智模型所需的工作量。

这重新定义了语言“能力”的角色。在智能体软件工程时代，最危险的语言不一定是最弱的，而是那些以扩展解释空间的方式展现“强大”的语言。一种允许密集元编程、巧妙运算符重载和无限制反射的语言，可以是专家的游乐场。但在智能体工作流中，它可能成为一种负担——因为 AI 队友能大规模利用这些特性，生成能编译但语义上极难审计的代码。强大的类型系统或许能帮忙，但前提是它们与人类可理解的习语，以及天生倾向于清晰而非取巧的底层基础相结合。

在这个叙事中，函数式编程也需谨慎对待。函数式世界观——强调不可变性、显式数据流和受限副作用——确实能改善局部推理能力，减少“远距离操作”。然而，“函数式”并非魔法咒语。许多函数式语言允许极其抽象的类型操作，而许多非函数式语言也能通过有纪律地使用不可变性和显式接口来达成局部推理。核心问题很实际：评审者能否在

不阅读整个系统的情况下，就理解副作用和依赖关系的波及范围。

10.11 重复代码变天了：当 AI 能自动传播变更时

人类工程师对重复代码深恶痛绝，这完全情有可原。手动克隆代码意味着未来任何修复都得在多个地方重做一遍，代码漂移几乎无法避免。但 AI 队友彻底颠覆了这套成本模型——它们可以被派去满代码库传播变更、进行重构，而且还不知疲倦。

到了智能体软件工程时代，只要同步是自动化的，重复代码不仅变得可以接受，有时甚至更可取。它能创造出更小巧、功能更聚焦、局部更易理解的代码单元。二十个相似的小函数，可能比一个让评审者摸不着头脑的“万能”抽象结构更安全可靠。关键在于，组织必须投资于自动化重构、模板管理和回归验证，确保克隆出去的代码不会悄无声息地各自为政。

这就引出了另一个关键点：语言选择至关重要。有些生态让自动化重构变得既安全又常规，而另一些则脆弱不堪。如果指望 AI 队友执行跨代码库的转换，那么组织最好选择那些格式稳定、语义可预测、并配备能可靠重写代码而不搞砸的工具的语言。

10.12 面向智能体时代的语言组合拳

“一门语言通吃”只是个安慰剂式的神话。领导者需要驾驭的是一个组合拳：有实时约束的嵌入式系统、需要分布式并发的后端服务、追求快速迭代的前端、处理混乱现实的数据管道，以及安全至上的策略层。智能体软件工程非但没有消除这种多样性，反而因为变更吞吐量剧增，将其放大了。

在系统和嵌入式领域，使用 C 和 C++的压力依然真实存在。遗留代码、硬件约束和既有工具链不会因为领导一纸命令就消失。然而，智能体规模化理应改变我们的默认立场。内存安全应当成为新代码在可行时的基线要求，而非遥不可及的理想，毕竟漏洞证据确凿，而验证成本又无法随规模扩展。这里的组合思维类似于划分明确的隔离区：将内存不安全的“危险区”尽可能压缩，通过狭窄的接口隔离，并将智能体的主要生产力引导至内存安全的“绿色区域”。当 C/C++实在无法避免时，应视其为危险品，设置更严格的门禁和边界。

在后端和分布式系统中，目标是在安全属性与易于理解之间取得平衡。Go 是专为大规模工程环境下的可读性和一致性设计的典范。Java、C#等托管语言通过运行时和成熟工具提供了内存安全。Rust 能提供强大的保证，但风险在于其不受约束的表达能力可能带来高昂的认知成本。同样的道理再次浮现：领导者应将组织期望的保证与期望的行为分开，然后选择那些让期望行为成为默认选项的底层技术与生态。

在前端和产品层，类型的意义与其说是证明，不如说是契约。TypeScript 能提升接口清晰度，减少模块边界处的歧义，这对于智能体生成代码尤为重要。但它并不能自动保证运行时行为，而 JavaScript 生态反而可能放大依赖和供应链风险。此处的治理存在于严格的编译器设置、可强制执行的约定和依赖管控中，而选择那些能最小化语义差异的模式总是有益的。

在脚本、数据或机器学习管道中，动态性依然关键，尤其是在 Python 主导的生态里。正确的姿态是控制其破坏半径，而非彻底禁止。类型化覆盖层、边界处的运行时验证、受限执行环境和密钥管理，远比争论语言是否优雅更重要。指导原则是：灵活性需要用更严格的边界来交换，因为智能体生成的粘合代码可能看似合理，实则暗藏脆弱的假设。

在安全至上的策略层，专用的可分析语言往往优于通用代码。Cedar 就是一个很好的参照，它从一开始就是为策略表达和分析而构建，并配备了旨在超越测试用例、推理策略行为变化的工具。当风险是“一念之差，满盘皆输”时，受限的语义和自动推理就不再是奢侈品，而是必要的风险控制手段。

10.12.1 语言比较：安全性、可评性与工具生态

下图浓缩了当人类需要在智能体规模下评审代码时至关重要的四个维度。两个坐标轴衡量的是决定组织在代码量激增时能否保持问责制的属性。点的颜色衡量第三个属性，它决定了组织能否顺利推行其期望的“规约子集”。单向条形图则可视化第四个在智能体时代独具重要性的属性：生态系统中现实惯用语的分布有多广泛，从而将 AI 输出从不良模式引开有多困难。

这张图故意坚持“实践优先”，并对变异度保持清醒认识。这是智能体软件工程应有的偏向，因为 AI 队友并非白纸一张。它们带着从公开代码中学到的先验知识而来，这些知识会将生成结果推向生态系统中常见的模式。变异度低的语言更容易保持规约，因而在大规模评审时成本更低。变异度高的语言并非不可用，但它们会施加持续的引导成本，并增加难以审查的模式混入的概率——因为评审者不可能穷尽现实世界中所有的可能性。

构造安全性（x 轴）衡量的是，有多少主要的故障类别在默认情况下就被消除，而无需非凡的纪律性。内存安全是头号例子，但该维度还包括空安全默认值、竞态避免机制，以及危险操作是否被明确标识并可设限。语言可以通过强大的编译时保证（例如显式的“不安全”围栏），或者运行在默认情况下就结构性移除了内存损坏的托管运行时上，从而在此维度获得高分。

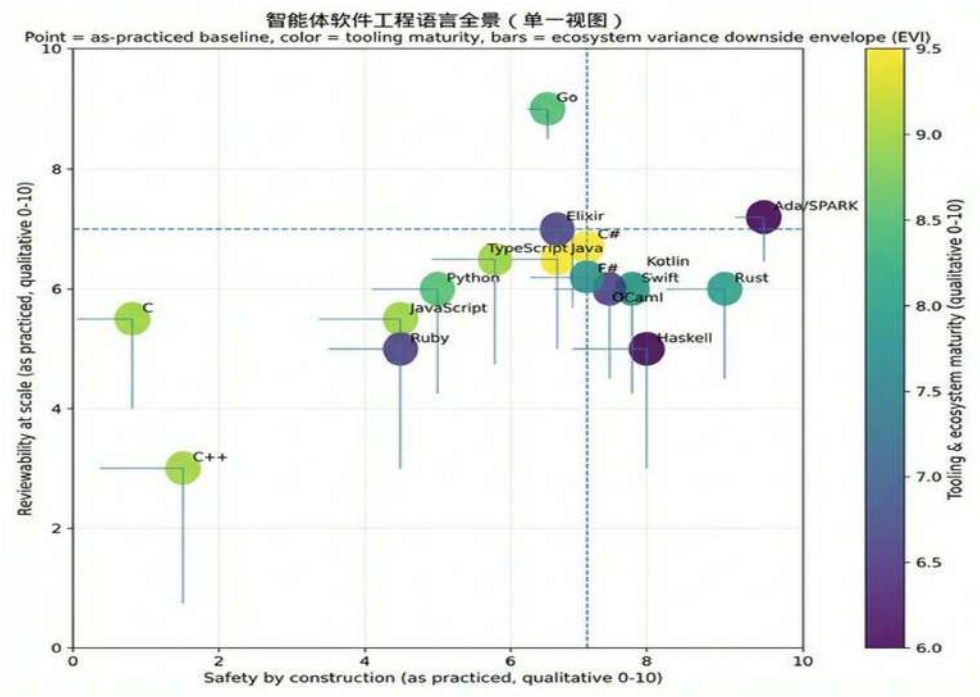
大规模可评审性（y 轴）衡量的是，人类能否在拥有众多作者（包括非人类作者）的大型代码库中高效审查代码含义。这关乎低语义变异度和有界的局部推理：可预测的惯用语、一致的结构，以及能将代码推向常规形态的生态系统。这不是关于专家能否写出漂亮的代码，而是关于大型组织能否在持续变更下保持代码库的规约。

工具与生态成熟度（颜色）是实际的倍增器。它反映了生态对格式化、代码检查、构建系统、自动化重构、静态分析、调试剖析、依赖管理以及保持代码一致性的制度性模式的支持程度。工具不能替代底层技术，但它决定了该技术能否大规模运行，尤其是在期望智能体执行跨代码库转换时。

领域变异度（单向条形图）是在智能体软件工程中变得举足轻重的维度。条形图根据生态系统变异度指数（EVI）得出，该指数估算了公开代码中风格和惯用语分布的广度。这些条形图是单向的。它们并非声称一种语言不能写得干净，而是在说，该生态系统包含一个庞大、唾手可得、但可评审性较低的模式空间，而 AI 队友受到从这一公共分布中学到的先验知识的强烈影响。在实践中，这意味着引导存在硬性限制：提示和策略可以微调，但它们很少能覆盖生态系统级别的默认值。具有长变异度条的语言，就是智能体会反复“探索”语言和生态系统刁钻角落的语言，组织将不得不持续投入评审和纠正预算，才能将输出拉回规约子集。EVI 可分解为四大类：

- **极低变异度（Ada/SPARK, Go）：**约定强大，惯用语空间受限。领域分布窄，因此智能体默认输出更容易与“规约”对齐。
- **中等变异度（Rust, F#, Swift, Java/C#）：**核心强大，但在抽象深度和生态驱动的分歧上仍有显著空间。这是智能体环境中的主战场。

- **高变异度（Python, TypeScript, Ruby）：**动态行为或存在大量“逃生舱口”和框架，加上现实世界中风格多样性极其丰富。
- **极高变异度（C++, JavaScript）：**功能表面巨大或生态系统极其多样，整个领域中充斥着众多相互竞争的惯用语和方言。AI 队友将在此空间内积极“探险”。



请将此图解读为一个决策曲面，而非严格的排行榜。x 轴（构造安全性）从左边的“多数缺陷类别需靠纪律和评审预防”向右边的“主要故障类别被结构性移除或变得明确可控”移动。y 轴（大规模可评审性）从底部的“含义难以在众多作者和变更中可靠审查”向顶部的“即便持续变动，含义依然清晰规约”移动。

- 右上象限是智能体软件工程的理想国，它同时降低了灾难性故障风险和人类评审成本，而后者在代码生成极其廉价时已成为主导瓶颈。
- 左下象限是危险地带，它将低内置安全性与低可审查性相结合，在高吞吐量下将验证成本转化为实实在在的操作风险。
- 另外两个象限虽可行但需明确权衡：高安全性但可评审性较低的语言，需要更强的惯用语标准化和更严格的“规约子集”约束来保持代码可评；高可评审性但安全性较低的语言，则需要更强的运行时边界和补偿控制，因为严重的故障类别在默认情况下仍可能发生。

点的颜色反映了工具与生态的成熟度：即组织执行约定、自动化重构和大规模运行该语言的难易程度。单向条形图可视化了领域变异度：由生态范围内惯用语多样性造成的下行压力。这一点至关重要，因为 AI 队友继承了来自公开代码的强大先验知识；除非受到底层技术的约束，否则它们会反复从生态系统的可达模式中取样。当一种语言的默认值与期望的约定一致时，治理便顺流而下。当二者相悖时，治理就变成了一场针对根深蒂固的先验知识和生态习惯的持久战。

10.12.2 各语言在四个维度上的解读

C. C 语言的安全分偏低，这没什么好奇怪的：内存不安全与未定义行为本就是其默认编程模型的一部分，所以这门语言从设计上就没打算消除那些灾难性故障。它的可审查

性看似中等，实则暗藏玄机。核心语言虽小，但宏、构建标志和那些微妙的别名假设，常常会在大规模代码中破坏局部推理。工具链倒是成熟得很，诊断和执行都帮得上忙，但终究无法根除底层风险。图中的方差条如实反映了现实世界 C 代码的光谱——从讲究纪律的子集，到被宏重度污染的构建时“迷你语言”。在野生数据上训练出来的智能体，会把这些统统复现出来；而我们的引导，往往也只能事后补救，难以防患于未然。

C++. C++是典型的高方差、低构造安全代表。内存不安全仍是它的底色，而它那庞大的特性集又支持着五花八门的范式和抽象风格，语义常常被隐藏在模板、重载规则和层层宏的背后。工具链固然强大，但语言的表现力意味着生态系统中充斥着各种惯用法，其中不乏给审计添堵的“奇技淫巧”。那条长长的方差条，就是给智能体软件工程敲响的警钟：模型早已从野生数据里学会了“聪明绝顶的 C++”，除非组织把语言严格限制在一个强制执行的子集内，否则它就会一直给你吐出这种代码。

Rust. Rust 在安全性上得分极高，它默认保证内存安全，同时用 `unsafe` 明确划出了逃逸通道，更从结构上杜绝了数据竞争。可审查性就比较复杂了：地道的 Rust 代码可以清晰易读，但这门语言也支持宏繁重、特质繁重的抽象，容易让人看得头晕。工具链强大且凝聚力强，有助于团队统一规范，但方差条说明生态里同时存在着干净利落的 Rust 和高度抽象的 Rust。在智能体软件工程工作流里，这意味着默认输出有时会偏爱那些基于宏的“巧妙”代码，除非你明确限定哪些惯用法才是被允许的。

Go. Go 是图中典型的“默认就乏味”的底层语言。安全性还算稳固，毕竟运行时保证内存安全，但诸如空值误用、并发误用这类语义错误，它默认可管不着。可审查性非常高，因为语言和生态都通过约定俗成的结构、外加强制执行的格式化工具，强力推动着统一的代码形态。工具链成熟度高，更重要的是领域方差低，这意味着智能体脑中的先验知识，

往往和你想要的“乏味”子集不谋而合。在智能体软件工程里，Go 那条短短的方差条就是个操作优势：你能省下更多精力去跟默认模式较劲，从而更专注地验证业务逻辑。

Java. Java 的安全性比那些内存不安全的语言强不少，毕竟运行时消除了内存破坏，静态类型也支撑着强有力的接口契约。在讲究纪律的代码库里，可审查性通常不错，但实践很容易走样，因为各种框架和企业模式常常把语义塞进注解、反射、配置和复杂的依赖图里。工具链成熟度极高，利于重构和保持一致性，但方差条的存在恰恰因为生态里包含着许多“框架方言”。在智能体软件工程的环境下，实际情况就是：模型倾向于生成生态里规范化的东西，包括那些充满“魔法”的模式，除非你主动约束框架，并强制执行一种更窄的架构风格。

C#. C#在各个轴上的位置和 Java 相近，但它得益于更强大的现代工具链和语言演进，在日常安全性上有所改善，比如采用后对空值处理更有纪律。可审查性也类似：语言本身可以保持高度统一，但常见的实践会用到强大的特性和框架，把语义藏在属性、反射和约定背后。工具链成熟度堪称业界翘楚，这也是为什么 C#在大规模项目中依然很好管理。方差条表明，智能体的输出通常会反映野生数据里那些功能丰富的模式，所以能否引导好，就看你如何决定语言的哪些部分是“允许的默认值”，并把它们固化到模板和关卡里。

Kotlin. Kotlin 安全性高，因为可空性是一等公民，数据建模工具也比许多老牌托管语言更强大，但互操作性和表达性特性也提供了逃逸通道。可审查性比不上 Go，因为 Kotlin 支持多种惯用法家族，包括类似 DSL 的模式，这些都可能增加语义方差。工具链很强大，毕竟 Kotlin 继承了 JVM 世界和出色的 IDE 支持，但方差条依然存在，因为生态系统默许了多种风格自由度。对于智能体软件工程，Kotlin 可以工作得很好，但前提是组织要有意收窄惯用法的选择空间，免得智能体把表达能力都转化成审查债。

Swift. Swift 的安全性不错，这要归功于可选类型和比手动指针语言更安全的内存模型，当然它仍然允许显式的不安全操作。可审查性中等：Swift 可以用清晰、常规的风格来写，但也支持更抽象的协议和泛型繁重的方法，这就会增

加方差。工具链良好，生态系统也比 Web 技术栈要清爽些，但风格广度仍然足以撑起一条可见的方差条。在智能体软件工程工作流中，当团队把少数几种惯用法和模式标准化之后，Swift 才最易于审查，这样智能体就不会自己发明一套方言。

F#. F#处在一个有趣的位置：它结合了内存安全的运行时、强大的静态类型，以及默认的函数式局部推理。因此安全性很高，但还达不到 Rust 或 SPARK 那种绝对程度，因为副作用、可变状态和互操作依然可用。如果团队坚持“乏味函数式”惯用法，可审查性会很强，但这门语言也支持计算表达式和类似 DSL 的模式，这些可能迅速拉大语义方差。工具链成熟度良好，因为它继承了.NET 的操作生态，但方差条表明，除非收窄约定范围，否则智能体输出会去探索那些表达性的犄角旮旯。在智能体软件工程中，F#恰恰在团队把它视为受约束的符号系统、而非 DSL 发明的游乐场时，才是可行的选择。

TypeScript. TypeScript 更像是一种契约语言，而非证明系统。安全性中等，因为它提升了接口正确性，能及早捕获很多错误，但类型在运行时会被擦除，而且系统容忍逃逸通道。如果强制执行严格设置，可审查性通常很好，因为类型让意图一目了然，但在野生数据中，类型级别的复杂度范围极广，`any` 风格的逃逸更是家常便饭。工具链成熟度极高，所以 TypeScript 能大规模操作，但那条方差条是对智能体软件工程的关键警告：模型已经学会了有纪律和无纪律的 TypeScript。如果你希望智能体生成可审查的 TypeScript，就必须让严格性和模式约束成为结构性的要求，而不是可有可无的建议。

JavaScript. JavaScript 的安全性有限，尽管运行时是内存安全的，但许多语义错误依然是运行时“惊喜”。在野生数据里，可审查性从低到中等，因为生态系统容忍极端的风格多样性、框架快速迭代、动态模式，以及那些把语义藏起来的构建时复杂度。工具链非常出色，但这本身并不会缩小可用

的惯用法空间。那条长长的方差条，捕捉了智能体软件工程的现实：在公共 Web 生态上训练的 AI 队友，会不断重新发现那些难以审计的“巧妙”模式。操作层面的含义就是，“用 JS 但要管好它”通常是场难打的仗，除非你把底层转向严格约束的 TypeScript，并把允许的惯用法空间收得很窄。

Python. Python 通过运行时实现了内存安全，这使它比 C 和 C++的安全基线高，但它把许多语义错误留作运行时“惊喜”。在小型或纪律严明的代码库里，可审查性还过得去，但一到大规模就直线下降，因为类型纪律差异巨大，动态模式在生态的很大一部分里都被视为常态。工具链成熟度高，但方差条反映了野生数据中包含许多相互竞争的约定和严格程度。在智能体软件工程工作流中，Python 在粘合和实验方面仍有价值，但领导者应预期更高的引导成本，并要把模式、运行时验证和边界检查视为底层的一部分，而不是可选的流程。

Ruby. Ruby 从运行时继承了内存安全性，但其生态系统通常大量使用元编程、DSL 繁重的模式和隐式的框架约定，这些都增加了语义方差。因此，除非团队非常努力地保持语言的“乏味”，否则可审查性在大规模下往往会恶化。工具链成熟度尚可，但不如那些最大生态系统那样具备统一的可执行性，这增加了操作摩擦。那条长长的方差条就是警告：在智能体软件工程中，Ruby 那些“魔法”模式正是模型会重现的东西，如果没有强大的结构性约束，很难让这些默认值偏离轨道。

Haskell. Haskell 的构造安全性高，强类型和纯函数式消除了一大类错误，减少了“远距离作用”。然而，可审查性高度依赖于惯用法的选择：Haskell 可以用直截了当的风格写，但生态系统也规范了非常密集的抽象和类型级别的表达。工具链成熟度相对小众，这影响了大尺度下的可操作性。方差条捕捉了智能体软件工程的风险：在公共 Haskell 数据上训练的模型，通常会提出优雅但密集的解决方案，组织必

须决定他们是否能足够强硬地强制执行一个“乏味”的惯用法子集，以保持审查的可扩展性。

OCaml. OCaml 结合了强类型与内存安全的运行时，并且通常以相对直接的工业风格使用，这可能使它比外界想象的更易于审查。安全性高，尽管不如那些专为形式验证设计的语言严格，因为可变状态和复杂的模块模式仍然可用。工具链小众但够用。方差条适中：OCaml 的使用方式确实有差异，但相比其他函数式社区，生态系统驱动的极端抽象压力较小。在智能体软件工程设置中，当团队将一小套惯用法标准化，并依赖编译器来保持接口显式时，OCaml 可以成为一个可行的组合选择。

Elixir. Elixir 是动态类型的，这限制了在编译时能消除的语义错误种类，但它受益于 BEAM 运行时和消息传递并发，从而消除了一大类共享状态危害。如果遵循 OTP 约定，可审查性可以很强，因为架构会落入可识别的形态，局部推理也保持有界。工具链健康但相对小众。那条较短的方差条反映出，生态系统比许多动态语言更受约定引导，这有助于智能体的引导性。在智能体软件工程中，当组织致力于 OTP 模式，并把偏差视为设计例外而非风格自由时，Elixir 的效果最佳。

Ada/SPARK. Ada，尤其是 SPARK，位于高安全性的顶端，因为其生态系统专为显式性和高保证性设计，在 SPARK 风格的使用中更包含了契约和验证纪律。可审查性也可能很高，因为约定强大，方差受到文化和验证工作流的约束。工具链是专业化的而非主流，但在其领域内非常深入。那条短短的方差条是对智能体软件工程的关键信号：领域分布更窄，因此智能体的先验知识不太可能提出差异巨大的惯用法。这是一种“意义保持可见”的底层设计，而这正是当人类需要承担责任时，你所需要的东西。

10.13 前路何方：代码成为新的二进制，意义上移一层

前瞻性的论点并非是要人类变成更快的代码阅读器。相反，行业将构建一套全新的工件栈，专门为人与 AI 之间关于“意义”的沟通而设计。

AWS 在形式化方法上的经验指明了一条路。精确的规范和模型检查可以在设计错误变成代码之前就逮住它们，尤其在复杂的分布式系统中。Cedar 则展示了另一条路。具备形式语义和分析工具链的领域特定约束语言，使得策略变更可以在语义层面进行检查。这两种情况模式相同：将审查从语法转向语义。

短期内，代码依然是人类审查的主要工件。这就是为什么语言选择依然是一个直接的杠杆。领导者应将当前时代视为过渡期。我们的目标是构建这样的系统：人类负责批准意图和不变量，而机器则生成并检查底层实现。随着那个未来到来，代码会越来越像今天的二进制文件：执行所必需，但不再是人类理解的主要媒介。

在那之前，语言选择是可用的、最高效的风险控制杠杆之一。这不是优雅不优雅的问题，而是一个组织在软件输出速度远超人类所能亲密掌控时，能否保持问责的问题。核心警告很简单：治理无法完全补偿选错的底层。从领域数据中得出的先验知识，让那变成一场必败之战。选择正确的基础，配上正确的默认约束，这个决策将决定所有其他控制措施是在逆流而上，还是顺流而下。

这就是为什么编程语言值得在关于智能体软件工程的书里拥有独立一章。前面的章节聚焦于实践和控制，但语言底层决定了在智能体软件工程的吞吐量下，什么是容易的、什么是困难的，甚至什么是可能验证的。一旦选定了底层，工作台工程的其余部分就变得具体了：可强制执行的“乏味

子集”、自动化重构、策略和配置语言，以及那些让智能体输出对人类可审查、对机器可检查的工具链。

Part IV

前进之路

你的软件工程 3.0 转型

至此，全景已在你眼前展开。第一部分阐明了何为智能体软件工程，以及 AI 队友 为何代表着一次根本性变革。第二部分为你提供了一套控制系统，让单个 AI 队友 变得值得信赖。第三部分则涵盖了平台工程这门学科，它是将协作从“一对一”扩展到“多对多”的关键。不妨这样理解：平台是空域，而第三部分的全部工作，就是制定飞行规则、提供全局视野并建立反馈循环，从而确保大量任务能够安全并行。

书中的实践、模式与反模式，绝非金科玉律。它们更像一份入门指南：哪些方法通常管用，哪些注定会搞砸，以及哪些证据足以平息争论。假以时日，每一项都会发展成独立的学科，拥有自己的负责人、专属工具、度量标准和持续改进机制。软件工程正驶入智能体时代，最终的形态为何，我们仍在探索之中。

万事第一步，总是殊途同归：把想法从脑子里倒出来，摊在阳光下。当我们用大白话将实践编纂成文，也就为团队创造了一个共同的参照系，供其检验、挑战、打磨与传承。工程学科正是如此诞生的：白纸黑字的运行规则、基于证据的辩论，以及稳步的迭代更新。一旦自主权边界、证据要求和问题升级路径都清晰明了，团队便无需再纠结于主观感受，转而可以就规则本身进行辩论。本书内容是你启航的燃料，请以此在你的环境中点燃引擎，并在学习过程中公开迭代——因为在智能体时代，你的软件工程体系本身，就是你交付的每一个产品背后最坚实的竞争优势。

停在车库里的法拉利，终究只是件昂贵的摆设。第四部分就是驾驶课：它告诉每位利益相关者，接下来具体该怎么做，才能将潜能转化为稳定、可重复的交付成果。

光知道不行动，无异于纸上谈兵。第四部分面向开发者、技术领导、业务高管、教育工作者和研究者，分别提供了具体的行动指南，帮助你启动或加速自身的软件工程 3.0 转型。

革命并非迫近，它已然降临。率先掌握智能体软件工程的组织，将定义软件的下一个时代。那些踌躇观望者终将发现，对手早已学会如何驾驭机器的速度进行开发，且并未以牺牲信任为代价。问题不在于你是否要引入 AI 队友，而在于你能以多快的速度，构建出让它们既可信赖又能规模化运作的工程体系。

这最后一部分，既是对全书知识的凝练总结，也是一份关于你下一步行动的实用路线图。法拉利已就位。是时候学会如何安全地风驰电掣了。

车库里的法拉利仍然只是车库的装饰品。

第四部分是驾驶课程：每个利益相关者接下来必须做什么，才能将能力转化为可重复的交付。

无法促成实际行动的框架，不过是精致的摆设。第四部分便为开发者、技术领导者、业务高管、教育者和研究者指明了道路，提供了启动或加速软件工程 3.0 转型的具体步骤。

革命并非迫在眉睫，它已经发生。能够驾驭智能体软件工程的组织，将成为软件新时代的定义者。若你选择等待，便会眼睁睁看着竞争对手以机器的速度疾驰，却并未丢掉“可信”这枚压舱石。真正的选择，无关乎是否采用 AI 队友，而在于你能多快打造出一套让它们既可靠又可扩展的工程系统。

这最后一部分，既融会贯通了你之所学，也为你绘制了清晰的下一步行动图。法拉利已在轰鸣。是时候安全地，全速前进了。

11　你的软件工程 3.0 革命，现在发车：开法拉利的感觉

旅程至此告一段落，但对你来说，好戏才刚刚开场。现在你应该明白了，智能体编程既不是魔法，也绝非软件工程的末日。恰恰相反，软件工程终于在此刻回归了它应有的面貌：一门真正的工程学科，能够以空前规模，从并不完美的部件中，锻造出值得信赖的成果。

跟你交个底吧。你的 AI 队友会犯错，但你今天手底下那 100

本书始终坦诚而辩证地看待风险与回报。我们没有承诺魔法。我们向你展示了，要想安全驾驭那些“随机性队友”，需要怎样的软件工程纪律。现在，轮到你做决定了：是要建立起这套纪律，还是假装买张许可证就等于完成了转型？

11.1　代码从来不是目标

几十年来，我们不知不觉给自己洗了脑，以为软件工程就是写代码。大错特错。软件工程是一个产出成果的体系，它建立在四大支柱之上：**行动者**（角色、激励、自主权、责任）、**流程**（工作流、关卡、节奏、协议）、**工件**（需求、设计、测试、代码、运维手册、证据）以及**工具**（编译器、持续集成、分析器、流水线，以及如今的智能体工具链）。代码只是众多工件之一，从来不是终极使命。

大多数领导者都忽略了一个关键的认知矫正：智能体编程工具链以**工具**的形式出现在你的系统中，却以**行动者**的方式运作。它们主动出击、制定计划、执行任务，并以足以淹没人类判断的规模，产出看似“完工”的成果。如果你把行

动者当工具管，就会建立错误的管控机制。这正是天真的尝试会惨败收场的原因。你推行的是工具，但你真正需要的，是让行动者“入职”——包括厘清自主权边界、证据标准和问题升级路径。

这场演变鲜明且不可逆转。软件工程 1.0 是“古典时代”，人类驱动每一个循环，产出受限于人力和注意力。软件工程 2.0 带来了帮你打字、搜索的副驾驶，但人类依然坐在驾驶位，充当核心的安全阀。软件工程 3.0 迈入“智能体时代”，AI 队友以机器速度进行计划、执行、分支并交付完成的工作。产出呈爆炸式增长，人类注意力成为瓶颈，整个工程体系必须随之革新。如果你把软件工程 3.0 当作更花哨的软件工程 2.0 来对待，那么你打造的组织，其产出变更的速度将远超其理解能力。

11.2 我们共同构建了什么

本书带你进行了一次系统性的旅程，从认清问题到设计解法。我们首先区分了“氛围编程”与“结构化工程”，展示了非正式协作如何带来速度而非信任。答案不是消灭氛围，它在探索和原型设计中自有其价值，关键在于认清何时风险要求我们引入结构。这种结构具体体现为一系列**工件**：阐述意图的“任务简报”、塑造能力的“指导包”、控制执行的“工作流程运行手册”、处理升级的“咨询请求包”、提供证据的“合并就绪包”，以及记录决策的“决议记录”。

我们引入了两种泾渭分明的模式，因为混为一谈只会两败俱伤。**人本软件工程**优化审查、调试、审计与长期理解。**智本软件工程**则优化执行吞吐量、工具编排与证据生成。人类需要一个能俯瞰全局、做出决策的指挥中心。AI 队友则需要一个边界清晰、反馈迅速的执行环境。把这两个工作台搅和在一起，你得到的将是聊天驱动的混乱，而非工程级的可靠。

我们向你展示了如何通过可复现的交互模式，将 AI 的廉价迭代与广博知识转化为工程优势，同时避免产出沦为混乱。我们引入了“保障工程”，以应对随机性队友的四个典型悖论：热情有余而理解不足、上下文过载、视野狭隘以及缺乏记忆的经验。解法不是手把手教，而是通过“任务工程”和“上下文工程”实现工程化的控制，让可信赖的行为成为默认项，而非英雄壮举。

在团队规模层面，我们揭示了平台工程何以变得至关重要。当你拥有众多人类、众多 AI 队友和众多任务时，故障模式就不再是糟糕的代码，而更像是糟糕的编排。答案在于协调工程、工作台工程、能力工程与信任工程，依次将整个团队视作一个必须在负载下保持一致的分布式系统来对待。最后，我们深入基础层，揭示了当读取与验证的成本超过编写时，编程语言的选择便成了一项治理决策。未来不在于加速人工代码审查，而在于将人类的关注点提升至意义、策略与约束层面，同时让机器去处理实现细节。

11.3 愚人的天堂

容我点破你即将犯下的错误：你会买下数百套智能体编程工具链的许可证，然后坐等魔法发生。你的开发者会啧啧称奇，演示效果会震撼人心，产出也会激增。然后，现实会给你当头一棒。

拿着工具的傻瓜，终究还是傻瓜。这可不是引入一套智能体工具链然后宣布胜利那么简单。这些工具链不仅仅生成代码，它们还会让你系统中所有原本就糟糕的东西加速恶化：含糊不清的意图、薄弱的验收标准、缺失的证据、混乱的集成、脆弱的审查以及装样子的治理。

如果你的工程体系没有对这四大支柱（行动者、流程、工件、工具）进行彻头彻尾的重新思考，那么你所谓的“采用

智能体软件工程”，不过是给官僚机构装上了一台高速打印机，然后惊讶于它为何大规模生产废话。

更令人沮丧的是，这些工具将以其十分之一的速度潜力运行，因为系统从未为它们优化过。你手下的“人类”又何尝不是如此？这不仅是质量和信任债务的灾难，更是买了一辆法拉利却只能在学区里爬行。你支付了巅峰性能的费用，却满足于龟速前进。

弗雷德·布鲁克斯几十年前在《没有银弹》中就警告过我们，没有哪一种单一的技术能带来数量级的改进。他区分了**本质复杂性**（问题本身固有的难度）和**偶然复杂性**（我们因糟糕的工具和流程而自找的麻烦）。智能体工具链能通过处理语法、样板代码和机械转换，大幅削减偶然复杂性。但它们无法消除本质复杂性：理解用户的真实需求、做出架构权衡、管理冲突需求，以及判断在具体语境中何为“足够好”。如果你把 AI 队友当作能神奇消除所有复杂性的银弹，那就完全曲解了布鲁克斯的核心洞见。本质复杂性依然存在，事实上，当偶然复杂性被剥离后，它反而变得更加关键。

残酷的现实是，当你的组织交付变更的速度，快于其能证明变更合理性的速度时，“信任债务”便开始累积。当决策无从追溯、证据缺失、审查沦为走过场，而“它通过了测试”成为你唯一能讲的故事时，你拥有的就不是工程，而是在用持续集成流水线赌博。信任债务起初感觉不像债务，它感觉像速度。直到某一天，你连一些基本问题都答不上来：我们为何做这个改动？谁批准的？有什么证据证明它是安全的？当时有哪些约束条件？如果它失败了，我们如何管控？

AI 队友能以人类从未体验过的速度制造信任债务。这不是因为我们蠢，而是因为我们遗留的流程是为人类吞吐量设计的，而我们现在正迈入机器吞吐量的时代。将其视为简单的工具推广，而非对整个体系四大支柱的系统性重思，不仅是天真，更是危险。你需要完成两种模式的转变：人类如

何治理，以及智能体如何执行。跳过这个转变，速度只会变成自我伤害。

11.4 工程学的本质就是管理不确定性

有一点或许能让你安心：AI 队友的概率属性并不可怕。工程学从来不是关于完美部件。桥梁安全，不是因为钢材永不失效，而是因为我们设计了具备冗余、安全裕度、检查制度、标准化实践以及对故障模式明确预案的体系。软件也一直在与不可靠的硬件、不稳定的网络以及会误解需求的人类打交道。问题从来不是贡献者是否完美，而是体系能否在不完美中产出可信赖的成果。

别再仅仅以“AI 是否确定”来评判它。你应该评判的是，你的软件工程体系能否从“随机性贡献者”那里催生出信任。未来的赢家，不会是坐等完美智能体的组织，而是那些懂得如何从概率性工作中构建确定性证据的组织。当你透过这个视角审视智能体工具，它们就不再可怕，反而变得熟悉起来。它们只是工程学科可以驾驭、用以产出可靠价值的又一个不完美来源。

正是这种概率性现实，使得本书所阐述的一切都至关重要。任务工程确保含糊的意图不会变成错误的实现。上下文工程防止信息过载演变为混乱的执行。基于证据的监督让审查关乎“证明”，而非“说服”。平台工程确保规模扩张不意味着混乱失控。这些不是你流程中可有可无的装饰品，而是让你能安全驾驶法拉利的工程控制装置。

智能体软件工程运作框架

智能体软件工程并非简单的工具变更，而是一场思维模式的革命，它迫使权力流向、决策权以及“完成”的定义发生改变。起初你会感觉更慢，因为你正在用结构取代氛围，用

可复现的控制取代个人英雄主义。最初的摩擦不是失败，而是为了构建一个在吞吐量垂直攀升时不会散架的东西，所必须付出的代价。

这从根本上是一个人力资源问题，而非信息技术问题。不妨把它想象成整合来自不同国家、拥有不同工作风格、沟通模式和文化假设的团队成员。你不会只给他们一台笔记本电脑就指望无缝协作。你会设计入职流程、建立沟通协议、明确决策权并构建反馈循环。AI 队友同样需要这种系统性的整合，只不过它们以机器速度运作，并且永远不会真正意义上“学会”你的组织文化。

要让这场转变落地，你需要三个层面的赋能，它们各自针对那些可能扼杀智能体软件工程转型的“组织抗体”。

- **在一线战壕**，**智能体软件工程导师**指导团队适应新的工作方式，防止他们将软件工程 2.0 的习惯带入 3.0 的工作流。他们强制要求清晰的意图、明确的约束、证据预期和恰当的升级路径。没有这个角色，团队只会搞出“带聊天机器人的迷你瀑布模型”，还美其名曰转型。
- **在运营层面**，**首席智能体软件工程导师**的协调范围超越单个团队，覆盖组织的骨干：平台、发布、安全、合规与运维。他们主导“智能体软件工程行会”，让组织在其中公开学习，将实践经验转化为共享标准和通途。没有这个角色，你得到的将是“智能体大塞车”，而非“智能体高吞吐”。
- **在战略层面**，**智能体软件工程教练**弥合一线现实、外部技术格局以及只有在工程体系重新设计后才会显现的真正潜力之间的鸿沟。他们戳破管理层的幻想，将工程现实转化为治理要求，并防止“速度”沦为政治武器。

关键的转变在于认识到，你的瓶颈已经转移。软件生产不再是限制因素，决策才是。在软件工程 3.0 中，生成代码是廉价的，但决定构建什么、如何构建以及它是否已准备好交付，却成了新的约束。好消息是，智能体系统擅长生成选

项和替代方案，能让你的决策更加明智。坏消息是，如果你的决策流程没有准备好应对机器速度的选项生成，那么你将在积累信任债务的同时，再添一笔“决策债务”。

11.5 致开发者：你的学员与队友正翘首以盼

亲爱的开发者，如果你正在读这段话，请先明确一点：你并非被取代，只是价值坐标发生了迁移。你的价值从来不在打字快慢，而在于约束条件下的判断力：定义意图、识别风险、设计接口、要求佐证，以及做出那些需要品味与担当的决断。AI 队友的产出可能是你的十倍，但它们无法为决策赋予正当性，更无法承担失败带来的后果。

你与 AI 队友的关系，必须从“使唤工具”转变为“亦师亦友”。当你把它们看作学员，一切就不同了。它们的成功让你脸上有光，它们的失败则暴露你指导尚有不足。这可不是什么温情脉脉的想法——而是硬邦邦的操作现实。立场一变，你的态度自然从恼火转为投入。你不再抱怨它们犯错，转而开始设计成长路径：用更清晰的任务简报、更明确的约束边界、持续迭代的指导包来铺路。

但它们不止是学员，更是队友。这种互为师生的关系，才是真正威力所在。你提供本土知识和领域上下文，引导它们在浩如烟海的参数空间里找准方向；它们则提供广度、备选方案和不知疲倦的迭代，从而极大拓展你的能力边界。这不是简单的一加一等于二，而是一加一变成十甚至一百。未来成功的开发者，绝不会把 AI 当作高级自动补全工具，而是那些能驾驭人机乐团、让双方优势彼此激荡的指挥家。

有句流行语说：“AI 不会取代人类，但会用 AI 的人会取代不用 AI 的人。”这话只说对了一半。完整的真相是：那些善用 AI 的人——不止是当工具使，而是借 AI 之眼彻底重审自身技艺——将取代所有其他人。使用工具只是入场券，重构系统才是制胜关键。这意味着要站在软件工程 3.0 的层面思考，而非仅仅自动化眼前那点事。这意味着要懂得何时该用氛围编程和即兴探索，何时又必须回归结构化的工程方法。

实验与仿真是 AI 队友让成本变得可控的核心工程技能。好好利用它们。并行尝试多种路子。该舍弃就果断舍弃，别被沉没成本套住手脚。大胆构建你本就打算丢弃的原型。当迭代成本近乎为零时，这些做法绝非浪费，而是你在广阔设计空间里寻优的必经之路。你的角色是巧妙设计实验、坦诚评估证据，并知道何时探索必须收敛为扎实的工程实践。

未来能产生十倍乃至百倍影响力的开发者，将不再自视为代码生产者，而是混合人机团队的工程领导者。你不再只是写代码，而是在设计系统、塑造能力、审查证据，并做出

任何 AI 都无法代劳的决策：关乎价值判断、权衡取舍，以及在具体语境中“足够好”与“真正完成”的界定。你的代码产量或许会下降，但你的杠杆效应将呈指数级增长。

11.6 致技术领导者：搭建平台，而非祈祷奇迹

技术领导者们，请认清一点：我们讨论的绝非简单地换个新工具，而是要对整个软件工程体系进行根本性重构。这涉及你使用的编程语言、需求构建方式、变更评审流程、集成管理方法，乃至大规模建立信任的机制。本书所涵盖的维度只是必要的第一步，后头还有更多挑战。例如，在一个每天可能部署数百个 AI 生成变更的世界里，部署与发布工程就需要彻底重新思考。

软件工程的存在，是为了让 99

你的职责是构建平台，让可信赖的智能体软件工程成为阻力最小的路径。别让团队一切从零开始。主动铺路：提供经批准的任务简报范式、可复用的指导包、自动化的证据收集机制、以及区隔人机关注点的工作台界面。软件工程 3.0 时代的平台工程，远不止部署流水线；更是打造一套让信任成本天然低廉的基础设施。

你还必须仔细斟酌何处需要规则，何处规则反成桎梏。并非万事都需僵化流程。探索和原型设计需要自由呼吸的空间。但当工作从探索转入工程阶段时，脚手架必须就位。你的角色是让这种过渡无缝衔接，使团队无需在速度与安全间艰难抉择。平台应当强制执行关键约束，同时为创造力与判断力留足余地。

这也是一场披着技术外衣的人力资源转型。你改变的不仅是工具，更是人们的工作方式、职责边界和价值证明途径。部分开发者将在这个新天地里如鱼得水，另一些人则可能

在从生产者向编排者转型的过程中倍感挣扎。你的平台必须兼顾二者，为初学者提供辅助轮，为准备好展翅高飞者提供高阶能力。目标不是让任何人掉队，而是把每个人的能力提升到新高度。

记住：若你袖手旁观，指望开发者各自为战，就等于逼他们在孤立中重复解决相同问题。这不是工程，这是浪费。构建共享平台、建立统一范式、创建反馈闭环、并衡量真正重要的指标。你的开发者需要你成为新世界的平台架构师，而非一个被动旁观、只能祈祷好运的看客。

11.7 致业务领导者：一场结构性的跃迁

业务领导者们，请认识到这绝非渐进式改良，而是软件构建方式和组织运营模式的结构性跃迁。软件工程已深深嵌入组织内无数反馈循环中，从产品开发到客户支持再到战略规划。当软件生产的经济学基础发生根本性变化时，每一个反馈循环都必须被重新审视，甚至彻底重构。

软件生产不再是你的瓶颈，也不再是方便的替罪羊。当一个配备 AI 队友的合格团队能在过去争论一个方案的时间里，原型化出十个解决方案时，你就不能再把延误归咎于工程部门。瓶颈已转移至决策层：决定构建什么、为谁构建、在何种约束下构建、以及达到何种标准。这些决策瓶颈正是你现在必须优化的。好消息是，AI 队友擅长生成选项、分析权衡和模拟结果。如果你重构决策流程以善用这些能力，你的决策过程就能更快、更明智。

这需要重新思考你的整个运营框架。智能体软件工程运营框架不止关乎工程，更关乎当创造变得廉价而判断变得珍贵时，你的业务如何运转。你需要新的工作评审节奏、新的证据标准、新的升级流程以及新的成功度量指标。当团队能在几天内构建并抛弃完整功能时，传统的季度规划周期

便告失效。当开发成本结构发生根本变化时，年度预算流程也显得荒诞不经。

如果你不主动管理，信任债务将拖垮你。当组织交付变更的速度快于其能证明合理性的速度时，信任债务便开始累积。它会悄无声息地利滚利，直到某天你突然无法解释系统在做什么、为何这样做、或如何走到当前状态。这不是技术问题，而是关乎存亡的业务风险。当监管机构、审计师或客户要求解释时，“是 AI 写的”绝非可接受的答案。你需要证据链、决策记录和可追溯性，为每一次变更提供辩护依据。

最终胜出的组织，不会是那些拥有最多 AI 许可证或能做出最炫演示的，而是那些构建了最可信赖、最具扩展性的软件工程系统的组织。这意味着投资于平台能力、建立清晰的治理体系、创建将事件转化为系统性改进的反馈循环，并构建让转型可持续的三层赋能框架。是的，这需要投入。但另一种选择，是被那些掌握了在信任前提下驾驭机器速度的竞争对手彻底颠覆。

你还必须为这场转型中“人”的一面做好准备。部分角色将实质改变或消失，新角色则会涌现。软件工程 2.0 时代让人有价值的能力，不会直接平移至 3.0 时代。这并非冷酷，而是现实。你的工作是以清晰且支持的态度管理这场转变，帮助人们在新时代找到自己的位置，而非假装一切照旧。

11.8 致软件工程教育者与研究者：你们肩负关键使命

软件工程的教育者与研究者们，你们或许在这场转型中扮演着最关键的角色。今天你教授的学生，将踏入一个 AI 队友不是可选配件而是基本合作者的职场。你现在进行的研究，将决定我们是基于坚实工程原则构建未来，还是在试错中跌跌撞撞。

我们必须直面那个老生常谈的质疑：你会看到不少研究声称，使用智能体软件工程的学生不会编码、逻辑理解有限。我的挑战很简单：这真的要紧吗？世界历来在向前发展。我们已不知现代编译器如何生成二进制文件，也读不懂它们，但行业照样蓬勃发展。Java 问世时，批评者哀叹指针运算和手动内存管理的消亡。我们挺过来了，学会了新技能。如今在特定场景下，Java 应用甚至能超越 C 应用，因为我们将注意力移到了更高层的技术栈。当然，总有一小部分专家能读汇编、玩转 C 指针——他们会一直存在——但行业的绝大多数将在更高层次上运作。

在软件工程 3.0 中，代码就是新的二进制文件。依赖人工从代码中找错（无论 AI 写的还是人写的）从来就是个败局。多年来我们都明白，唯手熟尔，练得越多越精通。对教育者而言，更深层的问题是：我们究竟希望学生精通什么？是死磕语法，还是成为能调度整支 AI 队友舰队的技术领导者？

软件工程课程需要根本性重构。当 AI 能完美处理语法时，再教语法和编译器错误已远远不够。你的学生需要学习系统思维、意图规范与证据评估。他们从一开始就应理解人本软件工程与智本软件工程，这不是什么高级议题，而是现代软件工程的基本二元性。他们必须学会在架构层面思考、精准描述意图并批判性评估证据。

在这个新世界里，研究机会俯拾皆是，但必须超越基础层面。既然软件本就带有随机性，如今真正的研究挑战在于规模。我们如何形式化验证由大量非确定性行动者构建的系统？如何度量与管理信任债务？哪些人机协作模式能产生最佳结果？如何教会 AI 队友理解组织文化与约束？你们的严谨能将炒作与工程现实区分开，为实践者提供负责任构建所需的基础。

如果有一项任务我希望你从本书中带走，那就是：这个周末就装一套智能体工具链。用它从头开发一个完整、可用的应用程序。然后，强迫自己每半年重做一次。唯有如此，

你才能真正体会到世界演进的速度有多疯狂，以及你原有假设过时的速度有多快。

切勿浮于表面预判这场转型。看似一切未变，实则天地翻覆。你的价值不再取决于单打独斗的能力，而在于你驾驭混合团队的效率。这意味着要将协作、指导、循证推理和系统思维作为核心技能来传授，而非选修课。那些掌握软件工程 3.0 原则的毕业生将占尽先机；那些停留在 2.0 时代的人将被无情抛下。你有能力为他们铺就成功之路，或让他们在扑面而来的现实面前措手不及。

11.9 最终抉择

我们之所以用《黑客帝国》开篇，正是因为这个比喻精准道破了你的真实处境。关于这场转型，有个根本真相是任何书籍都无法说透的：和 AI 队友共事这件事，非得亲自上手才能体会。言语终究无法填平这道体验的鸿沟。

无论多么生动的描述，都难以捕捉那样的瞬间：当 AI 队友突然领会你的架构意图，生成那个你苦思冥想却难以言表的抽象概念时；或是它一口气抛出五个你从未想到的替代方案，每个都条分缕析、权衡利弊时。你读再多关于从对抗工具到指挥团队、从编写代码到系统工程、从按部就班到并行探索的转变，也无法真正体会其中奥妙。这事儿光靠脑子想不明白，非得亲身试过才肯信。

但这里有个陷阱：只体验原始力量却不用工程方法加以约束，恰恰是灾难的开始。牛排诱人，那种以为不用改造工程体系就能直接套用智能体工具的错觉更迷人。AI 几分钟内搭出完整功能的演示令人沉醉。但工程从来无关舒适或演示。工程关乎现实：构建能够规模化、可持续运行的系统，在种种不完美中依然产出可信的结果。

选蓝色药丸，你会把智能体工具链仅仅视为一次工具升级。沿用原有的激励方式、审批惯例和“完工”标准。于是产出

暴增，信任债台高筑，直到某天醒来，面对无人能解的系统、无人能证其合理的变更，以及无人能挡的事故。你会怪罪 AI，换掉供应商，然后退回软件工程 2.0 的老路。只不过，那些选了红色药丸的对手早已绝尘而去，你连追都追不上。

选红色药丸，你会把 AI 队友看作一类全新的行动者，并为此重构整个软件工程体系。搭建三层赋能框架，铺设证据轨道，重设决策流程，只在系统证明安全时才下放自主权。起初，用令人别扭的规则取代一团和气的混乱，你甚至会感觉更慢。但总有一天，你会发现团队正以空前自信交付着更多价值。那时你的法拉利已在赛道飞驰，别人还在马厩里争论要不要出门。

选择在你，但请明白：开弓没有回头箭。能力已然就位，经济大势不可挡，谁能驾驭这场转型，谁就将定义软件的下一个时代。你可以参与定义，也可以被其颠覆。警告已经发出，装备也已备齐。现在，选择权在你手中。

选药丸吧。但要选得聪明。

后记

无限可能的重量

你好！我是 Ahmed。书写完了，但有些话不吐不快。在你合上这本书之前，有几件事想和你聊聊。

如果你读到了这里，说明你已经吞下了那颗“红色药丸”，构建了自己的工程系统，学会了如何驾驭这辆法拉利高速飞驰。你明白了软件工程从来就不仅仅是写代码；AI 队友是行动者，不是工具；可信赖需要亲手设计和构建，不是天上掉下来的馅饼。现在，你已是全副武装。从某种意义上说，你已经变得“危险”了。

但这篇后记跟这些都无关。

这篇后记是关于你的。是那个坐在方向盘后面的人。是那个即将合上这本书、打开电脑、然后猛然意识到自己几乎无所不能的人。我想和你聊聊接下来会发生什么，因为这事没人会告诉你。

顿悟时刻

我们在开篇说过，和 AI 队友协作就像第一次看见色彩。没体验过的人，你无法向他解释。这个比喻很贴切，但它并不完整。因为真正看见色彩的那一刻，其实是应接不暇的：视野里每一寸表面、每一道阴影、每一个物体，都突然涌来前所未有的信息。获得一个新感官，世界不会变简单。它只会更丰富、更嘈杂，对你的注意力索求无度。

《蜘蛛侠》第一部里有个场景，比任何商业书籍都更精准地捕捉了这种状态。被蜘蛛咬后的清晨，彼得 · 帕克醒来，感

官被调到了最大档。他能看清天花板上每一条裂缝，听见三个房间外的对话，感受到床单上每一根纤维的纹理。他跌跌撞撞走进走廊，差点把整条走廊拆了。他变强了，但那种强大带来的不是力量感，而是混乱。力量先于驾驭力量的智慧降临。

这正是你现在的处境。

读完这本书，你知道了如何把意图拆解成任务简报，用指导包塑造 AI 的行为，自信地运行并行智能体，把机器速度的输出驯化成可信赖的软件，再搭建起让这一切能规模化的工程系统。你学会的这些本事，不是技巧，也不是黑客手段。它们从根上改变了你对编排、证据、信任、委托和系统的理解。老实说，它们能用在软件工程之外的地方。任何一个掌握了精确表达意图、管理随机性贡献者、要求证据而不是氛围、从并不完美的部件里构建出可靠结果的人，学到的东西都可以迁移到领导力、运营、研究和生活里。

但这能力有它的代价。我想帮你做的准备，就是这个。

“慢”的馈赠

几十年来，软件工程一直自带一个调节器。一个没人设计过、也从没人感谢过的隐形限速器。它就是做成一件事的巨大摩擦力：等构建，等同事审完代码，等你自己的脑子想出那个对的抽象。一行行敲代码，编译，失败，重来。这种摩擦很烦人，我们整个职业生涯都在试图消除它。

但也是这种摩擦，一直在保护我们。

延迟、缓冲、人为的等待，这些不只是低效。它们是对我们认知负荷的阻尼。做一件事所需的时间，其实也是你的大脑用来吸收、处理、在两次发力之间休息的时间。你从没注意过，因为休息本来就和劳作交织在一起。等构建的时候，你的潜意识在悄悄消化你刚做的决定；等别人审查、反馈、回复的时候，你脑海的角落里，一个更好的设计正在成形。

缓慢不是缺陷。它是人类在以人的速度构建的系统里工作的一个特性。

而智能体软件工程，几乎把这层阻尼完全拆掉了。

当你的 AI 队友几分钟就能交付出成果，当你同时跑五个智能体、高速地审查它们的产出，当你任何想法都能在咖啡凉透之前被做成原型——阻尼消失了，限速器不见了。剩下的，是一个从未被设计成以这种速度运转的人类神经系统，连着一条正以机器节奏狂奔的生产线。

这不是理论上的担忧。这是第一代智能体软件工程师正在经历的日常，我把自己也算在其中。

“快”的陷阱

我和每一位顶尖的智能体软件工程师聊过，都发现了同样的模式：没有人用 AI 更快地干完同样的活然后提前回家。相反，他们用更快的节奏干更多的活，承担更广的任务范围，把工作延伸到一天里更多的时间。没人逼他们。他们自愿的，因为 AI 让“做更多”这件事变得可能、可及、且有回报。

这些模式惊人地一致。人们开始揽起以前属于别人的责任，因为有了 AI 队友，任何任务的进入门槛都消失了。工作和生活的界限模糊了，因为给 AI 发个提示，感觉更像聊天而不是工作，结果人们发现自己会在午餐时、睡觉前、等孩子放学时，再扔一个任务出去。交互界面让你轻易忘了自己正在工作。多任务处理也爆炸式增长：人们同时跑着好几个 AI 线程，同时处理着好几个输出，重启旧任务，启动新任务。感觉上势如破竹。实际上，是持续不断的上下文切换，和一堆永远收不了尾、不断增长的开放线程。

这就是强化陷阱。AI 不会减少你的工作量。它拔高了“可能”的上限，而你会自己膨胀去填满它。工作量的增加是悄无声息的，认知疲劳是无形中堆积的。因为多做的那些是

自愿的，因为感觉像是兴奋的实验而不是苦差事，所以没人会注意到，直到倦怠突然袭来。

能力越大

蜘蛛侠的故事里，教训从来不是关于超能力本身，而是关于责任。电影常常忽略那个沉默的真相：责任不只是选择去帮助别人，也是选择不去做什么。彼得·帕克没有拯救纽约的每一个人，他做不到。这个角色的痛苦在于，他必须学会接受这一点，与“无限的能力仍然会遇到有限的人类”这个事实和解。

你现在处于类似的位置。这本书给了你一套能编排大规模 AI 队友的软件工程系统。你可以开发功能、修复错误、探索架构、搭建原型、编写测试、重构整个代码库，并以两年前看来简直是妄想的速度交付生产质量的软件。天空才是极限。从某种意义而言，你就是智能体软件工程界的蜘蛛侠。

但天空从来都是极限。它从来不是真正的约束。真正的约束过去是、现在也依然是——你！你的注意力，你的判断力，你在不迷失主线的前提下切换上下文的能力。你审查、批准、并最终拥有那些产出（那些产出速度远超你吸收速度的队友的产出）的能力。在智能体时代，人类仍然是所有权的归属点。我们是信任层，是批准、签字、承担责任的。而人类这个物种，并不是为了处理这种系统所能产生的决策量而设计的。

所以，更深的问题不是“你能做什么”。那个已经有答案了：你几乎什么都能做！更深的问题是：什么事值得做？以及，作为一个人，你究竟能同时管理多少个线程，而不会让判断力下降、注意力涣散、幸福感崩盘？没错，随着时间推移，你的 AI 队友会接管很多细小的决策。但这不代表你要做的决策变少了。它意味着能递到你面前的决策，会更大、后果更重、更值得你保持清醒的头脑。认知负荷不会变小，它只会更集中。

中庸之道

我自己也感受过所有这些拉扯。那种想继续、再启动一个智能体、再揽一个任务的诱惑，因为它突然变得可行了。那种“任何空闲时刻都是浪费”的感觉，因为你的 AI 队友从不睡觉，待办清单也永无止境。当你明明知道你的智能体舰队正可以推进某些事，而你却离开它们时，那隐约的内疚感。

那种感觉，就是阻尼消失之后的样子。给它起个名字很重要，因为如果你不认识它，它就会吞噬你。

我们还在智能体软件工程时代的第一阶段。工具很出色，但围绕它们的人类实践还远未成熟。我们还没搞清楚，当一个人编排着一支不知疲倦的 AI 舰队时，一天的节奏、界限、操作守则该怎么安排。最终我们会搞清楚的。我们会构建更好的协调层、更好的收件箱式（而不是中断式）的工作流、能保护我们免受信息洪流冲击、只把需要判断的东西呈上来的 AI 管家。书里的“工作台工程”实践是个开始。但在现阶段，设定自身限制的担子，大部分还得你自己挑。

这里有一个我撞了南墙才学到的简单诊断：当你发现自己正以“僵尸模式”做决策，没真正读就批准输出，靠肌肉记忆合并更改，出于反射而不是意图去想下一个提示——你已经越界了。这是你应该停下来的信号。不是继续推进，不是再做一个任务，而是停下来。那种状态下你做的决策，不只是质量低。它们是危险的，因为在智能体时代，一次草率的批准，可以以机器速度传播。

即使是法拉利也需要进站。地球上最快的赛车被设计成定期停靠，不是因为坏了，而是因为持续的性能表现需要它。停下来不是失败。它是系统的一部分。

所以，这就是我想请你合上书时带走的。没错，你拥有了非凡的新能力。没错，你学到的本事真的能改变一切。没错，你现在能做的事远超你的想象。所有这些都千真万确，你应该感受到它们的分量，也应该为之激动。

但同时：对你选择承担的事，要有策略。对你什么时候停，要有规划。保护你的休息时间。别让摩擦的消失欺骗你，让你以为自己（这个人类）也变得无摩擦了。不，你不是。你仍然需要休息、恢复、放空、空间，和任何 AI 都无法给你的缓慢处理时间。你的心理健康不是可有可无的，它是承重墙。如果它垮了，你在它上面建的一切都会跟着垮。

前路漫漫

整本书里，我一直在用“开法拉利”做比喻。工程系统让你能安全地全速前进。但就算是 F1 车手，也不会每圈都油门踩死。他们要管理轮胎，管理燃油，在两小时的比赛里管理自己的注意力。最好的车手，不是单圈最快的那个人，而是整场比赛里管好自己的资源、跑到最后还有余力的人。

你跑的是马拉松。智能体时代不是短跑，它是一整个职业生涯、一种生活方式，是和一项会持续演化几十年的技术的漫长共处。能在这个时代活得好的人，不是第一年烧得最亮的那批。是那些学会了以可持续的强度运作，知道什么时候该推、什么时候该滑行，像对待软件系统一样，用同样严谨的工程态度对待自己认知健康的人。

你拥有了工程系统，你拥有了技能，你拥有了按任何诚实标准衡量都算卓越的 AI 队友。现在，请像对待任何一个关键系统那样对待自己：监控过载，构建恢复周期，永远别忘了——你，正在读这句话的你，才是整个软件工程系统里最重要的那个组件。

照顾好自己。这场革命，需要一个健康的你。

艾哈迈德 · E · 哈桑（Ahmed E. Hassan），人类，地球

A　参考表格

这些表格，我故意设计得事无巨细。它们是**参考模式**，实施智能体软件工程时，翻翻它们会有帮助；但要理解第一章的核心心智模型，**你完全不用**把这些全背下来。权当入门指南：先用最小可行子集跑起来，再逐步完善，别想一口吃成胖子。

表 1：任务简报结构

角色：	混合体（请求驱动，受治理约束）
归属：	人写的（智能体可以打草稿，人来批准）
确定性：	内容非确定，但引用的工作流程运行手册提供了确定性检查点
可审计性：	必须注明关联的指导包/工作流程运行手册版本、事实来源，以及（恢复时）涉及的连续性数据包版本。

任务简报组成部分	为什么要有它	里面该写什么	少了它会怎样
目标和意图	在成千上万个微观决策上校准方向	目标；用户价值；明确不做什么；硬约束	活干得挺“对”，但问题压根不是这个
概念性计划	防止走错路、早期瞎折腾，但又不用手把手地管	策略；检查点/里程碑；什么情况要踩刹车（“慢速模式”），以及决策边界（别列那种一碰就碎的步骤清单）	脑子里的第一个方案直接胜出；等发现要推倒重来时，已经晚了

续下页

（续）

任务简报组成部分	为什么要有它	里面该写什么	少了它会怎样
成功标准	把意图翻译成“怎样才算干完”	做哪些、不做哪些；验收标准（最好写成属性或不变量形式）；前置条件；后置条件；预算（延迟、成本）	“看着对”取代了“真的对”
精选上下文	既要防止胡编乱造，又别淹死智能体	关键文件/模块；架构边界；已知的坑；关联的指导包；对应的工作流程运行手册入口；示例；数据模型说明；（如果是恢复任务）链接到连续性数据包，或者写个“当前状态”小节	虚构事实；违反业务规则；在错误领域瞎折腾
实施指导	给方向，但不是手把手教	推荐的做法；哪些领域要避开；碰都不能碰的 API；兼容性、安全性约束	违反潜规则；欠下难以偿还的技术债

续下页

（续）

任务简报组成部分	为什么要有它	里面该写什么	少了它会怎样
验证计划和证据义务	把“必须验证”这事板上钉钉，且能审查	必须做的检查；测试分层；如何证明验收通过；把每个验收属性 → 检查方法 → 必须提供的证据构件串起来	审查变成找不同；正确性全靠猜
升级和权限（自主权边界）	防止闷声越界，也防止批准请求泛滥	哪些事自己说了算；哪些事必须升级（并明确找谁）；哪些事绝对禁止；何时该停；什么情况要发起咨询；能接受的风险多大	智能体悄咪咪越过红线，或者为个破事反复烦人
版本管理和汇总	保证任务随时间推移仍然可控	简报怎么增量更新；决议记录如何关联进来；规范的汇总简报怎么维护，怎么保留修改历史	“事实来源”散落在聊天记录里；交接时一头雾水；审计变成考古

表 1A：连续性数据包结构

角色：	混合体（首要任务是保连续性，为恢复提供有界状态）
归属：	通常智能体根据人定义的结构更新；人作为任务治理的一部分来拥有它
确定性：	混合型（内容可能非确定；必须的部分应是确定的）
可审计性：	必须链接到当时生效的任务简报版本、相关的决议记录，以及它引用的那些“此路不通”的存档。

连续性数据包组成部分	为什么要有它	里面该写什么	少了它会怎样
当前状态快照	为任务恢复存个稳定的工作集	目前已知为真的事；正在做的事；哪些算“已完成”；哪些还挂着	新会话进来，又把显而易见的东西推演一遍；进度归零
决策和约束	防止重复发明轮子，防止自相矛盾	已经拍板的关键决策；新发现的约束；附上批准这些决策的决议记录链接	同样的争论反复上演；无意中违反约束

续下页

（续）

连续性数据包组成部分	为什么要有它	里面该写什么	少了它会怎样
开放问题和后续步骤	保持势头，避免无谓地兜圈子	接下来最小的一步是什么；已知的未知；下一步该去哪找证据	新会话跑偏，开始探索错误领域
死胡同和已拒绝的方法	避免以后换了人又信心满满地去踩坑	试过什么；为啥失败；相关日志/基准测试/分支的链接	系统反复为昂贵的探索买单
证据指针	让连续性可审计，又不用把原始日志全转储出来	目前已产出的关键证据构件（测试、跟踪、基准测试、扫描）的链接	审查需要考古；证据淹没在聊天记录里
交接元数据	让交接可路由、责任可追溯	时间戳/重置原因（交接/压缩/换智能体）；责任人；风险说明；如果有待处理的咨询，给个路由提示	没人知道下一步该谁动，或者工作为什么暂停

表 2：指导包结构

角色：	治理（用来塑造能力、判断质量）
归属：	人写的（智能体可以提议更新，人来批准）
确定性：	非确定（可能带条件）
可审计性：	变更必须经过审查、受版本控制，并且通过任务简报/工作流程运行手册引用时，必须能链接回去。

指导包组成部分	记录了什么	实现了什么功能	典型内容
地图	系统长什么样，变更该归到哪	上手更快；少走弯路	系统概述；领域说明；架构图；关键模块和边界；UI 界面；技术栈；智能体扮演的角色
工程意图	说明“为什么”以及哪些是雷打不动的	做出更好的权衡；少违反原则	目标；非功能需求优先级；设计原理；明确不做哪些；碰都不能碰的边界

续下页

（续）

指导包组成部分	记录了什么	实现了什么功能	典型内容
操作手册	这儿的工作怎么干	交付可重复；少折腾	环境搭建；构建和运行；测试期望；调试实践；CI 和发布流程；约定俗成的规矩；模式和示例
上下文工程规则	上下文怎么加载、怎么排优先级、怎么保持连贯	减少偏离；减少矛盾；恢复更顺	上下文预算；优先级语义（不变量 vs 偏好）；检索规则；隔离策略；压缩可见性规则；矛盾处理和升级
治理与安全	谁说了算、权限多大、何时升级	正确升级；风险可控	责任人；批准门槛；咨询协议；敏感信息处理；高危操作；敏感变更需要什么证据
生命周期和引用	事实如何保持为真	防止腐化；保存上下文	状态；版本历史；已废弃内容；跨组织层级的层次/优先级；退役规则；指向 API、运行手册、示例的链接

表 3：工作流程运行手册结构

角色：	治理（工作流程主干和编排）
归属：	人写的（智能体可提议）
确定性：	触发和关卡是确定的，执行过程则混合了确定与非确定
可审计性：	必须支持追溯（触发了什么、为什么、运行了什么），并链接输出。

工作流程运行手册组成部分	为什么要有它	里面该写什么	少了它会怎样
入口门控	防止还没准备好就启动	必须加载什么上下文；环境基线检查	智能体盲目启动；瞎折腾；违反基线要求
阶段和门控	让执行过程可审计	划分阶段，每个阶段有检查点和停止规则	工作成了黑盒

续下页

（续）

工作流程运行手册组成部分	为什么要有它	里面该写什么	少了它会怎样
确定性检查	强制遵守那些没得商量的合规要求	测试、扫描、代码检查、构建步骤、必须触发的自动化	“指导意见”在压力下被跳过
探索策略	要求动手前先动脑，想清楚了再干	要比较几个备选方案；比较标准；怎么决策	第一个方案就上线；系统上线也脆弱
分解和所有权协议	防止职能重叠，责任不清	工作怎么拆；所有权边界在哪；接口什么期望；怎么交接	并行工作互相打架；出了事没人认领
升级触发器	防止闷声越界	什么情况需要咨询；到哪必须停；需要附带什么路由元数据	智能体悄咪咪越权，或者等到没法收拾了才升级

续下页

（续）

工作流程运行手册组成部分	为什么要有它	里面该写什么	少了它会怎样
必需输出	让下游审查省点劲	必须生成和链接哪些包（咨询请求包、合并就绪包、连续性数据包更新等）	审查又变成找不同
分层就绪和集成调度	团队规模大了，集成也得讲道理	定义（代码完成 vs 合并就绪 vs 集成就绪）；排序规则；时间约束	合并就绪的任务堆积如山；不安全的代码落地
计划台账和分歧规则	让“审计路径”能真正落地	要求记录：计划是什么、实际执行了什么、哪里有出入、为什么，附上理由链接；定义什么情况下分歧需要批准	可疑的捷径被隐藏；合规变成讲故事
命令和触发器	强制全局策略，不依赖个人记性	确定性触发器（“当 X 发生时”）；定义好的命令主干；永远适用的触发器；明确谁可用（仅人类、仅智能体、两者皆可）	策略看个人习惯；执行起来五花八门

续下页

（续）

工作流程运行手册组成部分	为什么要有它	里面该写什么	少了它会怎样
可追溯性和可解释性	让合规性能被调试	证明命令/触发器运行了（或没运行）、为什么、执行了什么；链接到日志/跟踪	信任崩塌，变成故事会；冲突起来没法解决

表 4：咨询请求包结构

角色：	治理（用于升级交接）
归属：	触达升级边界时，由智能体生成
确定性：	触发应是确定的（由门控/触发器决定），内容可能非确定
可审计性：	必须注明触发点并链接证据；必须能路由（带 ID、目标角色、风险级别、生命周期状态）。

咨询组成部分	为什么要有它	里面该写什么	少了它会怎样
路由和数据包元数据	让升级能被路由和追踪	稳定的标识符；当前责任人；目标角色；风险级别/紧急程度；生命周期状态；链接到任务简报 + 连续性数据包（如果相关）	咨询请求丢失、送错人，或者没人负责、扯皮不清
决策陈述	让咨询问题一目了然	一句话讲清楚需要决策什么	咨询变得含糊不清，效率低下

续下页

（续）

咨询组成部分	为什么要有它	里面该写什么	少了它会怎样
最小上下文	避免让人从头开始重建	相关的简报摘录；约束条件；受影响的模块	人把时间浪费在重现问题上
选项和权衡	辅助判断，而不只是来盖个章	2 – 3 个选项；各自利弊；关键因素（风险、成本、时间线）	决策全凭猜
证据包	把决策锚定在事实上	基准测试；日志；测试；复现步骤；数据量估算	谁嗓门大谁有理；风险被掩盖
影响范围和回滚	确保安全是明摆着的	可能破坏什么；影响谁；怎么回滚	线上出事了才傻眼
建议和问题	推动事情往前走	推荐哪个选项及理由；一个清晰明确的问题	议而不决，决而不行，没人负责

表 4A：决议记录结构

角色：	治理（永久记录决策）
归属：	按风险分层（智能体可起草；风险越高，越需要人来批准）
确定性：	结构化字段应是确定的；理由可以叙述
可审计性：	必须链接到触发它的包（一个或多个）、相关的任务简报版本，以及所依据的证据。

决议记录组成部分	为什么要有它	里面该写什么	少了它会怎样
决策与结果	给事情画个句号	一句话说清决策了什么；批准了哪个选项；批准的范围是什么	工作继续在模糊中推进；以后又要重新争论
理由与权衡	保留“为什么这么干”	为什么选这个；关键取舍是什么；否掉了什么，为什么	同样的错误反复犯；原始意图丢失

续下页

（续）

决议记录组成部分	为什么要有它	里面该写什么	少了它会怎样
约束与后续规则	把一次决策变成持久的护栏	新增的约束/不变量；操作限制；对未来工作“必须/应该”的要求	决策无法改变未来的行为
批准与授权	让治理关系明明白白	谁批准的；他的角色/权限；风险等级；批准时间	责任没了；审计过不了
链接的工件与证据	让决策可回溯	链接到咨询请求包和/或合并就绪包；引用的证据构件；工具输出	决策变成聊天传说；无迹可寻
反馈到系统	防止学到的东西只留在局部	必须更新哪些内容：指导包变更；工作流程运行手册关卡变更；任务简报模板更新	学到的东西转瞬即逝；系统不改进

续下页

（续）

决议记录组成部分	为什么要有它	里面该写什么	少了它会怎样
替代（Supersession）与历史	防止过时信息继续误导	状态：活跃/已替代；链接到替代它的决议记录	旧的决策还在被错误地使用

表 5：合并就绪包结构

角色：	混合体（治理优先，交付证据和审计包）
归属：	在人定义的要求下，由智能体生成
确定性：	证据要求是确定的，叙述部分可以不确定
可审计性：	必须链接到任务简报、工作流程运行手册、指导包版本、决议记录以及工具输出。

合并就绪包组成部分	为什么要有它	里面该写什么	少了它会怎样
范围到证明映射	防止局部修修补补，看起来像那么回事	把任务简报里的成功标准（属性/不变量）映射到具体检查和证据上	测试全绿，但真实需求根本没满足
验证包	证明“正确”是论证出来的，不是猜的	跑了哪些测试；新增了哪些测试；边缘/失败案例怎么处理的；对应的类属性检查；相关日志	绿勾变成表演

续下页

（续）

合并就绪包组成部分	为什么要有它	里面该写什么	少了它会怎样
工程卫生证据	防止可维护性崩盘	Lint/静态分析输出；复杂度信号；变更集是否一致；代码风格是否遵守	功能跑通了，但代码成了未来的债
理由与权衡	让多年后的读者还能看懂意图	做了什么；为什么用这个方法；考虑了哪些替代方案；为什么没选	“为什么”消失了，同样的回归反复发生
计划台账	让“审计路径”成为可能	概念性计划（来自任务简报）；实际执行的计划；执行偏差图；为什么偏差，附上理由链接（以及必要的批准链接）	评审失去发现可疑捷径的能力
探索存档（渐进式披露）	防止将来再探索，又不至于淹死评审者	做了哪些实验；否掉的方法；关键测量数据；完整存档的链接；总结性的结果	团队反复为同样的探索买单

续下页

（续）

合并就绪包组成部分	为什么要有它	里面该写什么	少了它会怎样
机器可读清单	让证据完整性可被检查	一个清单，列出每个证据/探索构件，包含稳定标识符、链接、（可选的）校验和以及访问分类	证据变成无法审计的叙事；构件丢失
渐进式披露的审计追踪	让随机性工作可审计，又不淹死评审者	分层追踪：先给摘要；然后链接到确切的任务简报版本、工作流程运行手册版本、指导包版本、决议记录，以及关键工具输出/日志/跟踪	失败发生时，要么无路可退，要么面对海量日志墙不知所措
风险与发布计划	确保安全是明摆着的	影响范围；向后兼容性说明；发布步骤；相关的回滚计划；还有什么风险敞着	线上出事了才傻眼

续下页

（续）

合并就绪包组成部分	为什么要有它	里面该写什么	少了它会怎样
合规性、访问与保留说明	保证证据存得安全、共享得明白	访问控制/编辑说明；数据处理约束；按策略要保留多久	证据没法安全共享，或者过早被删了
集成就绪与时间协调	支持团队规模的落地决策	依赖项；兼容性说明；推荐的合并顺序；“时机不对”标记	合并就绪的工作堆着，但就是不敢合

B 术语表

术语	含义
智能体软件工程 (Agentic Software Engineering, Agentic SE)	一门学科，通过让整个工程系统（行动者、流程、工具、工件）都做好准备，从而从那些随机性贡献者手中产出可靠、可信的软件。
随机性贡献者 (Stochastic contributor)	有一定概率会出错的贡献者，不管是人还是 AI 队友。
AI 队友 (AI teammate)	一种随机性贡献者，在给定的约束下干活、提变更、产证据。
智能体工具链 (Agentic harness)	早期那种把模型包装进开发者工作流的工具环境（比如 Claude Code，Gemini CLI）。目前主要还是人类工作台，但趋势是向更丰富的工作台和协议演进。
行动者 (Actors)	系统里的干活的人，包括人类和 AI 队友。
流程 (Process)	可重复的工作方式，包括关卡、升级、评审这些实践。
工具 (Tools)	用来干活和验证工作的软件和基础设施。

续下页

（续）

术语	含义
工件 (Artifacts)	结构化的持久化记录，承载着意图、执行过程、证据、连续性和学习成果。它们既管非正式场合（比如头脑风暴），也管正式场合（比如审计），以此来治理人机协作。
人本软件工程 (SE4Humans)	人类模式下的软件工程：意图、治理、判断、指导、审计。
智本软件工程 (SE4Agents)	智能体模式下的软件工程：在工程约束下，以机器的规模干活。
人类工作台 (Human Workbench)	一个环境，通过结构化工件和结构化的“人类信箱”，来支持人类的判断、评审、治理和审计工作。
智能体工作台 (Agent Workbench)	一个环境，通过确定性的反馈、稳定的接口，以及结构化的输入输出，来支持 AI 队友干活。
人类信箱 (Human mailbox)	一个队列，里面是需要人类判断的结构化工件：任务简报、连续性数据包、咨询请求包、合并就绪包、决议记录，以及需要更新的指导。
智能体信箱 (Agent mailbox)	一个结构化的队列，里面是任务和子任务，以及管理这些任务执行的工件：任务简报、工作流程运行手册、指导包、决议记录、连续性数据包，以及要求输出的包。

续下页

（续）

术语	含义
任务简报 (Mission Brief)	委派工作的基本单元：一份行动规范，用来结构化意图、概念性计划、上下文指针、验收属性、自主权边界和证据义务。角色：混合体（请求驱动，受治理约束）。
概念性计划 (Conceptual plan)	任务简报里的高层策略、检查点和慢速模式触发器；不是那种一碰就碎的步骤清单。
自主权边界 (Autonomy envelope)	明确的决策权限边界：哪些事可以自己定，哪些事必须升级（并指明升级给谁），哪些事绝对禁止。
证据义务 (Evidence obligation)	对工作收尾时“必须有什么样的证明”提出明确要求（不能停留在“看着对”）。
连续性数据包 (Continuity Pack)	一种有界的任务连续性载体。在任务重置点更新，或者在压缩时作为“当前状态”小节折叠回任务简报里。它保留可恢复的状态，并明确列出死胡同，防止以后再走回头路。
工作流程运行手册 (Workflow Runbook)	可复用的工作流程组件。通过明确的阶段和关卡来结构化任务的执行和编排，混搭确定性检查和智能体探索，并支持命令/触发器、可追溯性，以及（需要时）分层就绪。角色：治理。

续下页

（续）

术语	含义
咨询请求包 (Consultation Request Pack)	结构化的升级工件。目的是让决策在团队协作中既清晰可读，又可审计。理想情况下，它会引用触发了它的关卡/触发器，并附带路由元数据。角色：治理。
合并就绪包 (Merge-Readiness Pack)	结构化的评审包。目的是用证据（而不是靠肉眼找不同）来证明代码可以合并了。它会链接回治理工件和工具输出。可能包含计划台账、探索存档和机器可读清单。角色：混合体（治理优先，交付证据）。
计划台账 (Plan ledger)	一个评审界面，把“概念性计划”+“实际执行的计划”+“执行偏差图”+“理由链接”放在一起对比，从而支持“审计路径”。
机器可读清单 (Machine-readable manifest)	一份结构化的索引，列出了证据和探索构件（标识符、链接、适用的校验和/分类），让完整性可以被检查。
分层就绪 (Layered readiness)	大规模开发时使用的就绪阶梯：代码完成 → 合并就绪 → 集成就绪 → “时机合适，可以落地”。每一层都有排序和时间约束。

续下页

（续）

术语	含义
决议记录 (Resolution Record)	对决策或验收结果的持久记录。它链接回证明其合理的那些包，并把经验反馈到指导包、工作流程运行手册和未来的任务简报里。由谁撰写，取决于风险高低。角色：治理。
指导包 (Mentorship Pack)	可复用的指导材料。它定义了在这个地方“怎样算好”，通过可能带条件的方式来塑造判断和质量，并捕获上下文工程的规则。必要时，通过工作流程运行手册的关卡、命令或触发器来强制执行。角色：治理。

智能体软件工程

与随机性队友一起
以前所未有的规模
构建可信赖的软件

翻译：李豪