FORWARD DEPLOYED ENGINEERING

EIGHT-LESSON EDITION

FROM FUNDAMENTALS TO MASTERY

FDE
入门到精通

八堂课学会将AI送进客户现场

FDE: From Fundamentals to Mastery

Eight Lessons on Bringing AI into the Customer Environment

制作 | 高飞的电子替身

微信 | rohanjojo

来源 | AI Engineer World

CONTENTS

目录

从岗位源流、现场方法到产品复利：八堂课、课后总结、实践手册与附录。

序　为什么AI越会写代码，越需要有人走进客户现场

2026年6月30日，旧金山 Moscone West 二层的 2020 房间里，面向AI工程从业者的 AI Engineer World's Fair 单独开出了一条讨论“Forward Deployed Engineering”（前向部署工程）的议程轨道。本书收录的八场演讲，每场只有十几到二十分钟。台上的人来自不同类型的AI公司：Anthropic开发Claude模型；Factory和Cognition打造软件工程Agent；Ramp提供企业卡、费用管理与财务自动化；Sierra和Decagon建设客服Agent；Kepler建设高风险环境中的可信数据系统；Varick Agents则把Agent接入企业既有流程与系统。这里的Agent不是普通聊天机器人，而是能接收目标、调用工具，并根据反馈连续执行多步任务的AI系统。产品不同，他们遇到的难题却惊人地相似。

模型能写代码，不等于企业能得到结果。

一套演示漂亮的Agent，进入客户环境后会立即撞上现实：数据在旧系统里，流程写在人的习惯里，异常情况没人记录，成功标准彼此矛盾，安全团队不肯放行，业务人员也未必愿意改变已经用了十年的做法。于是，一个看似技术的问题，变成了产品、工程、组织、销售和变革管理的混合问题。

FDE——前向部署工程师（*Forward Deployed Engineer*）——就出现在这条缝里。

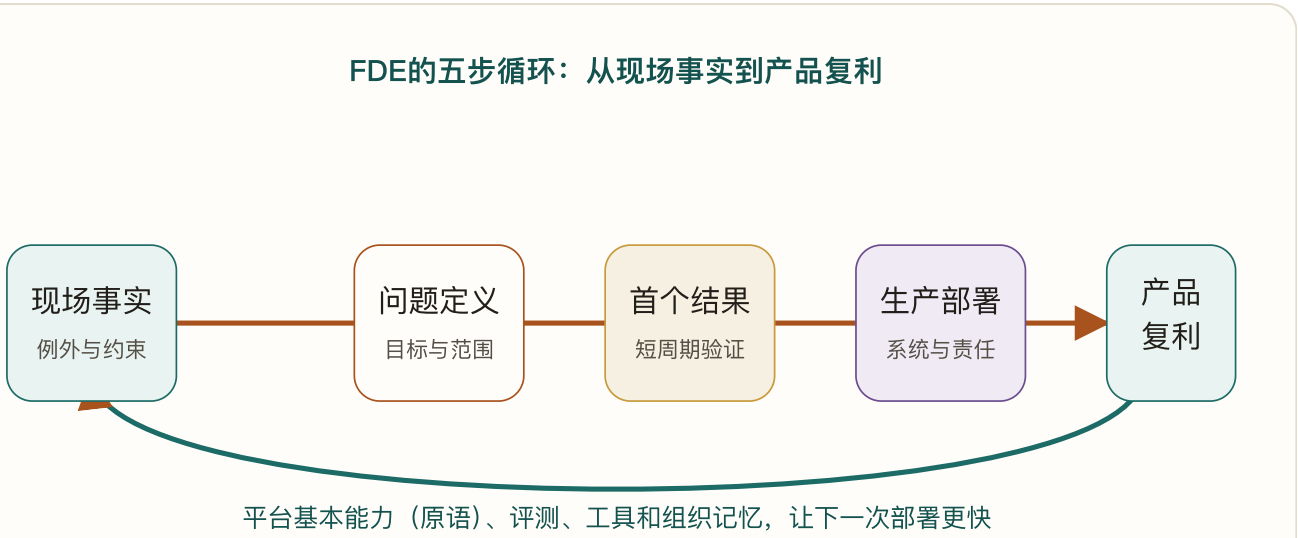
这不是一本招聘手册，也不是八家公司产品的宣传册。它想回答三个更一般的问题：

1. 为什么企业AI的瓶颈，正在从“模型会不会做”转向“公司能不能把它接进真实工作”？
2. FDE与售前、咨询顾问、解决方案架构师、产品经理、普通软件工程师究竟有什么不同？
3. 如果每个客户都不一样，如何避免FDE变成昂贵的定制开发部门，反而让每次交付都改善产品？

八位讲者并没有给出同一个答案。Sierra的 Natalie Meurer 说，FDE这个名称承载了太多不同工作，已经近乎失去意义；Kepler的 Vinoo Ganesh 则坚持，FDE从来不是一个销售角色，而是“伪装起来的产品战略”。Ramp把最稀缺的能力概括为“界定范围”；Decagon认为，AI编码越便宜，越需要克制一次性定制的冲动；Factory和Cognition则把现场部署变成了验证Agent能力、改写产品路线图的反馈系统。

这些分歧不是需要被抹平的噪声。它们正好说明：FDE不是一个固定职位描述，而是一套在不同公司里重新组合的组织能力。

全书把这套能力概括为一个循环：



真正可扩张的FDE，不以“这次客户满意了”为终点，而以“下一位客户不必重新付出同样成本”为检验。

八堂课沿着一条逐步深入的路径展开。第一课先回到企业数据与运营软件公司Palantir——FDE模式的重要历史源头——解释FDE为什么从基础设施、数据整合一路变成今天的复合角色；第二课判断什么公司真的需要这种团队。第三至第五课进入客户现场，依次学习问题发现、流程建模和范围控制。第六课处理产品化，第七课讨论结果与评测，第八课再把单个编码Agent放进完整的软件工厂。每堂课只跟随一场演讲，让讲者自己的案例和论证保持完整。

你可以把它当作一门新职业的入门课，也可以把它当作一张企业AI落地地图。最重要的不是最后要不要把团队叫作FDE，而是你的公司有没有人对这条循环负责。

关于中文译名

本书把 Forward Deployed Engineer 译为“前向部署工程师”，但正文多数时候直接使用缩写 FDE。业内也能见到“前沿部署工程师”“前线部署工程师”等译法。名称尚未稳定，恰好呼应本书的核心问题：比翻译更重要的是，这个人究竟对什么结果负责。

导读　先看懂FDE

FDE究竟在补哪一道缺口

读完导读，你会知道

- 为什么“软件已经交付”与“客户得到结果”是两件事；
- FDE与售前、咨询、实施、产品工程的区别；
- 判断一家公司是否需要FDE的五个问题；
- 为什么FDE真正交付的是一条学习闭环，而不只是代码。

从一个看似简单的请求开始

一家大型企业说：“请把你们的AI接进我们的财务流程。”

这句话无法直接写成代码。你至少还要问：是哪一段财务流程？谁拥有这项流程？系统记录在 SAP、NetSuite 等企业资源计划（ERP）系统里，还是 Salesforce 这样的客户关系管理（CRM）系统里，抑或一张几十年历史的表格里？正常路径如何走？发票与采购单对不上时，谁来处理？自动化出错的损失是一封邮件发错，还是财务报表失真？客户所说的“接入”，是一次演示、一个供五人试用的原型，还是必须通过安全审查并服务一万名员工的生产系统？

传统软件公司常把这些工作拆给不同角色：销售负责签单，售前工程师证明产品可行，实施团队做配置，咨询顾问重画流程，产品经理收集需求，工程师写代码，客户成功经理追踪采用率。拆分本身没有错。问题在于，复杂AI部署中的关键知识会在这些交接缝隙里丢失。

FDE的基本设计，是让同一名技术人员或同一个小团队穿过多道边界：从发现问题、界定范围、写出第一版方案，一直跟到生产采用；同时把现场遇到的共性问题送回产品和模型团队。

这也是为什么FDE很难用一句职位描述说清。它借用了多种工作的能力，却不等于其中任何一种。

相邻角色	通常优化什么	与FDE最容易混淆的地方	关键差别
售前/解决方案工程	证明方案能解决客户问题	都要面对客户并做技术演示	FDE通常继续拥有构建与生产结果，而不在签单后退出
实施顾问	按既定产品和方法上线	都处理配置、数据与流程	FDE更常改变产品本身，并把新模式抽象成平台能力
管理咨询	定义问题、设计流程、推动变革	都要理解业务与高层目标	FDE必须把判断落实成可运行系统，并承担工程质量
产品经理	决定做什么、为什么做	都收集需求、判断共性	FDE站在生产现场，往往同时写代码并处理部署约束
产品工程师	构建可复用产品	都要求生产级工程能力	FDE离客户更近，接受更多不完整信息和现场责任
客户成功	推动采用、续约和价值实现	都对客户结果负责	FDE用工程手段改变系统，不只是协调与培训

企业买的不是软件，是结果

个人工具可以依靠产品主导增长（*product-led growth*）：注册、试用、刷卡，用户自己判断价值。大型企业购买复杂AI系统时，常常做不到这一点。产品需要接入内部数据和权限，需要适应公司的特殊流程，需要证明在异常情况下不会造成不可接受的风险。客户真正购买的不是一串功能，而是“某项工作以后能可靠完成”。

这使软件公司承担了一个容易被低估的责任差：

> 供应商说“功能已经存在”，客户问“为什么我的业务结果还没有发生？”

FDE就工作在这段距离里。它先把抽象愿望改写成可验证结果，再让产品穿过真实环境中的阻力。

五层工作，而不是一个“超级工程师”

把八场演讲放在一起，可以把FDE的工作分成五层：

1. **事实层**：观察人怎样工作，尤其是失败、例外、绕行和没有写进文档的规则。
2. **判断层**：追问真正目标，压缩范围，决定什么先做、什么不做。

3. **工程层**：接入数据、权限与既有系统，构建、测试、上线并处理故障。
4. **产品层**：识别跨客户重复出现的模式，变成可以反复组合的平台基本能力（原语）、自助配置和标准接口。
5. **组织层**：让销售、产品、工程、安全与客户对成功标准保持同一理解，并保留连续责任。

单个人很少在五层都做到顶尖。成熟的FDE体系不是寻找一个同时拥有十年工程、六年销售和四年架构经验的神话人物，而是设计团队组合、平台与流程，让能力能够互补。

先认清书中的公司和产品

这八场演讲来自不同公司，它们所说的FDE也不是同一种工作。先把名称放回各自的业务里，后面的案例会更容易跟上。

公司/产品	它大致做什么	本书为什么谈它
Palantir / Foundry、AIP	Foundry是数据运营平台，AIP是把生成式AI接入企业数据与工作的AI平台	FDE的重要历史出处，也是多位讲者的共同经历
Sierra	帮企业在聊天、邮件、语音等渠道部署客服Agent	观察FDE角色怎样随平台成熟不断上移
Anthropic / Claude	研发Claude模型及相关AI系统	判断哪类产品和客户真正需要FDE
Kepler	面向高风险环境建设可信、可追溯的数据系统	从真实数据和用户动作中发现问题
Varick Agents	审视企业流程，再把Agent接到既有财务与业务系统	解释隐性流程和“记录系统”为什么难以绕开
Ramp	企业卡、费用管理与财务自动化平台	学习在写代码前如何界定范围
Decagon	为企业客服提供可调用后台工具的AI Agent	学习如何把定制逐步变成客户自助能力
Cognition / Devin	能读代码、改文件并运行验证的软件工程Agent	把代码生成放回企业完整的软件交付链
Factory / Droid	企业编码Agent与软件工厂平台	理解验证、权限和治理怎样决定Agent的自主上限

核心概念：反馈闭环

一次交付只解决一个客户的问题；反馈闭环让这次交付改变产品，使下一位客户更容易成功。闭环必须包含两条方向相反的信息流：产品能力走向客户，客户现场的真实约束返回产品。只有第一条，是实施；两条都存在，才是可积累的FDE。

AI为什么让这项工作更重要，而不是更不重要

AI降低了写出第一版代码的成本，却没有自动降低判断成本。相反，代码越便宜，团队越容易在错误问题上快速制造大量功能。Token是模型处理文本和代码时使用的基本计量单位；Ramp把无边界地消耗大量token、生成大量低质代码的失败讽刺为“token-maxxing slop cannon”。如果范围不清楚，Agent只是把含糊需求更快地变成难以维护的东西。

与此同时，AI让更多产品具备可塑性。过去，一个由供应商持续托管、通常按订阅使用的软件即服务（SaaS）产品，边界相对固定；今天，同一个Agent可以连接不同系统、采用不同工具、遵循不同政策、说不同品牌语言。可塑性提高了价值上限，也扩大了“能做什么”与“应该做什么”之间的空间。有人必须理解客户的业务，设计人机分工，确定评测，并承担上线责任。

因此，AI并没有消灭FDE，而是改变了它的稀缺部分：从纯粹执行，转向上下文、范围、评测、架构克制与客户责任。

你的公司需要FDE吗

先别因为职位热门就招人。可以依次回答五个问题：

1. **产品是否复杂到不能由普通用户自行配置？** 如果必须接数据、权限、工作流和模型评测，答案更可能是“是”。
2. **买家或最终用户是否缺少完成这些工程工作的能力？** 技术产品卖给技术团队，开发者关系和文档也许足够；复杂产品卖给非技术业务部门，缺口更大。
3. **单个客户的价值是否足以覆盖高触达交付？** FDE是昂贵能力，不适合低客单价、纯自助产品。
4. **公司是否拥有可复用平台？** 没有平台原语，FDE只能不断手工定制，最终变成软件外包。
5. **现场知识能否改变产品路线图？** 如果产品团队从不接收现场反馈，FDE会被困在客户与产品之间。

这五个问题中，前两个决定有没有缺口，第三个决定经济账能否成立，后两个决定这项能力能否形成复利。

最常见的误判

“客户提了很多定制需求”不自动等于“应该成立FDE团队”。有时真正的问题是产品定位混乱、销售过度承诺，或平台还没有形成稳定原语。把这些问题换一个热门职位名称，不会让它们消失。

导读小结

FDE补的是“软件能力”与“客户结果”之间的责任缺口。它的价值不只是快速完成一次定制，而是把事实发现、范围判断、生产工程、产品抽象和组织协调连成闭环。AI降低执行成本之后，这条闭环中的判断与责任反而更稀缺。

想一想

1. 你所在组织最近一次AI试点，卡在五层中的哪一层？
2. 如果客户提出十个需求，你会用什么证据决定先做哪一个？
3. 一次客户定制要满足什么条件，才值得进入核心产品？

第一课　FDE“不存在”：一份不断叠加的职位史

这堂课要讲清

- FDE在Palantir现场如何经历四次工作重心变化；
- 为什么新能力往往叠加在旧责任上，而不是替代旧责任；
- Natalie Meurer 所说的“FDE已死”真正指什么；
- 为什么连续的客户责任，比职位名称更值得保留。

谁在讲

Natalie Meurer 在 Sierra 负责 Agent Engineering，此前曾在 Palantir 从事执法与国防领域的系统工作。Sierra为企业客服提供AI Agent平台，Agent可跨聊天、邮件、短信和语音等渠道回答问题并执行业务动作。Meurer毕业于 Georgetown University，后来就读 Stanford GSB。她讲的不是一份标准职位说明，而是一段亲历者视角的岗位演化史。

本课对应演讲：The Dirty Secret of Forward Deployed Engineering。

先补背景：为什么故事要从Palantir讲起

Palantir早期提供的不是注册账号就能使用的普通SaaS，而是一套要进入政府、国防、金融和大型企业复杂数据环境的平台。客户的数据分散在不同系统里，权限、格式和质量各不相同，许多部署还位于本地机房或隔离网络。产品能否安装、能否稳定运行、能否把数据接进来，本身就是交付的一部分。

“Forward deployed”最初因而接近字面意思：工程师被派到软件真正运行的地方。后来基础设施、数据平台和应用工具越来越成熟，现场工作的重心不断上移。这堂课里的2008、2012、2016和2020，更适合看成四种主要瓶颈，而不是精确的公司编年表。

一个名称，装进了太多工作

Meurer开场抛出“肮脏的秘密”：FDE并不存在。

她并不是说公司里没有这些人，而是说这个名称已经被用来描述太多不同工作。两家公司都招聘FDE，可能一个人在客户机房处理基础设施，另一个人在设计行业解决方案，第三个人像产品经理一样收集反馈，第四个人负责搭建AI Agent。把他们放在一起，唯一稳定的共同点只剩“离客户很近”。

这个判断并不是旁观者对热门职位的冷嘲。Meurer自己就是这段历史的一部分。她最初在乔治城大学外交学院研究科技政策，后来学会编程，2016年进入Palantir。五年间，她先后做过隐私、基础设施以及面向执法和国防客户的现场工程。离开Palantir、读完商学院之后，她又在2024年加入Sierra的部署团队，负责把面向客户服务场景的AI Agent送进企业。

刚加入Sierra时，她甚至不喜欢“部署团队”这个名字。她觉得团队做的并不是把一个包装好的产品安装给客户，而是在对一个持续变化的AI系统承担结果责任。2024年，她写过一篇关于“Agent Engineer”的文章，试图给这种工作一个更准确的名称：它是AI工程的一个分支，却继承了FDE对客户结果的直接责任。两年后，“Agent Engineering”“Harness Engineering”等新称呼相继出现；后者指围绕模型建设上下文、工具、权限、验证和运行框架的工程工作。名称反而更多了。这段经历让她意识到，也许问题不在于还缺少一个更好的职位名称，而在于行业试图用职位名称固定一组本来就在变化的责任。

要理解这种混乱，需要回到Palantir早期。那时“forward deployed”近乎字面意思：工程师真的被部署到客户现场。客户可能使用隔离网络、本地服务器和特殊数据环境，平台是否稳定运行，本身就是交付的一部分。

Meurer用四个年代描述工作怎样变化。

2008：先让系统别倒

DevOps不是某一款工具，而是让软件开发与运维共同负责测试、发布、监控和故障恢复的一组方法；站点可靠性工程（SRE）则用软件工程方法保障系统稳定。最早的FDE很像DevOps、SRE和现场支持的混合体。软件要装进客户环境，出问题时不能把工单扔回总部等待。凌晨两点的故障，现场工程师必须理解网络、机器、发布和产品行为，把系统救回来。

Meurer展示了Palantir在2016年前后的招聘信息，其中最醒目的资格并不是某一种语言或框架，而是工作地点。工程师可能要长期待在芬兰、堪培拉或某个军事基地。之所以先写地点，是因为许多系统不是运行在供应商统一管理的云上，而是在客户自己的服务器、网络和安全边界内。远程看不到环境，标准化运维工具也未必能进入；工程师必须和软件一起去现场。

她入职后的第一个练习，是把Palantir软件部署到Amazon Web Services（AWS）云中的一台虚拟服务器——EC2实例——上。这个听起来像普通云计算作业的任务，实际上在重现早期FDE的核心处境：如果产品连稳定运行都做不到，后面所有关于数据和业务价值的讨论都没有意义。她还用一封夸张却真实感十足的客户邮件说明当时的日常——某台机器又被人误拔了电源，请工程师凌晨两点过去看看。

问题可能一点也不“高级”，但客户的业务已经停在那里，供应商不能以“这不是产品代码的问题”为理由离场。

这里形成了FDE最早的一项传统：**责任不以代码合并或安装完成为终点**。 系统没有在客户那里工作，工程就没有结束。

2012：系统稳定之后，数据成了问题

平台能运行，不代表里面有可用信息。客户的数据散在不同系统中，字段含义不一致，权限规则复杂，历史记录脏乱。FDE开始承担数据集成，并帮助客户建立本体（*ontology*）：把数据库里的表和字段，组织成业务人员认识的对象、关系与动作。

Meurer用了一个很形象的比喻：没有集成数据的数据平台，就像一座没有电影可放的电影院。建筑、座椅和放映机都在，却没有观众真正来这里的理由。

这项工作不只是把二十个数据库复制到同一个仓库。企业里的同一个概念，常常在不同系统里有不同名称和粒度：一个系统记录“客户”，另一个系统记录“账户”，第三个系统只知道合同编号；权限又决定了哪些人能看到哪些字段。数据接通之后，如果业务人员仍然只能面对表名、列名和连接语句，平台依旧没有真正进入工作流。

因此Palantir强调ontology。可以把它理解为数据与业务之间的一层共同语言：把分散表格中的记录组织成“飞机”“零件”“订单”“机构”这样的对象，说明它们之间的关系，并定义用户能够执行的动作。它既不是一份静态的数据字典，也不等同于知识图谱展示；它的价值在于让应用、权限和业务动作都围绕同一组业务对象运转。FDE之所以承担这项工作，是因为工程师既能读懂底层数据，又能在现场追问“这个字段在你们实际工作中代表什么”。

到这一阶段，FDE已经同时背着两份工作：底层平台依然要稳定，客户数据还要持续接入。新责任不是替代旧责任，而是叠加在旧责任之上。

2016：有数据之后，用户还需要解决方案

数据可用之后，下一个问题是业务人员怎样行动。Palantir当时的FDE会使用可视化应用构建工具Slate搭建仪表板和定制应用，把数据变成调查、运营和决策流程。工作重心从“平台是否可运行”，转向“这个客户能否用平台解决具体问题”。

工程师可以在Slate中把页面组件放到画布上，再把组件连接到已经集成的数据源。它和后来流行的低代码内部工具有些相似：不用从零搭建前端框架，就能迅速做出一个供特定团队使用的工作界面。由于FDE最清楚客户的数据和工作方式，自然也最适合把这些组件拼成解决方案。

但“看见数据”并不自动等于“改变决策”。Meurer特别提醒，只有读取、没有写回能力的仪表板，往往会慢慢失效。用户在屏幕上发现异常，仍要转到邮件、工单或另一个业务系统里采取行动；行动结果

又没有回到平台，数据便与真实状态逐渐分离。Palantir后来常说要从“data”走到“decision”，关键正是把观察、判断和动作连成闭环，而不是多做一块漂亮大屏。

这一步也埋下了定制债务：一个现场快速做出的方案，很容易成为客户每天依赖的生产系统。它既不像核心产品那样经过长期设计，又不可能在项目结束后随手删除。

这也是FDE历史里经常被忽略的一层：所谓“定制”并不一定是客户提出一个功能、工程师照单写代码。很多时候，客户只知道自己想更快地做出某类决定。工程师要从数据和日常操作中反推出应用应该长什么样，再判断其中哪些能力值得进入通用平台。现场项目因而既是交付，也是产品发现。

2020：让客户自己构建

Palantir的核心数据运营平台Foundry，并不是一个单独的数据库：它把数据集成、业务对象、应用与工作流置于同一套安全和治理体系内。随着Foundry逐渐成熟，FDE又开始做平台赋能：组织训练营、教客户使用工具、把反复出现的解决方案变成自助能力。目标不再是每次都由FDE亲手完成，而是让客户和更广泛的团队能在平台上自己构建。

Airbus与Palantir合作推出的航空数据平台Skywise，把飞机传感器、机队配置、维修工单、备件和航班等数据放到一个可分析的云平台中。Meurer用它说明客户自建为什么重要：对于一家拥有大量工程师、运营人员和合作伙伴的航空企业，少数Palantir FDE不可能永久承包所有数据应用。平台能否扩大影响，取决于能不能让数以千计的Airbus工程师在Foundry之上完成原本由FDE承担的工作。FDE的任务于是发生了又一次上移：不仅要交付方案，还要设计培训、模板和平台能力，让客户逐渐拥有自己的构建能力。

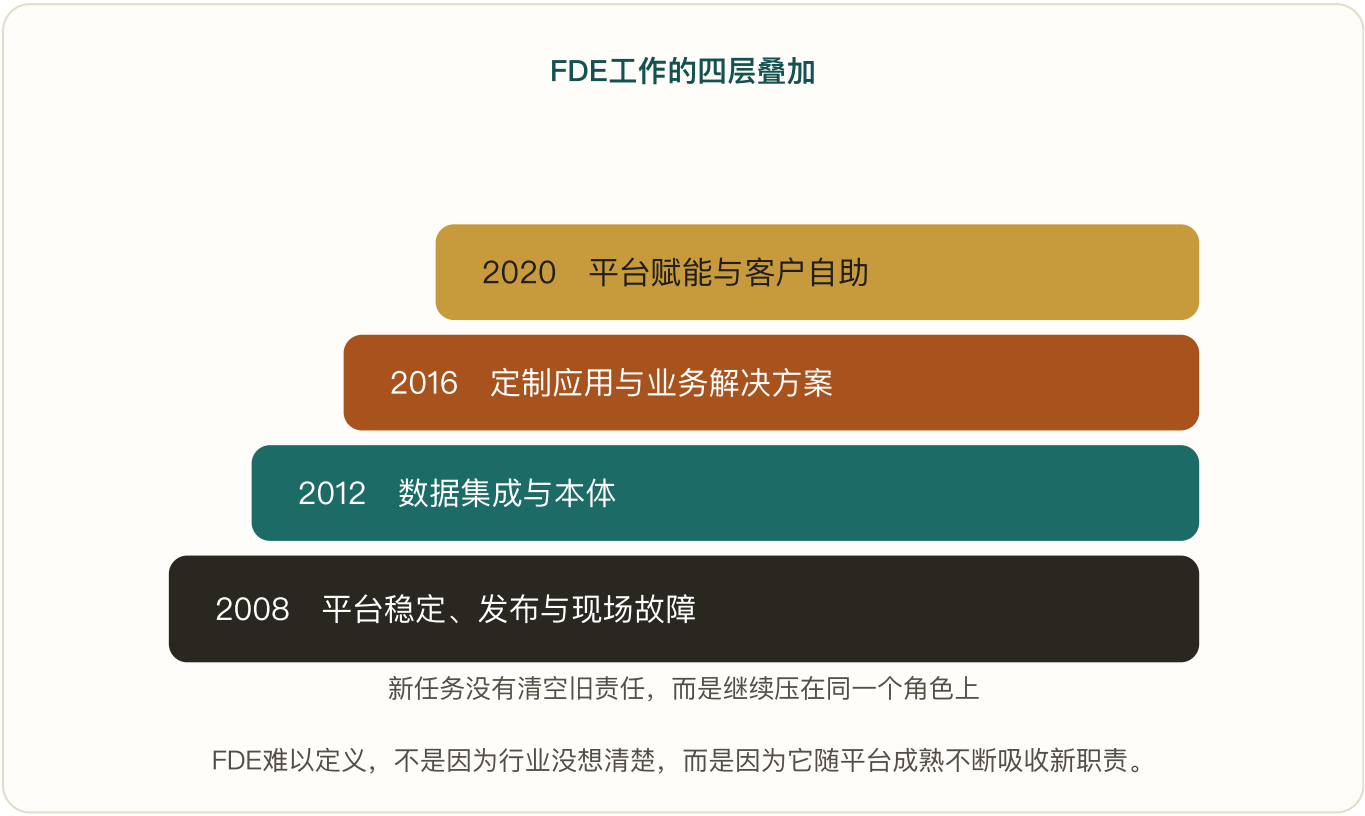
Palantir的生成式AI平台AIP用于把模型接入企业数据、工具和运营流程；它的AIP Bootcamp则是短期实战工作坊，业务与技术人员在Palantir工程师陪同下，以小时或天为单位把一个真实场景做成可运行用例。这种做法延续了平台赋能的思路。到2026年，Palantir甚至把“AI FDE”做成了产品能力，让Agent通过自然语言执行数据转换、代码仓库操作和ontology修改。这里出现了一种耐人寻味的循环：过去由FDE手工完成的部分工作，正在被平台和AI吸收；与此同时，客户开始提出更复杂、更接近经营结果的问题，又把FDE推向新的责任边界。

这四个年代不是前后替代。基础设施、数据、本体、应用、培训和产品反馈一层层叠加。到2026年，公司招聘一个FDE时，往往暗中希望同一个人同时拥有高级工程师的技术深度、销售的沟通能力和解决方案架构师的系统判断。Meurer的反讽就在这里：这样的统一职业画像并不存在。

她把这种差异称作“FDE年份”。如果一个人在平台尚不稳定的年代受训，他可能特别擅长排障、部署和在压力下恢复系统；在数据集成年代成长起来的人，更熟悉schema（数据结构定义）、权限和ontology；定制应用年代的FDE，往往有强烈的产品直觉；平台赋能年代的人，则擅长培训、抽象和组织扩散。问一位候选人“你是哪一种年份的FDE”，常常比问“你做过几年FDE”更有信息量。

这也解释了为什么Palantir校友后来创办了许多企业软件公司。现场工作把一个年轻工程师同时暴露在基础设施、数据、产品、客户、商业和组织问题中。它未必培养出某个狭窄领域最深的专家，却很容

易培养能在陌生环境中迅速建立问题模型的通才。



真正不该消失的是连续责任

Meurer认为，这个职位虽然模糊，却保留了一项重要设计：客户责任的连续性（*continuity of customer accountability*）。同一个人理解客户为什么买、系统怎样搭、用户为何不用、故障会造成什么损失，也能把这些信息带回产品团队。

当组织把售前、签约、实施、交付和支持切成多个部门时，每个部门都可能正确完成自己的局部任务，却没有人对端到端结果负责。FDE的价值，不在于一个人必须包办所有事情，而在于有人跨过交接边界，确保问题没有在组织图里消失。

这也是为什么2026年的招聘描述显得近乎荒诞。公司希望候选人既有多年staff engineer经验，又懂销售、解决方案架构、项目管理，最好还善于教学。不是因为招聘者真的相信世界上有大量这种全能人才，而是因为企业AI的交付链条尚未稳定，许多原本应该由产品、平台、售前、实施和客户成功共同承担的责任，被暂时压缩进了FDE这个名字。

因此，判断一家公司是否把FDE设计对了，不能只看招聘要求有多高。更应该看团队能否把不同责任拆开：平台团队消除重复的部署故障，数据产品提供可复用连接器，FDE聚焦问题发现和第一个生产结果，培训与客户工程帮助能力扩散。优秀的FDE组织不是依靠全能英雄维持，而是不断把英雄工作变成系统能力。

AI让边界继续溶解

当生成代码的成本下降，FDE可以更快地从需求走到完整方案，产品工程师也能更直接地面对客户。两类工程师的边界开始模糊。与此同时，软件定价也可能从席位、用量继续走向结果：如果Agent能自主完成工作，客户会更自然地问“我为什么不直接为完成的结果付费？”

代码变便宜之后，原来用于区分岗位的一条边界也变弱了。过去，产品工程师负责写可复用的核心代码，现场团队负责配置、集成和反馈，因为从零构建完整功能的成本太高。编码Agent让FDE有能力直接完成端到端原型乃至生产功能；产品工程师也能用同样的工具更快响应特定客户。两类人的工作不再由“谁能写多少代码”区分，而更多由谁掌握现场上下文、谁对长期平台负责来区分。

Meurer把另一条变化放在商业模式上。传统SaaS通常按席位收费，因为供应商提供的是使用权，客户如何使用、最终产生多少价值，很难精确归因。基础模型常按token或调用量收费，供应商能计量消耗，却仍不能保证用户得到了结果。客服Agent则可能直接完成“解决一次咨询”“促成一次销售”这样的任务，产品对结果拥有更高自主性，结果也更容易测量，于是按结果收费变得可行。

结果定价会把责任推回供应商。有人必须定义结果、度量结果，并在Agent没有交付时解释原因。这个人也许仍叫FDE，也许叫Agent Engineer、Applied AI Engineer、Customer Engineer或别的名称。

例如，“一次问题已解决”到底是Agent给出了回复，还是客户没有再次来电？退款被成功发起算不算完成，还是资金真正到账才算？一次销售由Agent促成，如何排除广告、价格变化和人工销售的影响？当收费单位从席位变成结果，这些不再只是分析团队的指标问题，而会直接进入合同、系统设计和日常运营。FDE需要把模糊的商业承诺翻译成可观察、可审计的系统事件。

Sierra的实践让Meurer修正了自己2024年的判断。Agent Engineering当然有自己的技术对象——提示、工具、状态、评测、护栏和运行时——但它不是一个与FDE平行、边界清晰的新职业。只要工程师要为客户现场的结果负责，它就仍然继承了FDE的核心逻辑。反过来，产品工程、基础设施工程和AI工程也都在更直接地接触客户反馈。不是FDE吞并了所有岗位，而是“离结果更近”正在成为更多工程岗位的共同要求。

因此，Meurer最后说：“FDE已死，FDE万岁。”死去的是把所有客户技术工作塞进同一个职位名称的幻想；留下来的，是贴近现场、承担连续责任、把经验带回产品的组织功能。

> **核心概念：岗位与能力分离**
>
> 岗位是组织图上的格子，能力是公司必须持续完成的工作。与其争论一个人是否“算FDE”，不如逐项检查：谁发现现场事实？谁界定问题？谁构建并上线？谁追踪采用？谁把共性送回产品？名称可以变，责任不能失踪。

本课小结

FDE的模糊来自历史叠加：先保证平台稳定，再集成数据、构建解决方案、赋能客户。AI又让产品与现场工程进一步重叠。与其寻找一份永恒的职位定义，不如保留端到端客户责任，并把不同能力分配给合适的人和系统。

第一课留下的悬念是：既然FDE不是一份稳定、统一的职业，企业是否还应该专门成立这种团队？答案不能靠职位热度决定。下一课会把产品复杂度、买方能力和平台经济放到同一张图里，先判断什么公司真的需要FDE。

想一想

1. 你的公司在哪一次部门交接中最容易丢失客户上下文？
2. 如果不设FDE职位，哪些责任仍必须被明确分配？
3. 结果定价会给产品和交付团队增加什么新义务？

第二课　先别招FDE：用一张二维图判断是否真的需要

这堂课要讲清

- “产品复杂度×用户技术能力”的FDE判断框架；
- 为什么平台原语是FDE模式的经济前提；
- 怎样区分一次性需求与可回流产品的共性需求；
- 如何用结对与知识迁移降低关键人风险。

谁在讲

Kevin Bai 是 Anthropic 技术团队成员（Member of Technical Staff）。Anthropic是Claude模型及相关AI系统的开发公司。Bai在演讲前曾在 Palantir 做FDE，也在人力与企业管理软件公司 Rippling 从第一名成员开始搭建FDE团队，并在一年左右扩展到约25人。他的“FDE 101”从一个很实际的问题开始：公司到底需不需要这种团队？

本课对应演讲：Forward Deployed Engineering 101。

先补背景：平台、原语与设计合作

Palantir Foundry是一套数据与应用平台。它可以把分散数据组织成客户、订单、仓库等业务对象，并在上面构造工作流；但平台不会替一家消费品公司决定怎样提高货架覆盖，也不会替一家能源企业设计设备维护流程。客户购买技术，最终却用业务结果验收，这段距离正是Bai讨论FDE的起点。

他反复使用“原语”这个词。原语是平台已经提供、可以重复组合的基础能力，例如身份、权限、数据连接、对象模型、工具调用和审计。FDE应当在原语上组装客户方案，而不是每次从空白代码库开始。这样做相当于把创业公司早期与少数客户共同打磨产品的“设计合作”，扩展到大型企业市场。

平台不会自动变成业务结果

Bai用Palantir的Foundry解释产品与结果的距离。平台可以提供数据组织、本体和应用构建能力，但企业不会因为“拥有一个平台”就自然得到结果。有人必须把客户的流程、数据和目标映射到平台上，搭出真正可运行的解决方案。

Foundry对一家企业的承诺，大致可以拆成三层。第一层是把散落在不同系统里的数据集中起来；第二层是建立ontology，把“表一、表二、表三”变成仓库、产品、设备、订单等业务人员认识的专有名词；第三层是在这些对象之上构建应用。对工程师而言，这三层已经构成一套能力很强的平台，但一位消费品公司的负责人听完以后仍会问：它能不能提高门店铺货率？能不能减少缺货？能不能让销售更快找到问题？

这不是客户“不懂技术”，而是买方与卖方衡量价值的单位不同。平台团队谈数据模型、计算和应用框架，业务负责人谈销量、吞吐量、成本和风险。客户也不应该为了证明自己配得上软件，先招聘一支平台工程团队，再花几个月学会供应商的内部概念。Bai把这部分额外投入称为客户承担的“培训税”：客户先为平台付费，再为培养会使用平台的人付费，最后才有机会构建真正想要的东西。

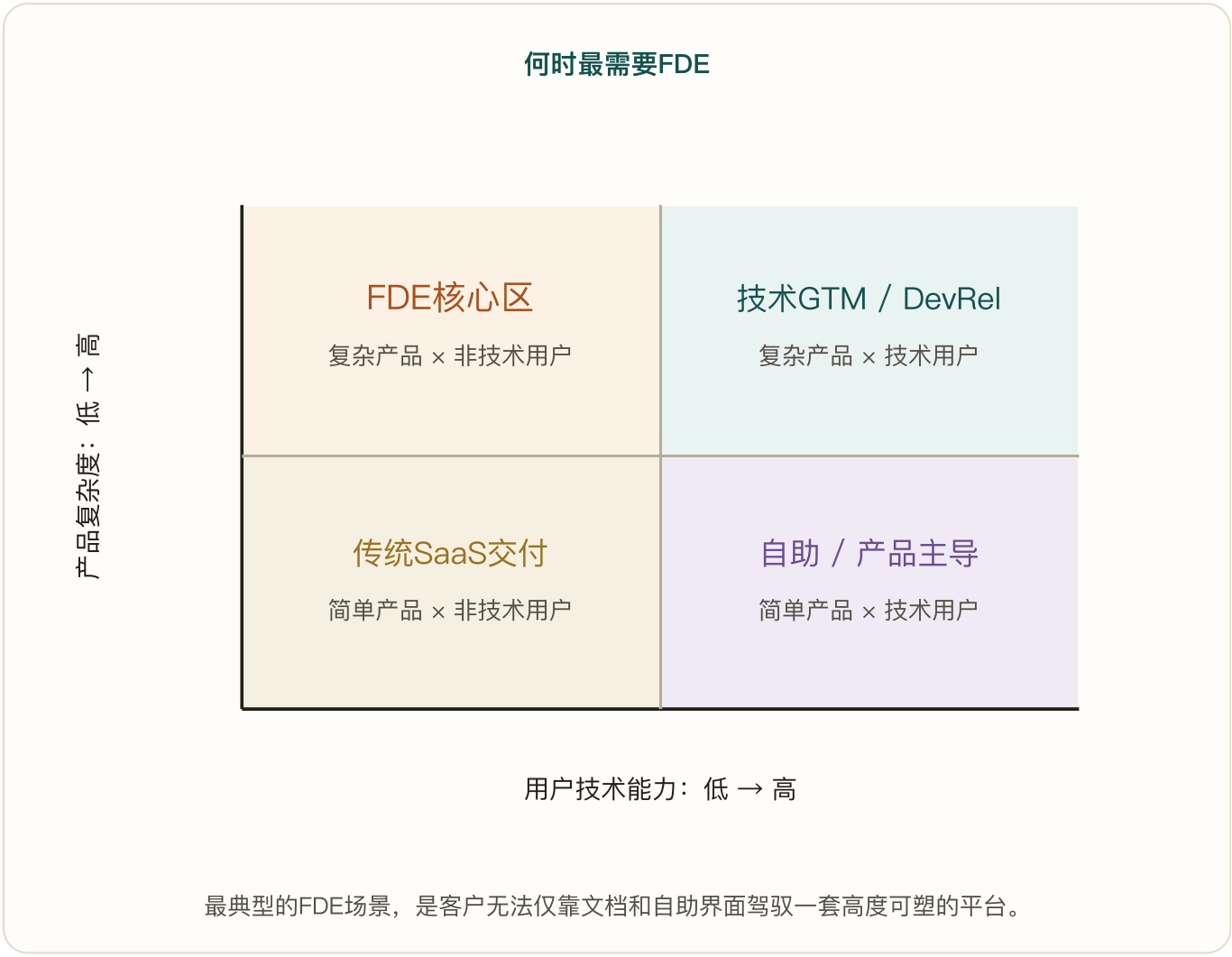
Palantir的解决办法，是不把产品和服务分开卖。客户既不是买一张软件许可证后独自研究，也不是按人天租一队咨询顾问，而是购买一个业务结果。供应商暂时“借”给客户一批已经熟悉平台的工程师，他们进入业务环境、理解问题，再用Foundry搭出方案。Bai把这种体验比作高级餐厅：顾客不需要研究厨房设备和烹饪流程，服务人员的责任是理解需要，并把最后的体验送到桌上。

这也解释了Palantir式模式与传统咨询的差别。客户既不是购买一套拿来即用的固定成品，也不是按人天购买完全从零写起的软件项目；它购买的是建立在共享平台之上的业务结果。FDE处于中间：用平台已有能力完成大部分工作，再针对客户环境做必要延伸。

Bai还用合同价值说明这种模式为什么值得付出昂贵的人力。他在演讲中引用了一组近似的公开市场比较：普通上市SaaS公司的平均客户合同价值很少达到几十万美元，而Palantir的大型客户合同可以高出一个数量级。具体数字会随统计口径和时间变化，但方向很清楚——当客户购买的是与核心经营直接相关的结果，而不只是若干账号时，合同可以覆盖高强度的工程投入。FDE不是免费附赠的“白手套服务”，而是产品定价和进入市场方式的一部分。

一张2×2的判断图

Bai提出两个维度。第一，产品本身是简单成品，还是技术复杂的平台；第二，买家和用户是否具备足够技术能力。图中的GTM是*go-to-market*，指产品怎样被客户发现、购买、部署并持续使用；DevRel是*developer relations*，即通过文档、示例、社区和技术沟通帮开发者用好产品。



最典型的FDE场景，是客户无法仅靠文档和自助界面驾驭一套高度可塑的平台。

技术产品卖给技术用户时，开发者关系、解决方案架构和清晰文档可能已经足够；简单产品卖给非技术用户，可以依靠传统SaaS实施；简单产品卖给技术用户，往往更适合自助。最需要FDE的是左上角：产品复杂，用户却不以软件工程为本职。

GTM与DevRel都能缩短产品与客户的距离，但不一定像FDE一样对客户的生产结果持续负责。

Bai给了几组便于辨认的例子。GitHub是代码托管与协作平台，Datadog是监控应用与云基础设施的可观测性平台；它们本身相当复杂，但主要用户是开发者、运维人员和CTO，他们的本职工作就包括吸收这种复杂度，供应商可以通过文档、开发者关系和解决方案架构帮助他们。Rippling主要管理人事、薪酬和企业IT，Jira用于追踪工作项与软件项目，Slack是团队消息与协作工具。它们也可能有丰富配置，但面向的是更广泛的业务用户，核心交付形态仍是可配置成品，而不是要求客户在底座上开发一个新应用。

Palantir落在最困难的象限：它把高度可塑的应用平台卖给不以软件开发见长的企业。大型科技公司本来就有强大的工程团队，未必需要外部平台替它们构建内部应用；石油、制造、消费品等行业拥有深厚业务能力，却不一定储备同等规模的软件人才。Bai开玩笑说，油气公司的“pipeline”首先是输送原料的管道，而不是数据管道。客户真正缺少的，正是把业务知识和平台能力接起来的人。

AI让更多公司进入这个象限。Agent平台越来越通用，可以连接多种工具、重写工作流、处理自然语言政策。它们“能做很多”，却因此更难直接拿来即用。银行、制造、医疗或财务用户了解自己的业务，却不一定知道怎样设计Agent、工具权限、评测和异常处理。

这也是Bai对FDE在2026年突然升温的解释。他不认为全行业只是迟了二十年才发现Palantir的做法高明。更根本的变化是，AI让几乎每一套软件都获得了某种“可编程性”：模型可以理解自然语言、调用工具、生成代码，并根据客户上下文改变行为。这里的Agentic特性，是指系统能围绕目标调用工具、连续采取多步行动，而不只回答一次问题。平台越具备Agentic特性，就越不像一件边界固定的成品；可定制空间越大，客户越难仅凭产品界面弄清它到底能为自己做什么。过去只有少数平台公司面对的交付难题，开始出现在大量AI公司身上。

没有平台，FDE会变成开发公司

二维图只回答有没有需求，还没有回答商业模式能否成立。Bai给出第二个检查：公司是否已经拥有，或愿意建设一套共享平台。

假设每次交付都从空白代码库开始。第一个客户的收入可以支付构建成本，但第二个客户提出另一套需求，团队又从头开始。随着客户增加，代码库、维护义务和人员数量线性增长，毛利被长期支持吞噬。公司名义上卖产品，实际上经营的是项目制开发。

Bai把边界说得很直白：如果每名FDE都从零开始为客户写一套独立软件，这家公司拥有的不是FDE团队，而是一家开发外包公司。开发公司完全可以盈利，问题在于它与软件平台的经济模型不同。每个新项目会带来新的代码库、部署方式和知识孤岛；几十个客户之后，工程师要同时理解几十个仓库，维护成本会先吞掉利润，再吞掉团队的耐心。

可持续模式需要“原语”（*primitives*）：认证、数据连接、工作流、策略、评测、界面组件等可组合能力。Bai举了一个示意比例：约60%的能力来自共享平台，约40%按客户场景组合和扩展。这个比例不是行业标准，重点是定制必须站在共用底座上。

“原语应该做多大”是现场问答里最实际的问题之一。做得过粗，FDE只能在一个几乎完成的应用上改颜色和字段，遇到新行业就无能为力；做得过细，工程师仍要从大量低级组件开始拼装，平台没有节约多少工作。答案取决于客户的相似程度：如果场景高度一致，平台也许可以预先完成六成甚至更多；如果客户跨度很大，就需要更细的配置和工具。

Bai用AWS作类比。企业当然可以购买机架、接入网络、自己维护硬件，但云平台把计算、存储、数据库等重复问题封装成服务。DynamoDB是AWS全托管的无服务器NoSQL数据库；NoSQL泛指不以传统关系表结构为唯一组织方式的数据库。使用者不用自己安装、打补丁和扩容数据库服务器。DynamoDB不替客户决定最终应用，却让客户无需重新发明数据库。FDE平台也需要找到自己的“DynamoDB”：它们不是最终方案，却是每个方案都会依赖、由平台统一维护的可靠积木。

这带来一个很实用的投资顺序。公司不能因为已经招到能赚钱的现场工程师，就一直推迟平台建设。越早获得客户，越容易被短期交付牵着走；如果没有明确预算把重复部分抽回平台，团队会在收入增

长的同时积累更快的维护负债。FDE人数增加不是平台成熟的替代品，反而应该让共性更快浮现。

> **核心概念：设计合作的规模化**
>
> 早期产品会与少数客户进行设计合作（*design partnership*），共同发现产品形态。FDE把这种深度合作带进大型企业，但必须用平台、方法和知识沉淀避免每个客户都成为一项独立工程。规模化的不是“少沟通”，而是“每次深度沟通都留下可复用资产”。

面向企业客户的B2B（企业对企业）创业公司在产品尚未成形时，通常会找几家设计合作伙伴。双方都承认答案尚不清楚：客户提供问题、数据和高频反馈，创业公司投入创始人和工程资源，共同找出有价值的产品。多数公司把它视为从零到一的临时阶段，等产品稳定后便转向标准销售。

Palantir更激进的假设是：为什么设计合作只能发生在创业早期？大型企业的问题同样会变化，通用平台也不可能预先知道每个行业的所有工作流。FDE就是把设计合作保持到企业规模。这里真正需要扩张的不是为每个客户永久养一支独立项目组，而是发现—交付—抽象的循环。每个新客户既贡献收入，也帮助平台看见下一组应该产品化的能力。

什么留下，什么回到产品

FDE在现场写出的东西可以分两类。

第一类真正属于客户：独有的业务规则、专用数据映射、内部审批链、品牌语言。这些内容留在客户空间里，但仍要有测试、版本和维护责任。

第二类是多个客户都会遇到的能力：某种连接器、权限模式、评测框架、配置界面或工作流原语。这类内容应该迁回核心平台，由产品团队维护。如果它长期停留在客户分支里，公司就错过了复利。

判断的难点在于，共性不一定在第一次出现时就显而易见。FDE需要记录需求形状、比较不同部署，并与产品工程保持短反馈周期。

第一次出现的需求通常只能被称作“客户特例”。第二次出现时，团队会发现两家客户使用不同术语，但底层动作相似；第三次出现时，才可能确认它是一项平台能力。过早抽象会把偶然差异固化进产品，过晚抽象则会让多支现场团队重复造轮子。FDE的产品判断，正是在这两种成本之间选择时机。

Bai在问答中给出的原则很简洁：真正只属于一个客户的东西留在客户侧，能够一般化的东西最终应进入平台。但“最终”不等于“立即”。现场代码可以先作为探针，证明问题真实存在、方案确实有效，再由核心团队决定稳定接口、兼容范围和长期维护责任。

别让客户只认识一个人

深入现场带来另一种风险：关键知识集中在一名FDE身上。这个人知道客户内部缩写、历史妥协、谁拥有批准权，也知道系统里哪些“临时”代码不能动。一旦离开，团队面对的是一片没有地图的生产环境。

Bai建议用结对方式降低风险。两个人不必全程投入相同工时，但重要会议、架构与决策至少有第二人理解；再配合共享文档、代码审查和轮换，让客户关系从个人英雄变成组织能力。

结对还有一个不那么明显的作用：阻止现场共识变成未经审视的事实。长期驻在一个客户里的FDE，很容易接受客户自己的术语和历史妥协。第二名工程师既是备份，也能追问“这真是行业必需，还是这家公司恰好如此？”这种对照正是把客户特例提炼为产品原语所需要的距离。

当然，不是所有公司都应该建立FDE团队。Bai反复强调要问“是否必须”，而不是“是否想要”。如果产品面向技术用户，可以优先建设开发者关系和开发者教育；如果是边界清楚的传统SaaS，标准销售、实施和客户成功可能更高效。只有当公司必须把复杂、可塑的技术卖给非技术买家，并且愿意建设共享平台时，FDE才是合理的进入市场方式。

他的收束非常朴素：FDE“不过是一个面向客户的软件工程师”。这句话不是把工作说轻，而是在提醒公司别用神秘光环掩盖基本要求——仍然要写可靠软件，只是这名工程师必须把客户结果一起放进工程边界。

所谓理想画像，也因此不需要神秘化。候选人首先应该达到公司对软件工程师的技术标准；其次，团队要相信他能够进入客户会议，承认不知道的事情，理解非技术语言，并在商业压力下保持工程判断。行业知识、咨询技巧和产品直觉可以继续培养，但如果公司为了“善于面对客户”而降低工程门槛，平台很快会用维护事故收回这笔账。

本课小结

复杂产品卖给非技术用户时，最容易出现FDE缺口；但高客单价和共享平台仍是经济前提。可持续的FDE把客户特有配置留在现场，把重复能力送回平台，并用结对和文档防止知识被一个人垄断。

确定公司需要FDE，只解决了组织设计问题。工程师真正进入客户之后，面对的往往不是清楚的问题，而是一份已经写好的解决方案。下一课从一次失败的数据库设计和一份47页需求文档开始，讨论怎样越过客户给出的答案，找到工作真正卡住的地方。

想一想

1. 你的产品落在二维图的哪个象限？一年后会不会移动？
2. 目前客户项目中，哪些代码其实应该迁回核心平台？
3. 如果最懂某个客户的工程师明天离开，团队能否继续交付？

第三课　客户带来的是方案，FDE要找出问题

这堂课要讲清

- 为什么客户的47页需求清单可能只对应一个行动问题；
- 怎样用一天以内的原型换取信任和现场访问权；
- 为何观察一次双击、复制粘贴或叹气，比再开一次需求会更有价值；
- 为什么“临时代码”必须按会存活18个月来设计。

谁在讲

Vinoo Ganesh 是 Kepler 首席执行官（CEO）兼联合创始人。他曾在 Palantir 工作七年，为国防与情报场景建设数据基础设施，后来成为全球投资机构 Citadel 的首任业务工程负责人（Head of Business Engineering）。Kepler官方资料把他过去十五年的主线概括为：让高风险环境中的数据系统值得信任。

本课对应演讲：How Forward Deployed Engineering is done at Kepler。

先补背景：这一课为什么会谈数据库和文件格式

Ganesh讲的“现场”并不只是与客户开会，而是让产品接触真实数据。Cassandra是一种分布式数据库，keyspace可以粗略理解为它组织数据的顶层容器；设计在干净样本上成立，遇到银行数据里的空日期后，却可能创建出数百万个结构并耗尽资源。Parquet则是一种适合大规模分析的列式文件格式，比CSV更高效，却不能像普通表格那样随手双击查看。

这些技术细节并非旁枝。Phoenix系统失败和Parquet迁移受阻，分别说明两类现场事实：数据本身比测试样本更脏，用户的工作动作也比架构图更具体。FDE要同时看见这两层。

一套在实验室里正确、在现场里崩溃的系统

Ganesh先讲了一个失败故事。Palantir曾在隔离环境中开发一个代号 Phoenix 的系统，测试表现不错；一放进真实数据，问题同时爆发。空白日期被默认成1970年1月1日，数据分区膨胀成数百万个

keyspace，Cassandra的文件句柄和内存行为让系统启动需要极其夸张的资源。

Phoenix的目标并不荒唐。团队想为大型金融客户保存海量交易记录，为了便于按时间清理旧数据，把数据切进按时间分桶的keyspace。只要日期正常，这种设计相当整齐：到了保留期限，就删除对应时间桶，不必逐行扫描。问题在于，真实银行数据里存在空日期。系统把空值回退到Unix时间起点——1970年1月1日——随后尝试为1970年至2013年之间每一个十分钟区间创建时间桶。

结果不是性能稍微下降，而是结构数量爆炸。Ganesh回忆，系统生成了大约230万个keyspace；Cassandra为相关文件句柄预留资源，启动估算需要约14TB内存。产品在设计条件下“完全正确”，到了客户数据上却几乎无法启动。这个事故尤其有启发性，因为失败不来自高深的分布式算法，而来自一个极其普通的脏数据事实：日期可能为空。

这里每个细节都指向同一个教训：产品团队以为自己在处理“典型数据”，现场却从不典型。真实世界里有缺失值、旧格式、组织留下的历史包袱，也有只有实际用户才知道的使用方式。没有现场，工程师优化的是一个想象中的客户。

团队事后才发现，真正缺少的不是又一轮市场调研，也不是更多样本，而是共同承担构建结果的人。产品工程师没有被嵌入客户环境，客户也没有进入设计循环，双方因而都无法在系统架构形成之前暴露那些看似琐碎、实际上致命的约束。

Ganesh借用 Palantir CTO Shyam Sankar 的说法，把FDE称为“伪装的产品战略”。现场不是产品完成后的交付末端，而是产品发现自己应该成为什么的地方。

这句话也是他整场演讲与其他讲者最明显的分歧。FDE后来经常被视为一种高客单价的进入市场策略：派工程师帮助客户使用复杂平台，换取更大的合同。但Ganesh回忆，Palantir在2013年前后建设Foundry时，现场工程首先是产品策略。工程师的考核重点不是促成多少销售，而是能否作为产品团队的延伸，发现新的能力机会，并把现场解法一般化。

他本人曾负责Project Frontline，一项把普通软件工程师轮训为FDE的项目。按他的演讲口径，大约350名后来分布在OpenAI、Anthropic、xAI等AI公司的工程师参加过这套训练。他在Citadel又做过类似工作，让工程师进入投资团队，围绕数据和软件帮助投资经理形成研究与交易能力。两段经历让他坚持：FDE首先要改变产品，而不只是提高交付速度。

47页需求，最后只剩一条消息

一家做运输与调度的客户交来47页需求，希望获得完整的商业智能工具、14项指标和开发环境。照单全收，团队大概要花三个月。

团队最初真的把这份文档当作需求。他们花了四个月讨论范围、页面和指标，却始终没有进入调度员每天工作的地方。后来，一名工程师因为家人恰好住在客户所在地，顺便去了现场，才问出那个改变项目的问题：“周一早上，你拿到这些信息之后，第一件事是什么？”

Ganesh没有继续讨论功能，而是问：“下周一早上，你看见这些信息之后会做什么？”

答案不是分析十四张图表，而是发现某种异常后通知一个人采取行动。于是团队先在四小时内做出一条告警消息。它不完整，却直接抵达了业务动作。客户很快能判断：这是否值得继续？阈值对不对？谁应该收到？什么情况是误报？

调度员的真实流程大致是：查看是否有车辆延误，如果延误会影响后续安排，就打电话给相关调度人员，重新安排货物或运力。47页文档描述的是一套想象中的分析系统，真正决定价值的却只有“发现延误—通知正确的人—采取替代行动”这条链。团队最后先做的不是缩小版BI（商业智能）平台——即用报表和仪表板分析经营数据的系统——而是一条能够触发行动的消息。

这个案例把需求发现从“把名词写清楚”改成“把动作追到底”。Ganesh建议连续追问三件事：

- 你真正想达到的目标是什么？
- 看到结果之后，下一步具体会发生什么？
- 今天没有这套系统时，你怎样绕过去？

如果一个版本能在一天以内完成，就先交付、观察并关闭反馈循环。速度的价值不只是提前上线，而是用真实行为替代会议中的猜测。

Ganesh甚至建议：如果问题能在一天内解决，就先不要把它上升成宏大产品战略，也不必立即召集所有产品经理讨论路线图。把最短方案交到用户手里，看看工作是否真的改变。小交付不会自动产生产品方向，但它能让团队从争论假设转向观察事实。

“谁定义问题，谁就拥有解决方案。”这句话揭示了快速交付的第二层价值。若供应商只接收客户写好的功能清单，就只能在客户设定的框架里竞争；若FDE通过现场理解重新定义问题，并用一个小结果证明判断，客户会逐渐允许他进入更重要的流程。信任不是靠善于聊天获得的，而是靠连续解决真实问题获得的。获得的现场权限越深，产品团队能看见的共性也越多。

> **核心概念：最短行动闭环**
>
> 传统MVP（minimum viable product，最小可行产品）常问“最少需要哪些功能”；最短行动闭环问“最少需要什么，才能让一个真实用户完成一次有价值的动作，并给出反馈”。前者容易做成缩水产品，后者以验证业务因果为目标。

小问题为什么通向大产品

FDE容易被“大项目”诱惑。Ganesh的经验恰好相反：先解决一个小而痛的问题，能赢得信任、访问权和更多现场信息。你越快证明自己理解工作，用户越愿意让你靠近真正关键的流程。

他讲了一个Parquet迁移案例。工程上，CSV转换成Parquet可以显著改善存储和处理效率；一位数据质量工程师却长期反对。技术团队不断解释性能，她仍然不同意。FDE到现场后才看到，她每天会双击CSV文件，快速抽查几行数据。Parquet对机器更高效，却夺走了她最重要的质量控制动作。

团队连夜做了一个简单的Parquet查看器。她第二天仍能像以前一样打开和检查数据，迁移很快获批。Ganesh称，这次改变让一条流水线的时间从约17小时缩短到约2小时。这是演讲中的案例数字，更重要的不是精确倍数，而是阻力的真正来源：反对者不是不懂技术，她是在保护一个没人注意到的工作环节。

此前，系统每天要向客户的AWS对象存储服务S3投放大约1TB数据；S3把文件和相关元数据作为“对象”，放进名为“存储桶”的容器。数量庞大的CSV文件给双方的数据管道和计算成本带来压力。工程师把迁移到Parquet视为显而易见的优化：列式格式更适合分析，压缩和读取效率也更好。那位数据质量工程师却反对了近一年，每次只说“Parquet更糟”“它不好用”。双方都在重复各自语言里的正确答案，却没有人观察她所谓“不好用”具体发生在哪里。

现场一看就明白了。她把CSV手动下载到Windows电脑，双击打开，抽查几行来确认当天数据是否正常。Parquet对计算系统更友好，但操作系统没有给她一个同样直接的查看方式。架构团队想优化17小时的数据管道，她想保住每天几分钟完成的质量检查。一个很小的查看器把两种目标重新接上，性能优化才真正成立。

工位旁边的信息密度

在访谈里，人会描述自己认为正确、正式、可以被别人理解的工作；在现场，身体会暴露真实流程。Ganesh建议观察：

- 哪一步总在复制粘贴？
- 用户何时在几个工具之间切换？
- 哪种情况会让他拿起电话，而不是继续点系统？
- 每天早上最烦的事情是什么？
- 哪一次叹气发生在什么界面之后？

他用一句带有现场感的话概括：“Residents get truth.”长期待在现场的人得到真实情况。工牌、邮箱和会议邀请，不只是行政安排，更像进入一座组织的数据采集许可。

这里的“驻场”不一定要求永远坐在客户办公室，而是要求进入工作发生的环境。Ganesh列举的观察信号极其具体：同一动作每天重复几次；内容从一个工具复制到另一个工具；用户频繁切换标签页；系统转圈时掏出手机；一提到某个步骤，对方下意识说“没办法，只能这样”。这些动作不是噪声，而是尚未被写成需求的产品机会。

文档通常记录组织希望流程如何运行，现场暴露流程实际上如何运行。会议里的人会省略自己已经习惯的绕行，也会用正式制度掩盖例外。只有拿到客户工牌、内部邮箱和真实会议邀请，工程师才能看

到问题在部门之间怎样传递。Ganesh形容这些访问权是“数据挖掘许可证”：不是让FDE随意收集信息，而是让他有机会从真实协作里建立产品判断。

他给出的起始问题很朴素：“每天早上最烦的事情是什么？”它比“你希望我们增加什么功能”更接近重复摩擦。FDE的目标，是让用户的明天与今天有所不同。只要一个重复动作被消除、一个等待被缩短，团队就获得了一条关于产品方向的高质量证据。

为客户的语言建模

不同公司会用“客户”“账户”“计费实体”“组织ID”指代看似相似、实际不同的对象。如果这些名词不先说清，数据模型、界面和指标都会不断打架。

这种混乱通常不是某个团队不专业。销售关心关系与机会，因此说“客户”；运营关心服务对象，因此说“client”；财务必须知道谁付款，因此说“billing entity”；开发者在数据库里只看见“org_id”。每一种称呼都适合局部工作，却可能在系统集成时指向不同边界。两个部门都说“活跃客户”，分母和时间窗口也可能完全不同。

FDE要和客户一起定义业务的名词与动词：有哪些对象？它们怎样关联？人可以对它们执行哪些动作？这就是本体工作。好的本体不只整理数据，也让团队逐渐用同一套语言讨论业务。用户开始自然使用产品中的概念时，产品已经进入了组织的思考方式。

Ganesh把企业拆成“名词”和“动词”。名词是客户、账户、发票、设备等实体，动词是批准、调度、退款、升级等动作。FDE要找出哪些词被过载使用，哪些系统之间需要翻译，哪些系统是短期不可能替换的记录源。然后把这套语言编码进平台，让数据、权限、应用和操作围绕同一套定义运行。

语言一旦进入产品，就会反过来塑造组织。今天行业里大量使用“skill”（可复用的能力封装）、“MCP”（Model Context Protocol，一种让AI应用以统一方式连接外部工具和数据源的开放协议）、“agent”等词，其中一些原本只是内部实现或旧概念的新包装；但当产品界面、文档和生态都采用同一称呼，它就成为人们讨论问题的默认单位。Ganesh所谓“定义语言，就控制叙事”，不是营销话术，而是平台锁定的一种深层形式：客户开始用你的名词描述自己的业务，后续工具也会自然围绕这些名词构建。

这也是Foundry能够支撑后来大规模FDE交付的基础。早期现场工程师先从无数客户环境里提炼名词、动词和系统边界，再把它们固化为ontology能力；后来的FDE才可以更快完成数据集成和应用组装。没有前一轮产品化，就不会有后一轮进入市场效率。

每个临时补丁都可能活进生产

Ganesh最后讲了一个令人不安的事实：现场为了赶进度写下的临时补丁（hack），几乎总会进入生产。他曾见过一段临时Groovy脚本在一个十万人规模的客户环境中活了一年以上，甚至有了自己的昵称。Groovy是运行在Java生态上、常用于写脚本和自动化的编程语言；这里重要的不是语言选择，而是临时脚本变成了长期基础设施。

那段脚本原本只为处理一次数据保留问题。Ganesh随手用Groovy写成，没有按生产软件设计，也没有考虑多处复用。十二个月后，它已经运行在这家近十万人的企业多个地方，团队不得不长期支持。脚本的文件名甚至变成Ganesh在Palantir的绰号；在他的婚礼上，还有同事把名字印到衣服上。笑话之所以流传，是因为每个现场工程师都见过同一种命运：越是解决了真实痛点的临时东西，越不可能被轻易移除。

为什么删不掉？因为它真的解决了问题，后来又被别的流程依赖；项目继续向前，没人愿意承担改写风险。“临时”只是写代码那天的心理状态，不是系统的生命周期。

因此，他建议把快速交付按“可能存活18个月”来写：至少要有清晰所有者、日志、基本测试、权限边界和退出方案。快不是不要工程纪律，而是用更小的范围保住纪律。

在按下部署按钮前，他建议FDE问几件不舒服的问题：六个月后它坏掉，我会不会接到凌晨两点的电话？为了今天的速度，我给整个产品留下了什么负担？我离开客户之后，谁接手——客户、另一名FDE，还是核心产品团队？故障怎样被发现，谁有权限修复？如果答案都是“以后再说”，那就不是快速产品发现，而是在透支未来团队。

解决方案架构师、客户成功和FDE也因此不能只按“让客户满意”区分。客户成功当然重要，但Ganesh给FDE增加了一条更严格的产品责任：交付必须让核心产品获得杠杆。一个补丁若只让当前客户暂时高兴，却没有带来可验证的产品学习，甚至制造永久维护债务，就没有完成FDE作为产品延伸的工作。

快速原型与草率代码不是一回事

原型可以只覆盖一条业务路径，但必须让失败可见、让责任明确。真正危险的是范围很大、验证很少，却因为叫作“试点”而绕开生产思维。

把整堂课压缩成一张工作清单，大致是四步。第一，重新定义问题，不把客户写出的解决方案直接当需求。第二，真正进入现场，用小交付赢得访问权，并从重复动作里收集证据。第三，提炼客户的名词、动词和系统边界，把语言变成平台。第四，对每个临时方案承担生产后果，并决定它应该进入核心产品、留作客户特例，还是尽快删除。

这四步最终都指向“产品杠杆”。一次交付当然要让当前客户的工作变好，但更高的标准是：它是否使下一次交付更快，是否让平台能处理一种以前处理不了的问题，是否改变了公司对产品边界的理解。对于早期公司，FDE如果只被当作销售后的救火队，会耗尽最宝贵的现场信号；把它放进产品发现循环，才可能从几个项目长出真正的平台。

本课小结

客户通常带着一个已经想好的方案来，FDE要继续向下追问目标、动作和现有绕行。一天以内的行动闭环能快速换取事实与信任；现场观察能发现会议里说不出的阻力；而任何被用户依赖的临时代码，都应按长期生产资产对待。

Kepler展示的是从一个具体动作里发现产品机会。但企业AI最终要进入的，往往不是一个动作，而是一整套跨部门流程。下一课把镜头从调度员和数据工程师移到财务部门，看看正常路径、异常处理和系统记录怎样共同决定Agent能否工作。

想一想

1. 把你最近收到的一条功能需求改写成“用户看到结果后会做什么”。
2. 哪一种日常小动作最可能被你的产品设计忽略？
3. 你会为一天内交付的原型保留哪三项生产纪律？

第四课　真正难的不是执行，而是把公司的隐性流程画出来

这堂课要讲清

- 为什么流程文档只描述“顺利时怎样走”，AI却总在例外处失败；
- 如何把人类流程重画成自主、人在回路和纯人工三类步骤；
- 为什么企业更愿意让Agent覆盖旧系统，而不是推倒重来；
- Varick怎样用“FDE Agent”扩张上下文处理能力。

谁在讲

Vasuman Moza 是 Varick Agents 创始人兼CEO。演讲后半段由一位名为 JD 的工程负责人介绍内部FDE Agent的技术设计；字幕未提供可由公开资料可靠核验的完整姓名。Varick从部门流程审计入手，把Agent接入企业已有的工具、数据和运营逻辑。

本课对应演讲：AI tools for Forward Deployed Engineering。

先补背景：财务部门不是一条流程

企业财务至少包含应付账款（AP）、应收账款（AR）、公司卡与费用、银行、账单和财务规划分析（FP&A）等工作。发票、采购订单、付款和会计记录又分布在邮件、银行及Oracle旗下云ERP NetSuite等系统里。ERP是管理财务、采购、库存等企业资源的系统，常被企业当作最终权威的“记录系统”；AI可以在上层帮人判断和执行，却不能随意另建一套互相矛盾的账。

流程图通常只画正常路径。Varick关注的是谁在匹配失败时接手、哪一步要等四天、同一个人的名字为什么有两种拼法。把这些关系表示成依赖图以后，模型才知道某个动作需要哪些前置事实，也能在修改流程时检查是否制造循环或遗漏负责人。

文档里的流程，与公司真正运行的流程

Moza认为，执行正迅速变便宜，下一道瓶颈是理解业务。这个判断带有演讲者鲜明的乐观立场——他甚至说知识工作的执行“近乎被解决”——但他指出的现场难题非常具体。

过去两年，模型从回答问题走向执行任务。API是软件之间按约定请求数据或动作的接口；MCP（Model Context Protocol）则是让AI应用以统一方式连接外部工具和数据源的开放协议。浏览器操作、API工具、MCP和各种Agent运行框架，让AI可以登录系统、读取数据、填写表单、发送消息。于是企业很容易产生一种错觉：只要模型足够聪明，再给它工具，流程就会自动运转。Moza的反驳是，执行能力增长并没有自动带来业务理解。模型能点按钮，却不知道这家公司为什么在这里停一下、为什么另一家公司在同一步骤必须多找一个人批准。

这种差异往往不在行业层面就能解决。同样是销售部门，医疗公司的客户资格、合规要求和采购周期，与SaaS公司的线索分配和续费动作完全不同；同样是应付账款，两家企业对发票容差、供应商例外和审批层级也可能完全不同。通用模型掌握了“AP通常怎样运作”，不等于掌握了“这家公司的AP今天怎样运作”。后者需要从人员、记录和异常中重新建模。

以财务部门为例。FDE会分别访问应付账款（AP）、应收账款（AR）、卡片对账、银行、账单和财务规划分析（FP&A）负责人。正式流程也许写着：收到发票、匹配采购单、审批、付款。但真实情况更像这样：正常单据由Sarah处理；采购单与发票对不上时，她转给Chris；Chris要等待另一个部门回复，四天后才能继续。

这些流程也不是彼此独立的六条线。AP发生的付款会影响现金预测，AR回款会改变FP&A的判断，银行卡对账又可能暴露费用系统里的异常。每个流程负责人都会合理地认为自己的问题最紧急，却只有跨流程观察才能看见真正的依赖。FDE若只采访一个部门，就可能优化局部步骤，同时把等待或风险推给下游。

公司的文档通常记录黄金路径（*golden path*），真正决定自动化成败的却是边缘情况：信息不全、角色重叠、审批人休假、两套系统的同一人拼写不同、某个部门临时绕过正式规则。Agent若只学到理想流程，异常一来就会停住或自信地做错。

所以FDE的第一项工作不是写Agent，而是建立一张足够真实的流程地图：谁在什么时候需要什么信息，依赖什么前置条件，错误交给谁，多久之后才算失败。

地图至少要同时容纳四种事实。其一是正式规则，例如采购金额超过多少必须升级审批；其二是系统状态，例如发票是否已经入账；其三是组织关系，例如某位负责人休假时谁能替代；其四是经验判断，例如哪些供应商虽然字段不全却通常可以放行。前三类较容易进入数据库，第四类往往只存在于老员工的习惯里，却恰恰决定Agent能否处理真实世界。

不要把AI贴在一条坏流程上

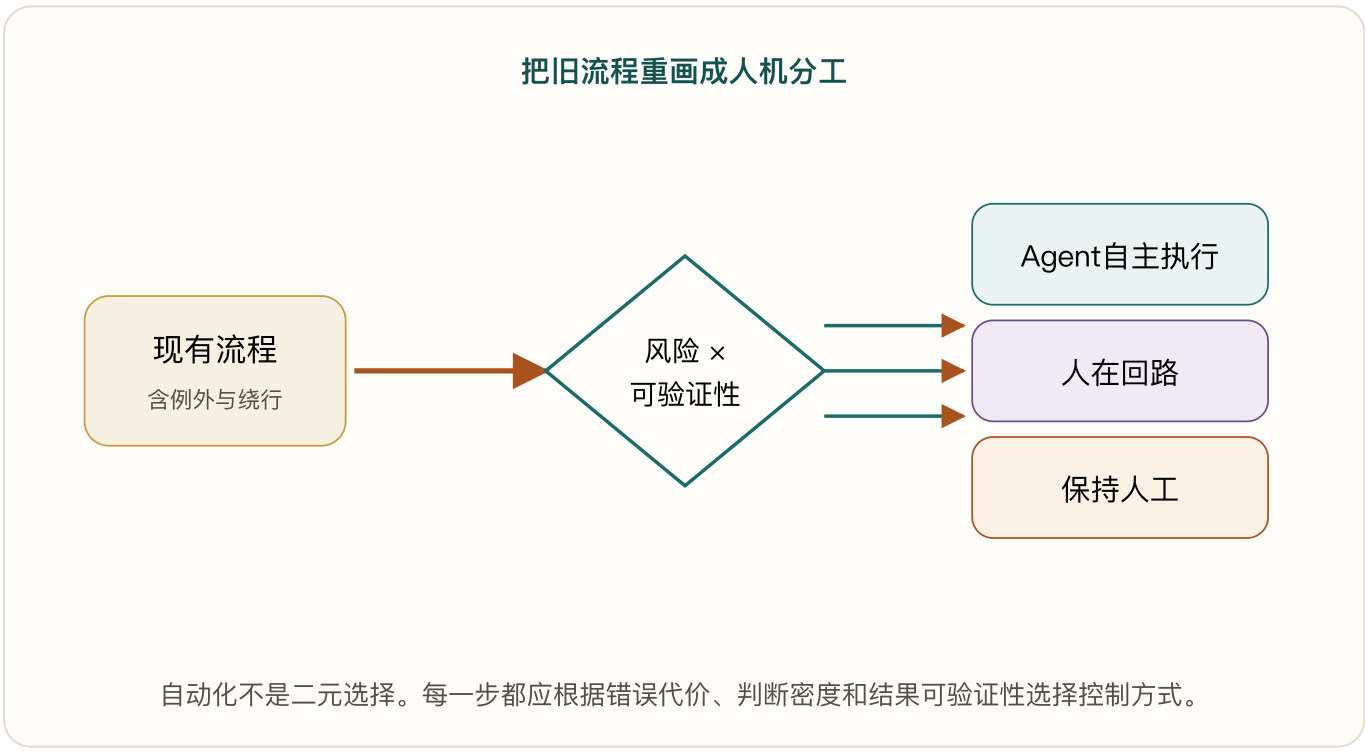
理解现状之后，第二步不是原样自动化，而是重新设计。Moza给出一种三分法：

1. **完全自主**：风险可控、规则清楚、结果容易验证的步骤交给Agent；
2. **人在回路**（*human in the loop*）：Agent准备材料或建议，人类批准关键动作；
3. **纯人工**：高风险、价值判断密集或极少发生的步骤继续由人处理。

假设原流程有八步，可能四步由Agent自动执行，三步需要人工确认，一步保持人工。目标不是把十一键点击粗暴压成一个按钮。业务人员需要理解系统怎样运作，组织也需要逐步建立信任。变化太小没有投资回报，变化太大则可能因采用失败而一无所获。

这里有一个容易被技术团队忽略的采用悖论。假如员工十年来一直按十一段完成工作，突然出现一个按钮声称可以全部代劳，即便它真的有效，人们也未必立刻信任。因为旧流程除了完成任务，还让员工看见控制点：他们知道什么时候能检查、什么时候能撤回、出了错该找谁。把步骤减少到一个按钮，同时也可能把可理解性一起删除。

因此，流程重构既不能只追求“与过去相似”，也不能只追求“自动化比例最高”。合理做法是先保留关键检查点，让Agent承担准备、匹配和重复执行，人类在高风险动作上确认；等评测和运行记录积累起来，再逐步扩大自主范围。组织信任不是培训会上宣布出来的，而是通过连续、可解释的正确执行建立的。



自动化不是二元选择。每一步都应根据错误代价、判断密度和结果可验证性选择控制方式。

五年、五百万美元的系统不会因为AI被扔掉

一家客户告诉Varick，他们花了五年和五百万美元迁移到NetSuite。现在若有AI供应商说“我们的方案很好，但你得先迁出NetSuite”，答案几乎一定是否定。

企业把ERP、CRM和财务系统称为记录系统（*systems of record*），因为关键数据、权限和审计责任都在里面。ERP管财务、采购、库存等企业资源，CRM管客户、销售与服务关系。Agent若完全绕开它们，只能成为另一个信息孤岛。Varick的选择是在 Salesforce（CRM平台）、NetSuite（云ERP）、Microsoft Dynamics（CRM与ERP应用套件）或 SAP 之上构建Agent，通过既有系统执行动作，而不是要求客户再次推倒重来。

“记录系统”这个词的重点不只是数据存放位置，而是制度上的权威。月底结账时，财务人员相信哪一套余额？审计追问一笔付款时，哪一套日志具有证据效力？员工离职后，权限从哪里收回？AI可以在邮件和文档里形成很好的建议，但若最终动作没有写回ERP，企业仍要由人手工同步，新的自动化层很快会制造另一套不一致记录。

在旧系统之上建设，也不是简单调用几个API。Agent要继承原有权限，区分读取、建议和执行；写入前要检查前置状态，写入后要留下审计记录；发生错误时要能暂停、回滚或交给人。对财务场景而言，一次重复付款或错误入账的代价，远高于模型回答得不够漂亮。FDE必须把模型行为放回企业控制体系，而不是让企业适应一套脱离现实的新控制体系。

这条现实约束也划清了演示与部署的差别。演示可以上传一份文档并输出答案；生产系统必须知道谁有权读写、动作怎样回滚、记录留在哪里、失败时由谁接管。

为FDE本人造一个Agent

Varick的FDE遇到了一个讽刺场面：平台工程团队用着各种编码Agent，隔壁负责客户的同事却被邮件、会议记录和上百页文档淹没。他们把材料上传给通用模型，等待很久，得到的分析又冗长又常抓错重点。

JD回忆，他所在的平台团队用Codex、Claude等工具写代码，工作节奏相对轻松；抬头看另一边，FDE却长期缺觉。客户全天发邮件，不同流程负责人把团队拉向不同方向，一次分析可能先要上传约150页材料，再等通用模型输出一段冗长且重点错误的回答。问题并不是FDE不会使用AI，而是通用助手没有一套适合持续客户参与的记忆结构。

这也暴露了FDE扩张最现实的瓶颈。理想人选既要是能够理解模型、系统和数据的顶尖工程师，又要有足够的沟通能力，在客户不完整甚至互相矛盾的叙述中提取事实。企业常见的两条培养路线——把咨询顾问训练得更技术，或把工程师训练得更善于沟通——都需要时间。Varick因此不指望无限招聘“高IQ加高EQ”的稀缺人才，而是尝试用Agent扩大每个人能管理的上下文。

JD据此设计了三阶段FDE Agent：

第一阶段：参与助手（engagement agent）。它读会议笔记、文档、幻灯片和消息，回答“这个流程由谁负责”“两种拼写是否是同一个人”等上下文问题。

这一阶段看似只是企业搜索，实际要求比一次性问答高。会议笔记里写“Sarah”，邮件签名可能写全名，Slack里又使用昵称；同一个缩写在财务和技术团队含义不同；流程负责人一个月后可能更换。参

与助手需要把来源、实体和时间对应起来，让FDE能追问“是谁说的”“当前负责人是谁”，而不是从一堆材料中生成一段无法追溯的概括。

第二阶段：工作流助手（workflow agent）。 它嵌入Varick自己的平台，在FDE搭流程时提醒遗漏的边缘情况、责任人和依赖关系。

嵌入平台很关键。若助手只在聊天窗口里给建议，FDE还要自己把答案翻译成流程节点；当它与构建界面并排工作，就能检查当前设计是否漏掉例外、发送对象是否正确、一个审批是否在前置步骤完成之前被触发。上下文从“可读材料”变成“可执行约束”。

第三阶段：自主维护助手。 未来，客户发来“把质检报告改发到另一个地址”之类的小请求时，Agent能查询公司知识、修改流程并提交变更，让FDE把时间留给访谈、判断和重构工作。

第三阶段在演讲时仍是目标，而不是已经全面完成的能力。它真正困难的地方不在修改一个邮箱地址，而在判断请求是否来自有权修改的人、变化会影响哪些下游节点、是否需要测试和批准。若这些条件满足，Agent才能把零碎维护从FDE手里拿走；若条件不满足，它应生成变更建议而不是自行上线。

公司的运行方式需要一种机器可读表示

这套系统的底层不是一堆散文式文档，而是一张依赖图。一个审批必须发生在另一个审批之后，一个人可能在邮件与Slack里有不同名字，一个流程可能包含回路。图数据库还是普通关系数据库并非重点；重点是建立统一、可查询的公司表示。

JD认为，大多数企业流程的主干其实相当线性：A完成后才能到B，B批准后C才有资格行动。复杂性来自分支、重试和循环——信息不全退回补充，对账失败交给另一人，再回到原节点继续。依赖图适合表达“哪些事实必须先成立”，也便于检查某次修改是否让流程出现无法结束的环。

技术选型反而是次要的。团队可以使用专门保存节点和关系的图数据库，也可以在通用开源关系数据库PostgreSQL里表达这些关系；重要的是每个节点和边都有稳定含义，并能追溯到原始证据。没有这一层，所谓“公司知识库”只是更多文档，模型仍要在每次请求中重新猜测流程结构。

JD把技术问题分成两半：第一，从已有上下文中写出准确、克制的流程分析；第二，从巨大知识图中抽取真正相关的上下文。Varick选择在开源模型上做后训练，并在强化学习环境里教模型使用实体消歧、发现冗余循环和检查有向无环图违规等专用工具。后训练是在基础模型完成预训练后，再用特定任务数据和反馈调整其行为；强化学习则让模型根据行动获得的奖励信号学习策略。字幕对具体模型版本的识别不够可靠，本书不锁定名称；方法比型号更重要。

第一半并不只是“模型会不会总结”。JD观察到，前沿模型写长分析时往往非常详细，却缺乏咨询顾问式的取舍：不知道哪些细节会改变客户决定，哪些可以暂时略过。Varick用自己的流程样本后训练模型，目标不是让文字更华丽，而是让输出在完整与清晰之间找到适合交付的密度。

第二半更接近检索与推理。即使已经有一张巨大的知识图，模型也未必能稳定走到正确节点。团队为此提供专用工具，例如判断两个拼写不同的人名是否指向同一实体、寻找重复步骤、发现本应无环的依赖关系中出现了循环。再在强化学习环境里训练模型选择和使用这些工具。先找到正确上下文，再写出有用分析；两步缺一不可。

核心概念：上下文不是文档堆

“把150页资料塞进提示词”并不等于理解公司。可用上下文需要实体一致、关系明确、时间和权限可追踪，还要能按当前任务抽取。FDE Agent真正试图规模化的，是这套组织记忆的整理与检索。

自动化FDE，还是放大FDE

Varick把项目称为“AI FDE”，但近期目标不是删除现场工程师。Agent先接管资料整理、重复核查和小型维护，让一名FDE同时维持更多客户上下文。最难的工作仍是：让业务人员说出没有写下来的例外，判断哪些流程值得改变，并在风险、回报与接受度之间做设计。

Varick自己的交付顺序也说明了这一点。一次参与先从审计开始，FDE和策略人员进入一个部门，理解它从内部怎样运行；随后才进入实施，在平台上构建Agent。技术平台不可缺少，但它出现在理解之后。如果顺序颠倒，团队很容易拿着现成Agent寻找使用场景，最终只在旧流程上增加一个新的聊天入口。

Moza主张按整个部门而不是单一点方案思考回报。只自动化获客或AP的一小段，收益可能被上下游等待抵消；跨流程重构才有机会同时改变收入、成本和风险。他在演讲中给出的25%、50%乃至75%回报属于公司项目口径，不能当作普遍保证，但它提醒我们，Agent价值常常取决于流程边界画在哪里。边界太窄，AI只能让一个局部动作更快；边界足够完整，才可能改变部门的运行方式。

换句话说，Agent最先自动化的是FDE的行政负担，而不是FDE的责任。

本课小结

企业流程的难处不在黄金路径，而在例外、依赖与隐性分工。FDE先画出现状，再按风险和可验证性重画人机边界；Agent接在既有记录系统之上。Varick进一步把组织知识表示、上下文抽取和小型维护交给FDE Agent，以扩张人的理解能力。

把流程画清楚之后，新的问题马上出现：可以自动化的东西仍然很多，团队先做哪一段？谁说的“紧急”才是真正的紧急？下一课进入Ramp的工程队列，讨论FDE最容易被忽视、也最能节省成本的一项能力——在写代码之前压缩范围。

想一想

1. 你所在团队哪条流程的正式文档与实际做法差距最大？
2. 其中哪一步适合完全自动、哪一步必须保留人工批准？
3. 要让一名新FDE接手老客户，需要把哪些隐性知识变成组织资产？

第五课　最稀缺的工程能力，是在写代码之前说清楚不做什么

这堂课要讲清

- Ramp为何把“永远在界定范围”列为FDE第一原则；
- 怎样区分客户紧急需求与销售内部的紧迫感；
- 如何用Agent缩短scoping（范围界定），却把品味与判断留给人；
- 为什么高质量规范和评测比生成更多代码更重要。

谁在讲

Leo Mehr 是 Ramp 工程总监（Director of Engineering）。他加入时FDE只有约两名工程师，演讲时已发展到约30人，覆盖部署、开发者API与AI服务。Ramp把FDE放在工程组织内，使命是帮助财务平台进入更大企业。

本课对应演讲：How Forward Deployed Engineering is done at Ramp。

先补背景：scoping（范围界定）为什么不是估工期

SAP是大型企业管理软件供应商，它的新一代企业资源计划（ERP）套件S/4HANA，常承载财务、采购、供应链等核心记录和流程。Ramp提供企业卡、费用和财务自动化产品；进入大型客户后，它必须与S/4HANA等核心系统协作。所谓“做一个SAP集成”，可能同时涉及身份认证、数据字段映射、同步方向、失败重试、安全审查和长期维护；它不是把两个接口接起来那么简单。

Scoping常被译作范围界定。Mehr所说的范围界定发生在估工期之前：先确认谁在使用、业务为什么紧迫、现有API能否解决、是否有临时替代方案，以及相同问题会不会在其他客户重现。只有问题成立，工程团队才有必要讨论做多久。

周五晚上的SAP请求

周五晚上，一名销售发来消息：一家重要客户需要 SAP S/4HANA 集成，能不能尽快做？

这个场景之所以典型，是因为每个参与者都在做自己的工作。销售想在季度结束前拿下一家战略客户；客户希望新财务平台能够进入已有ERP；工程师看见一个技术问题，第一反应是搜索SAP接口文档。没有谁明显做错，但如果团队沿着各自的局部目标直接行动，就可能在还不知道集成用途之前，先承担一项长期产品责任。

FDE最容易给出的答案是“能”。客户重要，交易临近，工程师又能直接动手。问题是，销售口中的“紧急”可能只是本季度额度的紧急，不一定是客户业务的紧急。团队若对每项请求都立刻承诺，很快会被几十个半成品拖垮。

Mehr特意反对一种流行形象：把FDE看成所有技术售前角色的“最终进化形态”，仿佛职责就是拥有更高权限、更快对客户说“可以”。Ramp的FDE属于工程组织，目标是让核心产品和新的Agent能力适用于大型企业。它当然服务交易，却不能只按单笔交易优化。每一次承诺都会进入代码库、发布流程和支持队列，影响下一批客户。

Mehr的第一原则是：**Always be scoping——永远在界定范围**。

这不是故意拖延，而是先把请求放回完整上下文：

- 客户为什么现在需要？不做会发生什么？
- 真正的用户有多少，何时会开始使用？
- 今天有没有手工替代办法？成本多高？
- 客户自己的技术团队可以承担什么？
- 其他在谈客户会不会遇到同一问题？
- 这个承诺会挤掉队列里的哪一项工作？

范围判断同时面向客户和公司。FDE既不能把销售目标伪装成用户价值，也不能因为追求完美架构而错过交易和学习机会。

对于SAP请求，一轮合格的scoping至少会继续追问：谁会触发同步，数据从Ramp流向SAP还是反过来，客户已经拥有哪种中间件，失败时是否允许人工补录，是否存在安全与审计截止日期，客户工程团队能否直接使用Ramp API。答案可能证明必须建设正式集成，也可能发现短期手工导出足以支撑试点，或者由客户现有集成平台完成更合理。

最重要的一问，是把当前客户放回整条需求管线：还有多少在谈客户使用S/4HANA？相邻客户使用的是同一种接口，还是完全不同的部署版本？如果只为一个季度末交易编写专用连接器，它会不会挤掉能服务十家客户的另一项能力？FDE的范围判断因此也是一种投资组合管理。它不是判断请求是否有价值，而是判断它相对于其他机会是否最值得现在做。

需求里缺少的那个限定词

Ramp曾为一家客户开发移动报销能力，团队自然地同时支持Apple的iOS与Google的Android这两种主流移动操作系统。后来才知道，客户的移动设备管理政策只允许iOS。Android部分从第一天起就没有必要。

当时Ramp自己的移动团队已经排满任务，FDE决定自己补位。两名工程师临时学习iOS和Android开发，连续工作数周，把两个版本都做了出来。团队带着完成大项目的兴奋向客户索取Android测试用户名单，才听说这家企业强制员工使用iOS设备。不是Android用户很少，而是根本不存在。

这个教训看似简单，却击中需求工作的常见偏差：工程师会自动补全缺失信息，而且通常按“完整产品”补全。客户说“移动端”，团队脑中出现“两大平台”；客户真正的约束却是“公司配发且受管的iPhone”。

scoping不是把需求写得更长，而是找出能删掉大量工作的事实。

浪费并非来自技术难度，也不是客户临时改口，而是团队没有验证最基础的使用环境。工程师越有能力，越容易快速填补空白；一旦“完整实现”被视为专业，就更难停下来问一个听上去过于简单的问题。scoping的纪律正是把未经验证的名词拆开：移动端是哪一种设备，集成是哪一组动作，实时究竟需要多少秒。

> **核心概念：范围压缩**
>
> 范围压缩不是降低客户价值，而是删除与目标无关的实现。最好的范围判断常让交付更小、价值更早出现。它问的不是“我们最多能做什么”，而是“哪一组最少工作足以改变结果”。

第二原则：用token扩张，而不是只用人头扩张

Ramp的第二原则是“scale with tokens”。如果FDE需求随大型客户增长，而每增加几个客户就必须增加同等人数，团队成本和沟通复杂度会线性上升。编码Agent提供了另一条路：把可重复的资料整理、规范生成、代码实现和测试交给Agent。

Mehr加入Ramp时，FDE只有两名工程师；两年半后，他负责的相关组织约有30人，覆盖FDE、开发者API和新的AI服务。团队扩大说明企业需求真实存在，也让人头扩张的上限变得清楚：更多工程师会带来更多客户上下文、更多交接和更复杂的优先级协调。若工作方法不变，人数并不会按比例转化为吞吐。

“用token扩张”也不是把工资预算机械换成模型账单。它要求不断重画人的工作：今天由FDE手工完成的资料汇总、补充提问和初稿规范，六个月后是否可以成为Agent步骤？若可以，需要什么工具、记

忆和评测？团队的扩张单位从“再招一个同样的人”，逐渐变成“让现有工程师监督更多高质量工作循环”。

Mehr把FDE生命周期拆成四段：上下文、范围、规范、实施。两端相对容易自动化：Agent可以收集团队沟通工具Slack、文档与知识工具Notion和历史项目中的材料，也能依据明确规范完成中等规模功能。中间最难，因为它需要对业务优先级、产品方向与工程代价做判断。

这四段看起来像一条顺序流水线，实际会不断回流。实现时发现API缺少能力，可能迫使团队重写范围；评测暴露成功标准不完整，又要回到客户上下文；相似需求在第三个客户出现，规范也许应该从客户项目升级为核心产品。Agent系统不能只把文档从一个格式转换成另一个格式，而要允许证据改变上游决定。

Ramp因此搭了一个内部请求Agent。员工在Slack或Notion提交需求后，Agent会继续追问，直到信息达到可写规范的程度。Mehr称，原先需要数小时甚至数天的往返可以缩短到秒级，并估算节省约20%的scoping时间。这是其演讲中的内部估算，意义在于工作形态：Agent不替人决定优先级，而是让决定所需的信息更快齐全。

原来的入口是一个名为“FDE requests”的Slack频道。客户成功、销售和客户经理在这里提交阻碍交易或采用的问题，详细程度差异极大：有人附上完整Notion页面，有人只写一句“需要SAP集成”。FDE必须阅读材料、查询产品现状，再来回追问用户、时间、替代方案和成功标准。大量时间花在把一句请求变成可以判断的请求，而不是解决问题本身。

第一版Agent非常简单：读取请求，只问几条缺失问题。即使如此，它也把第一次响应从小时或天缩短到秒，提交者会在记忆仍然新鲜时立即补充。后来的版本会进行多轮对话，直到判断信息足以生成初步规范。Ramp甚至给它配了一个企鹅形象，让内部员工更愿意与它互动。这个细节说明，内部自动化也有产品采用问题；一套严肃而冷冰冰的表单未必比一个会追问的入口收集到更多事实。

20%的时间节省只是第一阶段。更重要的是，Agent把scoping从分散在人脑里的习惯，变成一套可以观察和改进的问题序列。团队可以检查哪些字段最常缺失，哪些回答仍然需要人工追问，哪些请求最终被否决，再把这些反馈写回系统。

Agent流水线需要评测，而不是信心

如果从上下文到实施都由多个Agent衔接，每一段都会放大前一段的偏差。模糊的背景生成错误范围，错误范围生成精致规范，最后得到结构漂亮但无人需要的代码。

因此，Ramp强调评测（*evals*）、评分规则（*rubrics*）与人工反馈。系统要知道一份好规范包含什么、哪些需求曾被否决、什么情况下必须升级给人；Agent还要拥有稳定上下文、技能、记忆和工具。

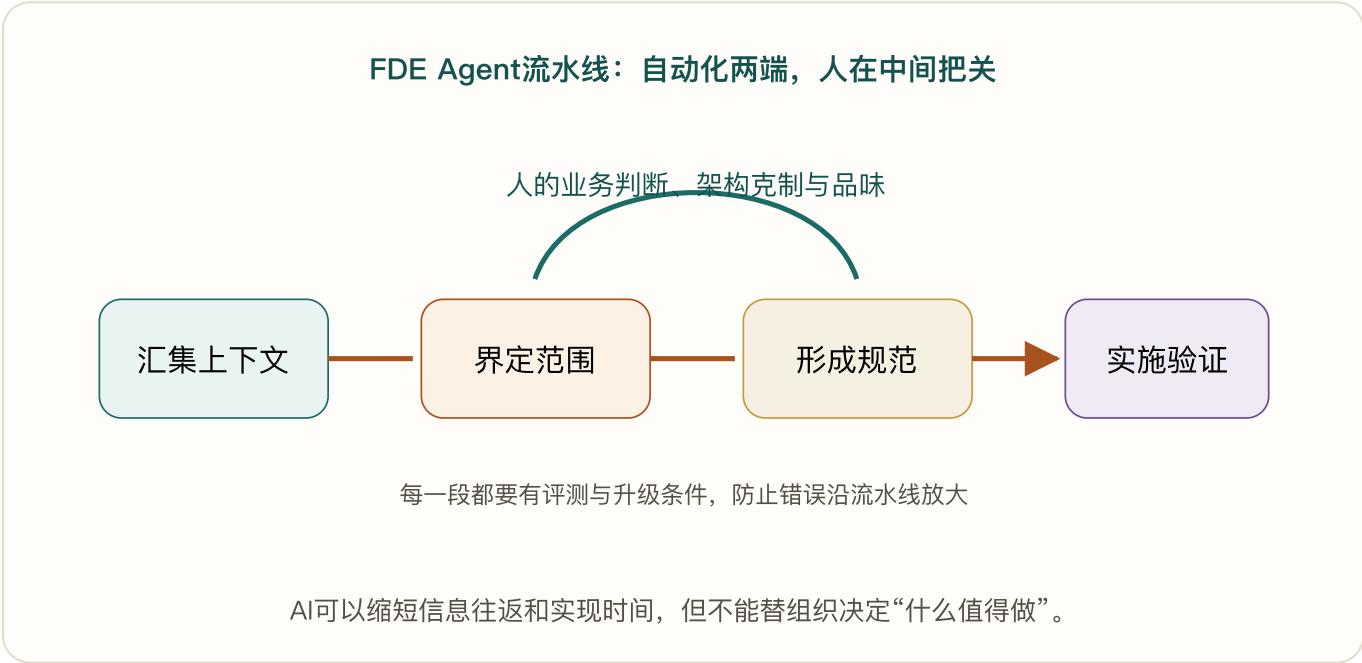
所谓rubric，可以是一套明确检查项：规范是否写出真实用户、当前绕行、成功指标和非目标；是否验证了产品已有能力；是否列出权限、安全和维护影响；是否说明为什么现在做。它不保证Agent拥

有正确判断，却能让明显缺口在进入实现前暴露。评测样本还应包含历史上的好决定和坏决定，而不只是最终被批准的项目，否则Agent只会学习怎样把请求写得更像应该通过。

上下文是更难的部分。Notion、帮助中心和产品文档能告诉Agent“产品声称怎样工作”，却未必包含产品经理脑中的历史：某个接口为什么故意不开放，过去哪种方案造成过事故，哪位客户的例外不应一般化。要让Agent接近这种判断，团队必须把决策记录、代码状态、客户反馈和失败案例逐步连接起来，而不是只扩充知识库文章。

人在系统中的位置也随之变化。过去工程师亲自完成每一步；现在人更像评审者和编辑，集中使用“品味”（*taste*）：目标是否值得，范围是否过大，方案是否符合产品方向，代码是否在制造未来负担。

这里的“品味”不是无法解释的天赋。它来自看过足够多的客户请求、维护过自己曾经承诺的代码、理解公司产品方向，也知道什么样的局部胜利会制造全局失败。好的Agent会让这些判断出现得更频繁，因为它降低了收集和实现成本；它并不会让判断本身消失。



两种失败，必须同时避免

Mehr用两个很有画面感的极端收束全场。

如果没有扎实scoping，Agent工厂会变成“拼命烧token的垃圾炮”：它高速生成不需要的功能、重复集成和维护债务。如果只重视判断、完全不利用Agent扩张，竞争者又可能用更快的工程循环赢得客户。

真正的路线是同时提高两种能力：更严格地决定做什么，更自动化地完成已经决定的工作。前者保护方向，后者扩大吞吐。

本课小结

FDE的稀缺能力不是对所有请求快速说“能”，而是辨认客户价值、组织紧迫感和工程代价。Agent可以收集上下文、追问缺失信息、生成规范并实施；人必须保留范围判断、评测设计与架构品味。速度与克制缺一不可。

范围压缩让第一次交付变得可控，却没有回答同一类需求第二次、第二十次出现时怎么办。下一课观察一家从50人长到500人的Agent公司，看看现场定制如何经过组织分工和产品漏斗，变成客户能够自助使用的能力。

想一想

1. 最近一个“紧急”需求，究竟是谁的紧急？
2. 哪个限定事实一旦问清，可以删除一半实现工作？
3. 你的Agent流水线在哪一步最容易把小偏差放大成错误产品？

第六课　AI越会写代码，越要克制“一次性做掉”的冲动

这堂课要讲清

- Decagon为何把FDE拆成“配置Agent”与“改进产品”两条线；
- 公司从约50人增长到约500人后，交付组织怎样专业化；
- 为什么AI编码让“克制与远见”成为稀缺能力；
- 如何建立从custom到self-serve的漏斗。

谁在讲

Sunny Rekhi 是 Decagon 的 CTO of Forward Deployed Engineering。Decagon提供面向企业的AI客户服务Agent，覆盖语音、邮件等渠道。Rekhi的现场经验跨越了公司从约50人到约500人的高速扩张阶段。

本课对应演讲：How Forward Deployed Engineering is done at Decagon。

先补背景：客服Agent到底由什么组成

企业客服Agent不只是一个聊天窗口。它要识别用户意图，查询知识库，调用认证、订单、会员和退款系统，遵循品牌与业务政策，并在信息不足或风险过高时把对话交给人工坐席。回答得自然只是其中一项能力，后台动作是否正确、交接是否完整，才决定它能否进入生产。

Decagon把现场工作逐渐拆成两条线：Agent Builder负责在产品内配置Agent的语气、意图、动作和人工交接；Agent Software Engineer处理产品仍然缺少的共同能力。两条线之间的目标，是把反复需要工程师完成的定制逐步变成客户可以自行配置的功能，也就是custom-to-self-serve。

同一个产品，进每家公司都要重新学习

客户服务Agent不是换一个公司Logo就能上线。它要学习品牌语气，知道哪些用户意图可以自动处理，哪些必须转给人；要连接认证、订单、会员、账单等后台系统，才能真正替用户修改信息或发起退款。

传统客服的用户体验很容易辨认：打电话后在语音菜单里“按1查账单、按2改会员”，发邮件后等待两三个工作日。Decagon要替换的不是一个问答框，而是这整段交互。用户通过电话、邮件、短信或其他渠道接入后，应当直接遇到一个能够理解自然语言、保持品牌语气，并且有权限完成动作的Agent。

“完成动作”让部署难度骤然上升。回答“退货政策是多少天”只需要找到正确知识；真正办理退货还要确认身份、读取订单状态、检查商品和时间是否符合政策、调用后台接口、把结果写回客服系统，必要时完整移交给人工。每一家企业的政策、系统和风险边界不同，所以底层Agent平台可以相同，进入生产前的学习却不能省略。

Decagon的forward deployment因此有两类工作。

第一类是**让Agent的大脑适应这家企业**：写清政策、工具、转人工规则、渠道与成功指标。尽可能通过产品界面和自然语言配置完成。

这里的“大脑配置”至少包含四层：Agent知道什么，即知识与政策；Agent能做什么，即可调用的工具与权限；Agent怎样表达，即文字语气、语音风格和品牌规范；Agent何时停下，即转人工和风险升级规则。训练一名人类客服时，这些内容会散落在手册、跟岗和主管反馈里；训练Agent时，它们必须变成可测试、可版本化的系统配置。

第二类是**把现场需求变成产品能力**：客户A提出一个功能时，FDE要预判客户B、C、D是否很快也会需要。不是为A写一段孤立补丁，而是让后来的客户还没提出请求就已经得到能力。

Rekhi甚至说，在Decagon，FDE与产品工程使用同一工程门槛、同一汇报体系，常常就是同一支团队。面对一家大型企业的痛点，现场请求与产品路线图之间没有清晰墙壁。

这和客户结构有关。中型品牌的需求可能通过现有配置解决；《财富》美国500强排名最前的20家公司，通常被简称为Fortune 20，一家这种规模的企业提出的问题却常常暴露平台层缺口。这里强调的是客户规模和复杂度，而不是一种技术分类。这类客户常带来新的安全模型、跨区域治理、特殊渠道或大规模运行能力。现场工程师若没有修改核心产品的能力，只能在外围不断增加补丁；产品工程师若不接触这些现场事实，也很难提前理解企业级要求。

从一个人全包，到两条专业化通道

公司规模较小时，Agent Software Engineer可以包办一切：与客户并肩配置Agent、连接后台、处理产品缺口。客户不多，这种英雄式模式能够运行。

当公司只有约50人时，同一个工程师了解客户、调提示、接接口、补产品功能，信息几乎不需要跨团队传递。速度很快，问题也能靠少数熟练者的记忆解决。但一年内增长到约500人之后，原来的优势会反过来成为风险：客户数量、行业跨度和内部协作同时扩大，没有人还能独自掌握全部上下文。

高速增长之后，瓶颈出现了。Decagon把工作分成两条通道：

- **Agent Builder**：熟悉不同模型的行为和Decagon平台，主要在界面内配置、测试和优化Agent；
- **Agent Software Engineer**：解决需要代码和平台变化的问题，把企业请求上游化为产品。

专业化不是重新筑墙。Agent Builder遇到界面无法处理的问题，要把它送入产品；Agent Software Engineer每次写代码，都要问怎样让Builder、客户或Agent自己完成下一次相似工作。

Agent Builder并不是“技术要求较低的实施人员”。他们要理解不同模型的行为，知道怎样在自然语言指令、知识、工具和评测之间定位问题，尽量在产品界面内完成配置；一旦发现必须写代码，真正重要的输出不是一张临时工单，而是清楚说明平台为什么表达不了这个需求。Agent Software Engineer则从这些缺口里寻找通用设计，让前线下一次不必再离开界面。

两条通道的协作质量，可以用一个简单问题检验：同类需求第二次出现时，交付方式有没有改变？如果第二个客户仍然需要同样的核心工程师、同样的私有脚本和同样的手工排障，组织虽然分了工，却没有形成扩张能力。

代码便宜之后，最贵的是克制

一个重要客户提出需求，工程师现在很容易让编码Agent迅速写出定制实现。这种诱惑比过去更强：既然一小时能做，为什么不做？

过去，一次性功能至少会受到开发成本约束。需要排期、写代码和测试，团队会被迫讨论它是否值得。编码Agent把第一版成本压得很低，也拿掉了这层天然阻力。工程师可能在等待产品评审之前就把补丁做完，而“代码已经有了”又会成为上线理由。生成速度因而会系统性地推动过度定制。

Rekhi的回答是，因为一次性方案的真实成本不在第一小时，而在未来。提示词、特殊分支和补丁会叠成黑箱；客户无法理解和拥有自己的Agent；下一个客户仍要重做；每次模型或平台更新都可能破坏隐藏依赖。

因此，AI编码时代的稀缺技能不是生成，而是“克制与远见”：这项需求会怎样扩展？应该成为配置、接口还是核心能力？一年后客户还能看懂吗？

Decagon强调客户应当拥有自己的Agent。这里的“拥有”不仅是合同上的控制权，也意味着客户能看懂行为从哪里来，能在政策变化时自行修改，并能追踪某次决策依据。若Agent由大量隐藏提示、客户专属分支和没人敢动的补丁构成，供应商看似迅速满足了请求，实际上把客户锁进一个脆弱黑箱。克制就是拒绝这种短期便利。

Decagon官方在会前发布的文章也给出相同方向：forward deployment若要扩张，不能只依靠嵌入现场、什么都能做的英雄人物，而要把交付本身当作系统设计，让一次手工工作变成下一位客户不必重复的能力。

核心概念：Custom → Self-serve

一次定制不是失败，它常常是产品发现的入口。失败是同一种定制反复出现，却始终没有变成配置项、通用接口、平台能力或清晰方法。可扩张的FDE会追踪每个手工步骤，决定何时把它移入自助漏斗。

先把“成功”写下来

面对大型企业，团队很容易急着开工。Decagon的教训是，最早几次对话就要把成功标准写清：客户要支持哪些渠道？想改善哪种意图？转人工率、解决率、响应时间、客户满意度或收入目标分别是什么？哪些情况不在第一阶段？

渠道本身也会改变问题。同一个“修改预订”，在邮件里可以等待几分钟并请求补充信息，在实时语音里则要控制沉默、确认用户身份并处理口音；消息应用WhatsApp可能要求短消息和异步恢复。若合同只写“上线客服Agent”，供应商可能完成了邮件，客户却在等待电话；双方都能拿出证据证明自己理解正确。把渠道、意图和指标写在最初的成功定义里，是避免这种错位的最低成本做法。

书面标准能防止双方在几个月后发现，供应商以为“Agent已上线”，客户却以为“所有客服都应被替代”。它也给评测提供了靶子：没有成功定义，就无法判断Agent是在创造价值，还是仅仅处理更多对话。

行业知识会复利

当客户跨金融、旅行、科技等行业，通用FDE每次都要重新学习监管、术语和常见流程。Decagon开始把有行业经验的人配置给同类客户。知识不仅帮助建立可信度，也缩短需求发现：金融客户D出现时，团队已经知道A、B、C反复遇到的边缘情况。

垂直化的价值不只是“会说行业黑话”。熟悉金融服务的人知道哪些动作通常需要强认证，哪些解释必须保留审计证据；熟悉旅行的人知道行程变更会牵动库存、付款和合作伙伴。团队因而能更早提出客户尚未想到的问题，也能区分行业共性与单家公司特例。

当然，横向经验也可能变成偏见。FDE不能因为前三家金融客户使用同一种流程，就假定第四家必然相同。成熟的行业知识更像一组高质量假设和检查表：帮助团队加速提问，而不是替代现场验证。

这使“每次部署都比上次快”成为可测目标。速度不只来自更熟练的人，还来自三种积累：垂直知识、产品自助能力和跨部署共享的评测数据。

先证明价值，再扩大边界

大型客户常在第一天就带来“整个厨房”：多渠道、复杂意图、所有集成、全球语言和严格治理。Decagon选择先找到能迅速证明价值的一小段工作，上线后再扩展到整个支持流程和收入场景。

例如，一家租车公司最初可能只想自动处理复杂的入站支持；Agent已经连接后台并与用户建立关系后，又可以主动提醒续租或延期，把支持入口扩展成收入触点。演讲以租车公司Hertz说明“land and expand（先落地、再扩张）”：先赢得一个可验证结果，再利用已经建立的数据、信任和连接扩大边界。

Rekhi对Hertz案例的描述正体现了这种顺序。Decagon先进入复杂的入站支持流程，已经拥有与租赁后台系统的连接，也理解客户状态。团队随后发现，同样的基础设施可以用于主动沟通：在租期即将结束时联系客户，询问是否续租或延长。客服Agent由成本中心里的自动化工具，扩展成可能创造收入的客户触点。

但扩张不能成为第一阶段无边界的借口。《财富》按营业收入排列的美国500家大型公司，即Fortune 500，往往会一次提出所有渠道、语言、意图和集成，Rekhi称之为把“整个厨房”都扔过来。FDE要从中找出最短的价值证明，在数周而不是数月内让一组真实用户得到结果。建立信任之后，再沿着已验证的路径扩大覆盖。

顾问，而不只是执行者

客户说“先自动化X”，FDE不必总照做。Decagon会分析历史支持数据，判断哪类请求量大、自动化可行且回报高，再建议先做另一项。因为FDE同时看过多家企业，它拥有单个客户看不到的横向经验。

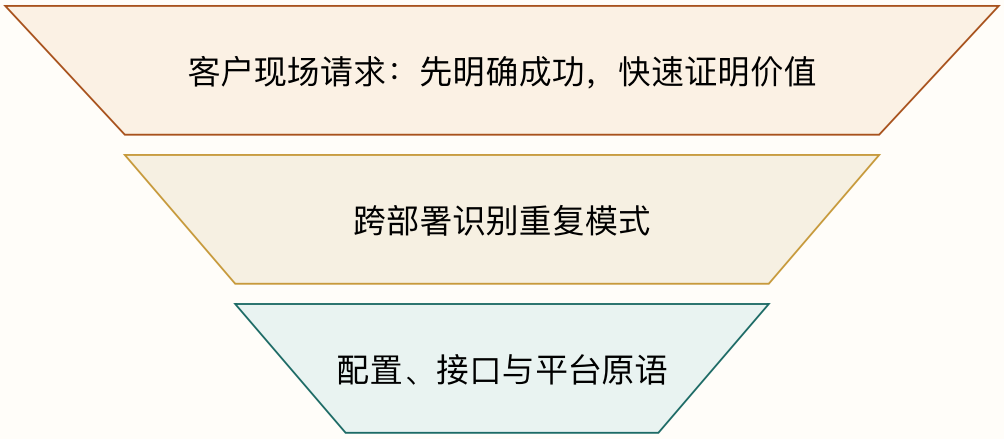
历史支持数据可以把争论变成排序问题。某类请求数量很大，却需要高风险判断，不一定适合第一阶段；另一类数量稍小，但流程稳定、后台动作清楚，可能更快形成自动解决。FDE要把业务影响、技术可行性、风险和上线速度放在一起，而不是只选择呼声最高的团队。

这并不授权供应商替客户决定一切。它要求FDE同时做执行者和顾问：尊重业务目标，又能用数据挑战方案；快速交付，又把每次手工动作送入产品漏斗。

Decagon早期曾为不同客户反复手写客户关系管理系统（CRM）集成。CRM保存客户身份、联系、销售与服务记录；客服Agent若要完整交接或写回处理结果，往往必须连接它。大约做到第二十五个时，团队意识到继续复制已经没有意义，于是把连接能力改造成自助方式。原本必须由工程师写代码的工作，后来可由Agent Builder甚至客户完成。这个过程说明custom-to-self-serve不是一句原则，而是一笔可以追踪的账：某个手工动作出现了多少次，消耗多少工程时间，抽象之后又让多少部署变快。

自然语言配置是Decagon给这条漏斗设定的方向。如果一项Agent行为仍必须由工程师进入代码修改，团队就应问它能否上游化到产品，让业务人员以自然语言表达规则，同时由平台提供版本、测试和护栏。并非所有复杂性都能消失，但工程师介入应当成为发现产品缺口的信号，而不是永久交付方式。

从现场定制到产品自助的漏斗



漏斗的目标不是消灭定制，而是让重复定制不断上游化。

本课小结

Decagon把forward deployment看成两条互相连接的工作：为具体客户配置Agent，把共性需求送回产品。规模扩大后，团队通过角色专业化、书面成功标准、行业知识和custom→self-serve漏斗，替代单纯依赖英雄人物的交付方式。AI让定制更容易，也让架构克制更重要。

产品化不能只凭“很多客户都提过”来判断。团队还需要知道现场问题是否真的影响结果，修复之后是否带来可测改善。下一课把客户部署当作一组高保真评测，区分Agent跑了多少与企业实际完成了什么。

想一想

1. 你们最近重复了三次的手工步骤，为什么还没有进入自助漏斗？
2. 第一阶段应怎样定义一个既快又有业务意义的成功指标？
3. 哪些行业知识应沉淀成团队资产，哪些必须留给客户判断？

第七课　别再用token证明价值：把客户现场变成最高保真的评测集

这堂课要讲清

- Cognition怎样理解产品能力与客户问题的重叠；
- 为什么写代码只是软件交付链的一部分；
- 现场部署如何同时解决客户问题并降低产品路线图风险；
- 为什么“组织变快”比“一个工程师变快”更难也更有价值。

谁在讲

Jia Wu 是 Cognition 部署工程负责人（Deployed Engineering Lead），并在AI编程环境Windsurf并入Cognition后加入。Cognition的Devin是一种软件工程Agent：它可在隔离的沙盒环境中，通过shell（向操作系统输入命令的终端）、代码编辑器和浏览器读代码、改文件并运行验证，既可独立完成任务，也可与工程师协作。

本课对应演讲：How Forward Deployed Engineering is done at Cognition。

先补背景：会写代码与交付软件之间差了什么

Devin与普通代码补全工具的差别，是它能在隔离环境中读取代码库、使用终端和浏览器、修改文件并运行验证。公开基准通常把任务整理成相对清楚的输入和结果，企业项目却包含遗留系统、内部审批、发布窗口、权限和维护责任。模型写出语法正确的改动，仍不等于组织敢把它合并并上线。

这一课中的eval是“评测”：用任务、样本和成功条件反复检查系统是否改善。一次Agent任务运行可以称为一个session；session数量和token消耗只能说明模型运行了多少，拉取请求是否被接受、迁移周期是否缩短、团队是否真的更快，才是客户能够感知的结果。

从13%的演示到企业生产

Wu先拿Devin在2024年刚发布时的印象开玩笑。最初的基准成绩和“首位AI软件工程师”叙事引发巨大关注，随后很多工程师发现，演示能力与自己的复杂代码库之间仍有距离。

SWE-bench是一套用真实GitHub问题及其修复来评估模型的软件工程基准：系统要生成代码补丁，再由项目测试判断问题是否真正解决。Wu提到的13%，指向Devin早期在这类任务上的表现。发布瞬间，工程师一面惊叹Agent能够自己使用终端、浏览器和编辑器，一面担心职业被替代；一周后，很多人又发现它在真实工作中经常需要救场。Wu用Cognition后来在旧金山投放的自嘲广告概括这段变化：“我们现在真的好用了。”笑话背后的问题是，一项基准能力怎样穿过两年产品迭代，变成企业愿意依赖的生产能力。

到2026年，Cognition提供云端Agent、CLI（命令行界面）和IDE（集成开发环境，把编辑、运行和调试放在同一工具中）等多种入口。Wu真正想解释的不是功能列表，而是这些能力怎样进入企业。她画了两个圆：一个是产品能做的事，一个是客户真正的问题。产品市场契合，不是圆里各有多少内容，而是重叠面积有多大。

这些产品界面对应不同的人机分工。命令行适合工程师在熟悉的终端中调用Agent；IDE让人可以在代码旁边持续协作；云端Agent则拥有独立计算环境，可以在后台运行较长任务。Wu本人在Windsurf并入Cognition后加入，因此也亲历了从IDE内协作到云端委派的产品组合。真正的企业部署往往不是三选一，而是根据任务形状安排入口。

Cognition在内部也用Devin改变自己的开发方式。Wu展示的公司口径称，过去六个月里，在招聘相对滞后的情况下，团队仍然合并了接近一个数量级更多的高质量pull request（PR，拉取请求）——也就是开发者提交一组代码变更，请团队审查后合并进主代码库。这个数字不能直接推导到客户，但说明“部署”并不是售后包装：如果厂商自己都没有把Agent接入日常工程流程，就很难理解客户会在哪些环节卡住。

FDE的工作就是从两边扩大重叠：把Devin部署到最有价值的工程问题，也把现场反复出现的障碍送回产品。

只做第一件事，会让FDE变成善于使用现有产品的实施团队；只做第二件事，则容易收集大量意见却没有当期客户结果。Cognition把两边视为同一任务：一次现场工作既要证明某个工程问题可以被Agent解决，也要告诉产品团队下一项能力为什么值得建设。

写代码大约只是问题的20%

在一个企业软件生命周期里，需求进入积压清单，工程师理解遗留系统，开发、测试、审查、发布、监控，再长期维护。生成一段代码只是其中一环。

现实企业很少从空白仓库开始。团队面对的是多年积累的服务、没人愿意碰的迁移、版本落后的语言、发布冻结窗口，以及只有少数老员工理解的依赖。一个用户故事进入backlog（待办开发任务清

单）以后，要先判断影响范围，再找到代码位置，补测试、通过审查和CI（持续集成，代码提交后自动构建与测试），协调发布，最后观察生产行为。Agent在中间写出代码，并没有自动完成这条链。

Wu在演讲中把编码估为整个问题的约20%。这个比例不是通用研究结论，但方向很关键：模型即使能写出正确代码，也未必知道哪项战略计划最重要、怎样在旧系统中验证、谁批准发布、出了问题怎样回滚、几年后谁来维护。

所以Cognition的FDE不是把Agent随手扔进代码库。一天可能有四五小时客户会议，再有四五小时亲手配置和构建。会议不是“非技术工作”，而是用于识别最高杠杆的业务计划、理解代码交付链，并决定哪里适合自动化。

在客户生态里，FDE会检查积压很久的功能、尚未完成的安全修复、缺失测试、告警分诊和迟迟推进不了的迁移。不同任务需要的Agent配置不同：有的适合在明确工单到来时自动启动，有的需要先由人拆分，有的只能生成候选改动并等待专家审查。所谓“把产品能力映射到问题”，不是给所有团队开通账号，而是为每一种任务设计触发、上下文、验证和交接。

会议与键盘工作各占半天，也说明FDE不能只做关系管理。工程师听完战略目标后，要亲自进入代码库验证假设：依赖是否真如客户所说，Agent能否运行测试，权限缺口在哪里。没有动手验证，客户访谈容易停留在愿望；没有客户理解，技术实验又容易落在低价值任务。

自动化自己，才算完成部署

找到场景之后，FDE会把Agent接入事件与告警，使它自动响应，而不依赖工程师每天手工触发。目标是“把自己从任务里自动化掉”：建立稳定上下文、触发条件、审查与交付路径。

例如，某类告警出现后，Agent可以自动读取相关服务、查找近期改动、提出修复并运行验证；积压中的重复迁移任务可以按模板持续执行。FDE最初可能手工示范每一步，但如果三个月后仍必须由同一个人登录、复制提示和点击启动，部署没有形成系统。真正完成的标志，是客户自己的事件能够稳定驱动Agent，结果进入原有工程流程。

这一步也迫使团队回答投资回报。运行了多少session、消耗多少token，只说明机器很忙；客户关心的是项目是否更早完成、积压是否减少、有效PR是否被接受、关键系统是否更快维护。

Wu给出三组匿名或公开案例口径：在一个三个月部署中，Agent提供了相当于约150人规模的额外工程容量；某些交付周期缩短约82%；相较单点工具，合并的PR数量接近翻倍。她还提到巴西数字银行Nubank的ETL迁移、拉美银行的遗留税务系统，以及美国住房奖励与支付平台Bilt的工程输出。ETL即“抽取、转换、加载”，是把数据从源系统取出、清理转换，再送入目标数据平台的工程流程。这些数字来自Cognition在演讲中的案例展示，不应被当成所有客户都能复制的行业基准。

她特意把指标一层层拆开。第一层是session和由此估算的“有效工程小时”，它回答Agent运行了多少有生产意义的工作，但仍可能被质疑只是另一种使用量。第二层是项目交付周期，把部署前后的ticket

（工作项）、sprint（短周期迭代）和完成时间进行比较，观察工作是否真的提前结束。第三层是被合并的PR，因为只有进入主干的改动才更接近组织接受的产出。

三层指标的关系很重要。大量session却没有合并PR，说明Agent也许在反复尝试；PR增加而项目周期不变，瓶颈可能已经转移到审查、发布或组织协调；周期缩短却故障增加，又不能称为价值。FDE要把Agent活动量连接到交付结果和质量，而不是挑一个最好看的数字。

公开案例让这种差别更具体。Wu提到Nubank一项ETL迁移原本投入约50名工程师，Devin参与后按公司口径在约三分之一时间内完成；另一家拉美大型银行的税务识别系统迁移，包含COBOL、JCL等遗留技术。COBOL是长期用于银行、保险和政府批处理系统的编程语言，JCL是IBM大型机上用来描述批处理作业如何运行的控制语言。该项目所需人力约减少一半；Bilt案例则强调合并PR和每周工程输出。案例的共同点不是某个统一提升倍数，而是Agent被放进了边界清楚、可以用完成时间和接受结果衡量的工程任务。

真正的评测集在客户那里

如果FDE只完成客户目标，工作只做了一半。现场遇到的问题需要被整理成产品团队能使用的证据：

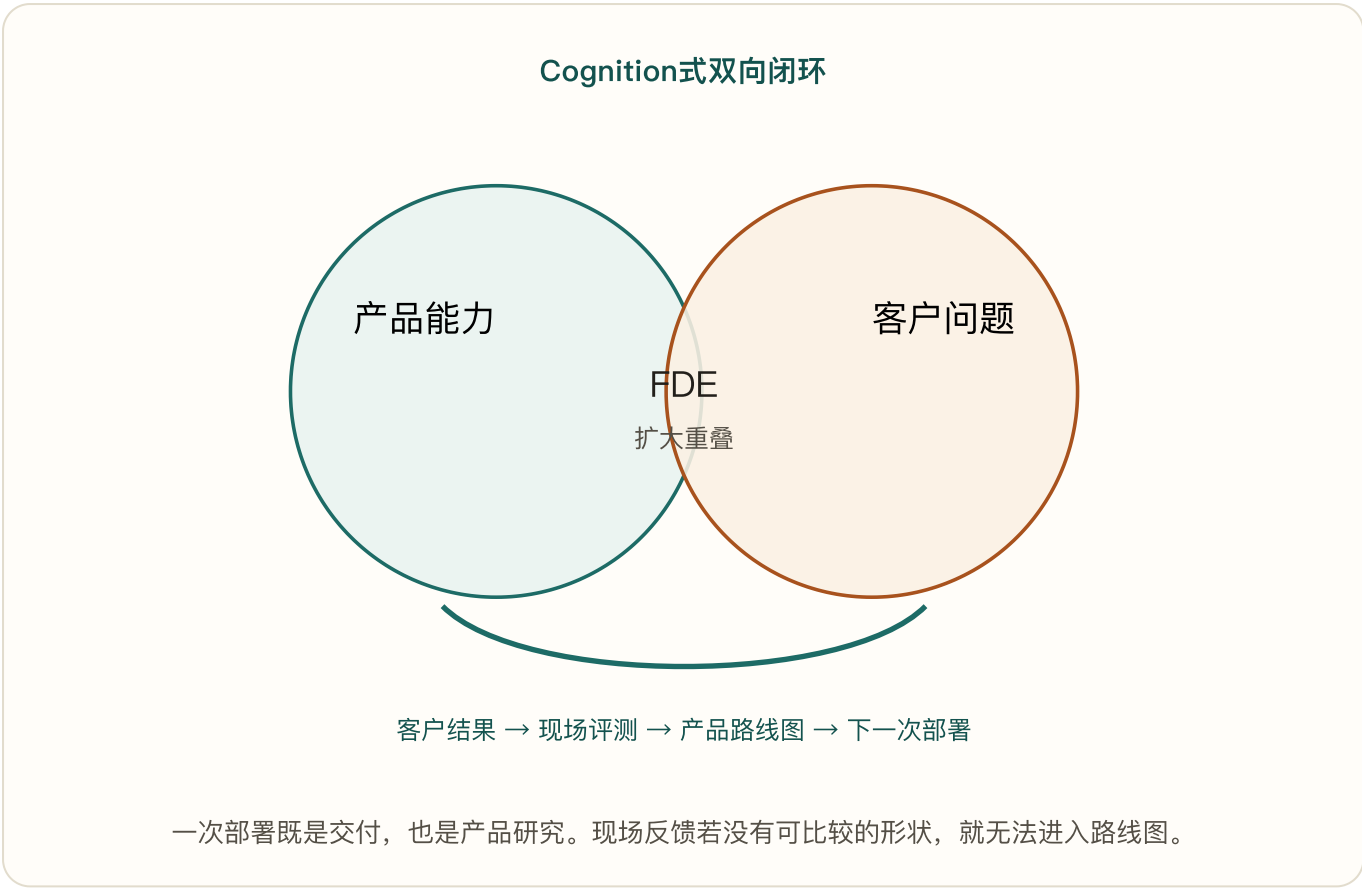
- 这是所有企业都会遇到的挑战，还是一名用户的特殊习惯？
- 一个临时绕行方案（workaround）是否暴露了产品缺口？
- 这项功能解决后，会影响多少部署和多少收入？
- 模型在哪种代码库、工具链或流程中系统性失败？

Wu把客户现场称为最高保真的评测集。实验室基准用预先定义的任务衡量能力；企业现场同时包含模糊目标、权限、遗留代码、人员协作和实际代价。它不仅告诉产品“模型做错了”，还告诉公司“这个错误为什么值得优先修”。

高保真也意味着高噪声。客户问题可能来自产品缺陷，也可能来自错误配置、缺少权限、代码库本身没有测试，或组织不愿改变审查流程。FDE的任务不是把每次失败原样转发给产品，而是给失败建立形状：在哪类仓库出现，前置条件是什么，有多少客户重复遇到，修复之后能解锁哪类任务。

这样整理过的现场证据能够降低路线图风险。产品团队不必只凭最大客户的声音或最激动的销售请求排优先级，而可以看到某个能力缺口影响多少部署、价值多大、是否已有可靠评测。workaround也不再只是临时技巧：如果多支团队都用同一种绕法，它很可能指向应该原生支持的功能。

> **核心概念：双向产品市场契合**
>
> FDE一边把产品能力映射到客户问题，一边把客户问题映射回产品能力。第一条路径创造当期价值，第二条路径降低路线图风险。只有两条都运行，产品与市场的重叠才会持续扩大。



一次部署既是交付，也是产品研究。现场反馈若没有可比较的形状，就无法进入路线图。

从token-maxxing到可测结果

早期Agent市场常把使用量当作成功：更多用户、更多session、更多token。模型成本被补贴时，这种指标看起来像增长。进入大型和受监管企业后，预算负责人会问：这些消耗改变了什么？

Token指标还有一个结构性问题：供应商收入可能随使用量增加，客户成本也同时增加。若系统把失败重试、无效探索和过长上下文都算成“采用增长”，双方利益并未对齐。结果指标则迫使团队面对更严格的问题——一美元模型成本换来了什么，节省的人类审查时间有没有被别的瓶颈吃掉。

Wu把下一阶段称为智能编排：Agent承担合适任务，并用结果衡量。一个CLI或IDE可以让个别工程师更快，但要让整个组织更快，还要改变积压管理、代码审查、发布流程和非技术协作。组织速度不是个人速度简单相加。

这也是她对单点工具边界的判断。一个很强的IDE助手可以让最积极的工程师效率显著提高，但企业总体产出还受项目选择、依赖协调、合规审查和上线节奏约束。只有把Agent接入这些组织流程，并让技术与非技术参与者都能看见和信任结果，个体速度才可能转化为组织速度。

什么样的人适合这项工作

Cognition寻找“T型而带尖峰”的人。横向需要客户沟通、业务、流程和技术理解；纵向可以来自不同强项：有的人像产品经理，擅长设计产品组合；有的人是创始人型通才；有的人拥有很深的工程专

长。

Wu并不要求每个人在所有维度同样强。产品经理背景的人如果擅长判断产品怎样嵌入流程，可以在产品设计上形成尖峰；创始人型人才习惯在模糊条件下跨职能推进；深技术工程师则能在陌生代码库和客户会议中成为真正的专家。业务感可以通过现场逐步培养，技术可信度却很难只靠话术替代。

团队反复追问两个问题："为什么要解决这个问题？"以及"怎样让这个经验改善所有客户？"前者防止在低价值任务上烧token，后者把现场劳动变成产品资产。

这套工作也要求相当强的客户承诺。Wu提到团队曾让一名成员在巴西靠近客户生活约十个月，只为确保部署成功。不是每个项目都需要如此长期驻场，但它说明Cognition理解的"deployed"并非偶尔做演示，而是愿意进入客户的节奏，对结果负责到系统真正运行。

她最后说"每个人都是go-to-market"。这不是要求所有工程师去卖软件，而是强调产品、现场和商业不能互相甩锅。产品修复一个现场缺陷是在帮助进入市场，FDE让客户成功是在验证产品，客户反馈改变路线图又回到工程。只要企业购买的是实际工程结果，团队就共同处在价值交付链上。

本课小结

Cognition不把Agent使用量当成终点，而把交付周期、有效产出和组织能力当作结果。FDE既要把Agent接进客户的完整软件生命周期，也要把现场失败整理成高保真评测与路线图证据。客户成功和产品进化是同一个闭环的两半。

到这里，我们已经有了问题、范围、流程、产品化和评测。最后一课把这些零件放进一座完整的软件生产系统：信号怎样进入，Agent怎样构建，验证怎样约束，发布之后又怎样产生下一轮信号。FDE也将在这里从单个项目负责人变成软件工厂的设计者。

想一想

1. 你们现有AI指标中，哪些只是活动量，哪些真正接近业务结果？
2. 客户提出的一个bug，怎样判断它值得进入产品路线图？
3. 让整个工程组织变快，除了写代码还要改变哪三条流程？

第八课　从编码Agent到软件工厂：验证循环决定自主上限

这堂课要讲清

- Factory如何把软件开发画成一条不断回流的工厂循环；
- 为什么模型无关的Agent框架与集中治理对大型企业重要；
- Agent准备度如何由确定性验证循环决定；
- FDE怎样从一个样板项目扩展到成千上万代码库。

谁在讲

Eno Reyes 是 Factory 首席技术官（CTO）兼联合创始人。Factory把企业编码Agent称为Droid，并在2026年把产品叙事从单个编码Agent扩展到“软件工厂”：一个由信号、规划、Agent、验证和治理组成的端到端系统。

本课对应演讲：How Forward Deployed Engineering is done at Factory。

先补背景：模型、Agent框架与软件工厂

模型负责理解和生成，真正让它在企业里工作的还有一层Agent运行框架（harness）。框架向模型提供上下文和工具，限制权限，记录每一步轨迹，控制成本，并决定任务失败后怎样恢复。模型可以更换，企业的治理、数据和验证体系不能跟着丢失。

软件工厂则是更大的系统：客户反馈、缺陷和业务要求先变成计划，计划进入代码和文档，改动经过代码审查、测试、安全扫描、lint（代码静态检查）与类型检查，最后发布并产生新的运行信号。SAST是在不运行程序时检查危险代码模式；lint检查可疑写法；类型检查确认数据是否按预期流动。这些确定性反馈越完整，Agent越能在较少人工打断下完成长任务。

软件开发本来就是一座工厂，只是多数公司没有画出来

Reyes把软件组织描述成一个循环。外部信号先进入：客户反馈、错误报告、内部对话和业务要求；团队排序、规划并形成任务；代码与文档成为事实来源；更改经过审查、测试、安全扫描、静态分析、lint与类型检查；最后发布、监控，再产生新信号。

“信号”比“需求”更宽。客户在电话里抱怨一项功能，生产监控出现异常，Slack里有人发现重复手工操作，管理层决定进入新市场，这些都可能成为软件工作的起点。人类团队会给信号赋予不同权重，把一部分变成计划，再由工程师写进代码。软件上线后产生的新错误、使用数据和反馈，又返回循环开头。

这条循环在每家公司都存在，却经常只存在于人的协作习惯里。某位产品经理知道哪条客户反馈应该进入季度计划，某位资深工程师知道安全审查要找谁，某个团队用自动测试，另一个团队依靠人工质量保证（QA）。单个环节都有工具，环节之间却没有统一状态。所谓软件工厂，首先是把这套隐式生产过程画出来，再判断AI能在哪些连接处持续接手。

很多公司拥有这些部件，却没有把它们设计成系统。信息散在工单、Slack、代码库和个人经验里，验证标准也因团队而异。编码Agent进入这种环境后，能生成代码，却不知道完整生产链怎样闭合。

Factory的FDE站在“产品最前端”：进入大型客户，把这条循环显式化，将现场信号送回Droid和平台。它不是传统专业服务，因为目标不只是完成客户项目，而是改变Agent框架与软件工厂本身。

Reyes专门划清了与咨询项目的边界。如果客户拿着大型咨询公司的报价，请Factory用Droid代为完成一场代码现代化，项目本身可能带来可观收入；但如果Factory工程师只是替客户执行迁移，公司得到的更像项目收入，产品未必因此变强。Factory不希望FDE成为使用自家工具的专业服务团队。

“矛尖”意味着另一种工作：FDE进入最复杂、最关键的客户，听工程负责人怎样规划，也观察一线工程师怎样使用Agent；把安全、成本、可观测性、模型路由和运行框架里的问题持续传回产品。客户项目仍要产生结果，但结果应该通过产品系统实现，并推动这套系统更容易在下一家企业“自组装”。

自组装并不意味着真正零配置，而是规模约束。大型客户可能有4.5万名乃至更多工程师、数万代码库，任何依赖Factory人员逐个安装、调试和维护的方案都无法覆盖。FDE要帮助建立一套部署模型，让客户团队可以按统一治理逐步扩张，而不是把现场人员变成永久操作员。

模型不是系统

企业很少愿意把命运押在单一模型上。模型价格、能力与政策变化很快，不同任务适合不同模型；敏感环境还要求私有网络、本地或隔离部署。Factory强调模型无关的框架（*model-independent harness*）：模型可以替换，任务上下文、权限、工具、轨迹数据、验证与治理仍由企业掌握。

所谓harness，可以把它想象成模型周围的工作环境。它决定Agent能看见哪些代码，能调用哪些工具，任务运行多久，失败后从哪里恢复；也记录模型采取过的动作、花费和验证结果。相同模型放进不同harness，实际能完成的工作可能完全不同。企业采购的若只是一个模型入口，便很难形成自己的运行资产。

模型无关还有现实的性能理由。代码生成、长上下文分析、视觉检查和快速分类未必由同一模型最擅长；价格与延迟也不同。平台可以按任务路由模型，但必须把路由决定和结果纳入同一套审计。否则"多模型"只会变成多个互不相通的风险边界。

Reyes甚至用"潜艇里也能运行"形容隔离能力。这个说法强调的是边界：Agent系统不能假设永远连着公共互联网或把代码发送到外部服务。对大型企业，采用速度往往由安全、数据所有权和集中治理决定，而不只是基准分数。

在金融、医疗和政府环境中，代码、运行日志乃至提示上下文本身都可能是敏感数据。客户不仅要知道模型输出了什么，还要控制数据流向哪里、哪些团队可以调用哪些模型、轨迹保留多久。Factory强调企业应拥有软件工厂中流动的trace（Agent执行轨迹）和数据，因为这些记录会成为后续评测、改进和合规的基础。若记录全部被封在供应商系统里，企业很难真正演化自己的工厂。

"软件工厂靠建设、不是买来即用"（built, not bought），并不是说企业不能采购平台，而是说软件工厂无法靠一次安装凭空出现。平台提供Agent、编排、治理和观察的构件，客户仍要投入工程工作，把自己的信号、代码、验证和发布路径连接起来。FDE的责任是帮助客户建立这套能力，而不是假装一个许可证已经完成组织改造。

Agent准备度，不是一句"模型更聪明了"

Reyes提出一个有用判断：一个代码库能让Agent自主工作多长，取决于它拥有多少高质量、确定性的验证循环。

类型检查能明确告诉Agent某类错误；单元测试和集成测试能验证行为；安全扫描能发现已知风险；编译与部署检查能确认更改能否进入环境。验证越及时、越可机器读取，Agent越能自己发现偏差并修正。

确定性在这里很重要。测试通过或失败、类型匹配或不匹配、安全规则命中或没有命中，都能转化成明确反馈。Agent可以据此修改代码再运行，形成密集循环。若反馈只是"资深工程师觉得不太对"，它既难自动读取，也往往到代码审查末端才出现，Agent此前的长链工作便可能全部建立在错误方向上。

反过来，如果完成标准只能由一名资深员工"看起来差不多"判断，Agent即使会写很多代码，也很难长时间自主工作。

Reyes把这一点与模型后训练联系起来。复杂任务中的模型需要密集奖励信号来保持方向；生产环境里的编译、测试和检查正是类似信号。验证器数量不是越多越好，关键是覆盖真实失败并快速返回。一

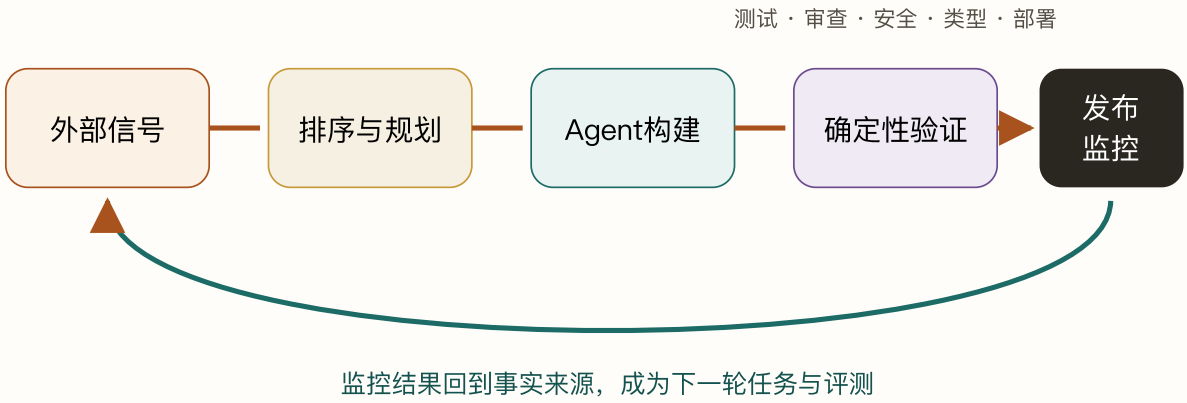
个永远通过的测试不会增加准备度，一个运行六小时且结果模糊的检查也很难支撑长任务。

Reyes把这种条件称为Agent Readiness（Agent准备度）。提升Agent准备度常会暴露企业原有的工程问题：构建无法在干净环境重现，测试互相影响，代码所有权不清，安全检查只在上线前人工进行。即使最后不使用Agent，修复这些问题也会改善人类工程师的交付。因此，FDE带来的第一批价值有时不是更多AI代码，而是让软件系统第一次拥有可重复的完成定义。

核心概念：Agent Readiness

Agent准备度不是采购了哪种模型，而是任务是否有清晰边界、可用上下文、足够工具和可靠验证。一个任务若能明确证明“完成”，通常比开放式任务更适合自主Agent。提高准备度，往往要先改善软件工程本身。

软件工厂：每一次验证都给Agent一条可学习的反馈



Agent自主性的上限由反馈系统决定。没有验证，更多生成只会更快地积累不确定性。

长任务需要有界、可验证的任务容器

Factory把长时间运行、多步骤、多Agent协作的任务称为Missions。关键不是让Agent“自由工作几天”，而是把任务放进边界清晰的容器，持续保留上下文、检查点和验证。

Missions把规划阶段和执行阶段分开。人先给出一个有界任务，说明目标以及“完成”由哪些检查证明；随后系统持续投入推理和工具调用，直到通过验证或触发升级。人的角色不是每隔几分钟继续提示，而是在开始前设计任务，在出现无法形式化的判断时介入。

Reyes的强主张是：如果“完成”能够被验证，许多任务今天就能交给AI解决。现实中仍有大量完成标准难以形式化，例如视觉质量、界面闪烁或复杂用户体验；这正是FDE要与客户一起建设新验证器的地方。

演讲提到大型代码迁移、合规与生物医药等场景，以及客户环境中约15%至20%的当前自主程度和更高的潜在上限。这些是Factory的项目经验和估计，不代表普遍能力。值得保留的是衡量方法：不要问“Agent会不会写代码”，要问“某条工作流中有多少步骤能在治理边界内自主完成”。

Reyes提到数千万行代码迁移、金融研究和生物医药问题，意在说明只要任务能被验证，长时Agent可以进入非常复杂的领域。这类陈述必须连同条件一起读：不是模型下载后就能处理数千万行代码，而是组织先要建立环境、拆分任务、提供工具和验证器。任务规模并不会取消工程约束，只会放大它。

Factory还区分了“有多少工作完全自主完成”与“每次人工干预之间Agent能连续执行多少动作”。演讲中的15%至20%和“80%以上”分别接近这两类口径，不能混成一个自主率。某个系统可能只有少数任务真正端到端无人参与，但在这些任务里，Agent已经能连续执行很长。准确区分指标，才能知道下一步应该扩大任务覆盖，还是减少现有任务的中断。

开放式视觉问题仍然是边界。终端界面是否闪烁、交互是否让人困惑，有时很难被现有自动检查稳定捕捉。Factory自己的核心harness也不能因此宣称完全自主；相反，团队需要设计新的视觉验证或保留人工判断。这再次说明，自主上限不是一条固定的模型能力曲线，而是验证工程不断向外推进的边界。

低垂果实之后，难的是重设计流程

最容易的一批任务也许占30%到40%：文档、简单测试、模式明确的迁移和维护。剩余部分通常要求团队改变工作流、补齐验证、整理代码库和权限。企业采用Agent的后半程不是继续换更强模型，而是重构软件生产系统。

Reyes说，大约三四成“低垂果实”可以直接让Droid修复，例如补齐明确的静态检查问题。剩余部分之所以难，不一定是代码更复杂，而是会触碰人的工作方式。更严格的自动检查可能让开发者觉得过度挑剔，新的合并门槛会改变团队节奏，补测试又需要先明确历史行为。FDE必须设计过渡，不能用Agent效率为名突然把整套开发流程强加给人。

这一步要求同时理解技术和组织。验证策略要足够强，才能给Agent反馈；又不能让人类工程师每天被低价值警告淹没。权限要允许Agent执行工作，又不能绕过责任边界。优秀设计通常从少数高价值流程开始，用实际缺陷下降、周期缩短或发布稳定性证明改变值得，再逐步提高标准。

这也是FDE存在的理由。它必须同时理解客户的代码库、治理、安全与业务目标，找出第一批能证明价值的任务，再把成功模式扩到成千上万代码库。

Epcot：从未来城市构想变成主题公园的警示

Reyes用Epcot做比喻。迪士尼最初把它想象成未来城市的实验样板，后来成为主题公园。企业AI试点也常如此：为了展示而搭建的“未来工作方式”非常漂亮，却依赖特殊团队、手工数据和例外权限，无法复制到普通代码库。

Epcot原本是“未来实验原型社区”的构想：先建一座可运行的未来城市，让其他城市借鉴交通和规划。最终落地的却是一座供人参观的主题公园。它可以启发想象，却不等于普通城市能够照着迁移。企业AI样板同样容易因环境过于精心设计，被业务团队视为一次展览。

好的样板必须既展示未来，又允许组织真正走到那里。它要使用可以扩张的身份、权限、验证和治理；要记录成本与失败；要证明普通团队而不只是明星工程师能够重复。

样板太保守也不行。如果它与旧流程几乎没有区别，只节省几分钟，组织看不到改变的必要。FDE需要走一条窄路：展示足够先进的未来，让人相信软件工厂值得建设；同时保留现有组织能理解的接口、控制点和迁移步骤，让团队相信自己走得到。

成功样板的真正产出不是演示视频，而是一套扩张模型：哪些代码库最适合下一批，最低验证要求是什么，谁负责治理，采用成本和结果怎样衡量。客户团队拿到这套模型后，应能自己把能力复制到更多部门。FDE若永远是唯一会操作样板的人，项目就仍然停留在主题公园阶段。

Factory寻找的FDE也由此带有混合特征：技术沟通能力、业务判断、管理层表达、系统思维，以及像创始人一样从模糊问题走到完整结果的倾向。

Reyes特别看好三类人。前创始人习惯在没有清晰边界的情况下同时处理产品、技术和商业；开发者体验或高质量开发环境背景的人，已经在思考怎样让其他工程师更有效；系统型人才则喜欢建模组织中的流动、关闭反馈循环。产品经理如果愿意快速补足技术深度，也可能适合，因为软件工厂的每个环节——代码审查、事故响应、QA、测试——都可能成长为独立产品面。

人的长期位置也随之改变。工程师不再只直接修改软件，而会越来越多地设计、维护和改进“构建软件的系统”：选择信号，设定验证，观察失败，调整工具和治理。这不是工程工作的消失，而是抽象层级上移。FDE处在这个变化最前面，因为他必须先在客户真实组织里证明新层级如何工作。

最好的Agent改造，常从改善工程基础开始

补测试、统一构建、整理依赖、明确所有权，这些工作看起来不像“最前沿AI”，却直接提高Agent能安全自主工作的范围。FDE若只带来模型，没有带来验证系统，扩张会很快撞墙。

本课小结

Factory把编码Agent放进一座完整软件工厂：信号进入规划，Agent构建，确定性验证把错误送回，发布与监控产生新信号。模型无关框架、数据与治理所有权，以及可机器读取的完成标准，共同决定自主上限。FDE负责让样板不是主题公园，而是可扩张的生产系统。

想一想

1. 你的代码库有哪些完成标准仍只存在于资深工程师直觉里？
2. 若明天替换底层模型，哪些上下文、轨迹和治理能力仍由你掌握？
3. 现有AI样板中，哪些特殊条件阻碍它复制到普通团队？

课后总结

八位讲者真正同意什么，又在争论什么

八场演讲表面上都在谈FDE，实际上站在不同产品、客户与公司阶段上。把差异抹平，会得到一句没有用的话："FDE就是贴近客户的工程师。"更有价值的做法，是看清他们在哪些地方达成一致，又为什么给出不同组织答案。

共识一：真实问题不在需求文档里

Kepler看用户怎样双击CSV，Varick追踪发票匹配失败后转给谁，Ramp追问客户是否真的需要Android，Decagon分析历史支持数据来挑战客户的自动化顺序。四家公司用不同语言说明同一件事：客户最初说出的通常是一个方案，不是已经完成的问题定义。

FDE必须进入工作的因果链。只收集功能名词，会让团队在错误问题上高效执行。

共识二：执行越来越便宜，判断没有同步变便宜

Varick和Cognition都用很强的语言宣称编码或知识工作的执行不再是主要瓶颈；Ramp和Decagon则更谨慎地描述后果：生成变容易之后，最稀缺的是范围、克制、评测和品味。

这些说法强弱不同，却指向共同结构：代码供应增加，不会自动增加正确需求。相反，组织更需要一道机制阻止"能做"直接变成"应该做"。

共识三：一次性交付必须回到产品

Factory称FDE是产品的矛尖，Cognition把现场视为高保真评测集，Decagon追求custom→self-serve，Kevin Bai强调共享平台，Kepler把FDE称为产品战略。没有一家主张长期依靠完全孤立的客户代码扩张。

区别只在回流对象：有的回到平台原语，有的回到模型评测，有的回到Agent配置界面，有的回到行业方法和组织记忆。

共识四：生产采用，而不是演示，是完成标准

Sierra回顾的最早FDE要在凌晨两点保证平台不倒；OpenAI当前FDE职位也把成功写成生产采用、可测工作流影响和基于评测的反馈。八场演讲都拒绝"原型能跑即完成"的定义。

生产意味着系统必须穿过权限、数据、旧系统、异常、责任、维护和用户习惯。FDE不是把demo送到客户，而是把变化送进日常工作。

共识五：FDE不能永远依赖英雄

演讲者普遍承认优秀FDE兼具技术与沟通能力，像创始人一样主动。但成熟体系又必须减少对单个英雄的依赖：Kevin Bai提倡结对，Varick造FDE Agent，Ramp把scoping变成内部系统，Decagon专业化并建设自助漏斗，Factory建设统一治理与验证。

英雄能完成第一个项目；系统决定第十、第一百个项目。

分歧一：FDE是一个角色，还是一个过渡阶段

Natalie Meurer认为名称已经装进太多工作，未来产品工程师更面向客户，各种角色会继续模糊。Kevin Bai则给出很清晰的岗位条件：复杂产品、非技术买家、共享平台。两者并不真正矛盾。

Bai回答的是“什么时候需要一组特殊责任”；Meurer追问的是“这些责任是否必须永远集中在同一个职位名称里”。早期或复杂部署可能需要显式FDE团队；随着产品自助和整个工程组织更贴近客户，一部分工作会被吸收、自动化或重新命名。

分歧二：FDE应该属于工程，还是属于GTM

Decagon把FDE与产品工程置于同一门槛和组织，Ramp也将其放在工程内；Cognition则说“每个人都是GTM”，因为全员目标是客户成功。Palantir的公开职位体系把直接面向客户、对技术与运营结果负责的前向部署软件工程师称为Deltas，把建设核心平台产品的软件工程师称为Devs，同时强调两类角色有意重叠。

组织归属没有唯一答案。更重要的是三条通路是否畅通：FDE能否写生产代码，能否影响产品优先级，能否理解交易与采用。若任何一条被组织墙切断，名称放在哪里都无济于事。

分歧三：先做快速补丁，还是先建设通用系统

Kepler鼓励一天以内先交付，Decagon强调不要被一次性补丁诱惑。这是FDE最真实的一对张力。

答案不是选边，而是把“范围”与“质量”分开：可以快速做一个范围很小、用于学习的版本，同时明确所有者、验证和退出条件；不能在没有学习目标的情况下，快速做一个大而隐蔽的客户分支。速度应该缩短反馈周期，通用化应该在重复模式出现后发生。

分歧四：AI是FDE的工具，还是未来的FDE

Varick明确建设FDE Agent，Ramp自动化FDE生命周期，Factory与Cognition让编码Agent承担更多实施；与此同时，所有讲者都把上下文发现、范围、架构判断、客户责任和组织变革留给人。

近期更准确的图景不是“AI替代FDE”，而是FDE的任务重新分层：资料整理、小型变更和明确实现逐渐自动化，人被推向更高密度的判断与责任。未来如果Agent能承担更多判断，它也仍需要一套可审计的升级、评测和责任系统。

一张比较表

公司/讲者	他们认为最稀缺的东西	FDE最重要的回流	最大失败模式
Sierra / Meurer	连续客户责任	角色能力与平台成熟经验	用一个职位名掩盖互相冲突的期待
Anthropic / Bai	复杂平台到非技术用户的桥梁	共性需求进入平台原语	没有平台却不断接定制项目
Kepler / Ganesh	从现场看见真正问题	小解法沉淀为产品战略	按客户方案做47页系统，错过一个行动
Varick / Moza、JD	隐性流程与组织上下文	依赖图、知识和FDE工具	把AI贴在坏流程或要求客户推倒旧系统
Ramp / Mehr	scoping、品味与优先级	规范、rubric与Agent流水线	token驱动的无效开发
Decagon / Rekhi	克制、系统思维与行业经验	custom变self-serve	提示词和补丁堆成客户无法拥有的黑箱
Cognition / Wu	可测业务结果	现场问题变高保真eval与路线图	用token和session代替价值
Factory / Reyes	验证循环与系统治理	软件工厂、harness与agent readiness	样板像主题公园，无法扩到普通代码库

核心概念：FDE是一套压缩学习周期的组织设计

FDE把原本分散在销售、咨询、产品、工程和客户成功之间的信息放进更短循环：现场事实更快到达能构建的人，构建结果更快回到真实用户，失败模式更快进入产品与评测。名称和汇报线可以不同，学习周期是否缩短才是核心。

总结

八位讲者共同相信客户现场、生产责任和产品回流；他们的分歧来自公司阶段、产品形态与组织边界。FDE可以是一支团队、一个过渡角色，也可以逐渐变成所有工程师的工作方式。唯一不能丢的是：有人把真实问题、可验证结果和产品学习连起来。

八堂课解释了FDE为什么存在，也展示了八种不同的现场。现在把方法收拢成一套可执行的操作系统。它不要求公司照搬任何一家讲者的组织图，而是为一次部署设置九道关口。每道关口都应有明确产物和责任人。

第一道：选择值得深度部署的客户

FDE资源昂贵，不能平均分给所有机会。优先选择同时具备三项条件的客户：问题价值足够高；愿意开放真实流程与数据；需求可能代表更广市场。仅有大合同而没有合作意愿，团队会被困在信息不全的承诺里；仅有有趣技术而没有业务价值，试点很难进入生产。

产物：客户选择简报。 写明业务价值、平台契合、学习价值、数据与决策者可达性，以及不做的理由。

第二道：建立说话者与利益相关者地图

客户不是一个人。购买者、流程负责人、最终用户、IT、安全、数据、法务和高管对成功有不同定义。先画出谁拥有预算、谁每天使用、谁能否决、谁在失败时承担代价。

尤其要区分“提出需求的人”和“承受工作的人”。销售或管理者说要一个仪表板，星期一早上真正操作的人也许只需要一条告警。

产物：利益相关者表。 每人记录目标、权限、担忧、成功指标与当前替代办法。

第三道：画两张流程图

第一张是官方流程：文档规定应该怎样工作。第二张是真实流程：正常路径、异常、绕行、等待、复制粘贴、电话和个人记忆。

不要只问“下一步是什么”，还要问“出错时怎样办”“上一次例外发生在什么时候”“如果这个人休假呢”。能现场观察就不要只访谈。

产物：现状地图。 每一步标记输入、负责人、系统、前置依赖、时长、失败方式与升级路径。

第四道：把愿望改成可验证结果

“部署AI”“提高效率”“改善客服”都不可验收。可验证结果要包含对象、行为、基线、目标、时间和边界。例如：在一个渠道内，对三类高频意图自动完成身份校验与状态查询；出现退款或身份不一致时转人工；四周内比较解决率、转人工率和错误率。

这里还要写反指标。自动解决率提高，但投诉、返工或风险也提高，就不算成功。

产物：结果合同。 不是法律合同，而是客户、销售、产品和工程共同签认的一页成功定义。

第五道：压缩到最短行动闭环

找到一段能在几天而不是几个月内改变真实动作的范围。删掉暂时不需要的渠道、平台、边缘功能与完美界面，但保留日志、权限、测试、人工接管和数据边界。

一个好首版具备三点：有真实用户；有可观察结果；失败能安全停止。

产物：首个价值计划。 明确第一位用户、第一条路径、第一项指标、最迟反馈日期和停止条件。

第六道：设计人机控制面

对流程每一步评估四项：错误代价、结果可验证性、发生频率、所需判断。低风险且可验证的步骤可自主执行；高频但有风险的步骤适合人在回路；低频、高风险、高判断密度的步骤保留人工。

同时明确Agent权限：读什么、写什么、可调用哪些工具、一次能造成多大影响、何时必须升级给人。

产物：控制矩阵。 每一步写清自动化等级、验证器、审批者、回滚和审计记录。

第七道：进入生产，而不是停在演示

生产检查至少包括：身份与权限、数据来源、版本、可观测性、评测、错误预算、回滚、人工接管、支持时段、所有者和变更流程。对于Agent，还要保存任务轨迹、工具调用与关键决策依据，使失败可以复盘。

“模型偶尔会错”不能成为模糊豁免。概率系统更需要清楚的风险边界。

产物：生产就绪清单。 每项有负责人和证据，不以口头“应该没问题”通过。

第八道：把一次性资产分类回流

每个项目结束一个周期后，把新东西分到四个篮子：

1. 客户特有配置：留在客户空间，有版本和测试；
2. 行业模式：沉淀为模板、数据模型、评测集和方法；
3. 平台共性：进入产品路线图，成为原语、界面或接口；
4. 交付系统：进入FDE自己的工具、知识库、rubric和自动化。

产物：回流账本。 记录模式出现次数、影响客户、当前临时绕行方案（workaround）、通用化方案、所有者和截止时间。

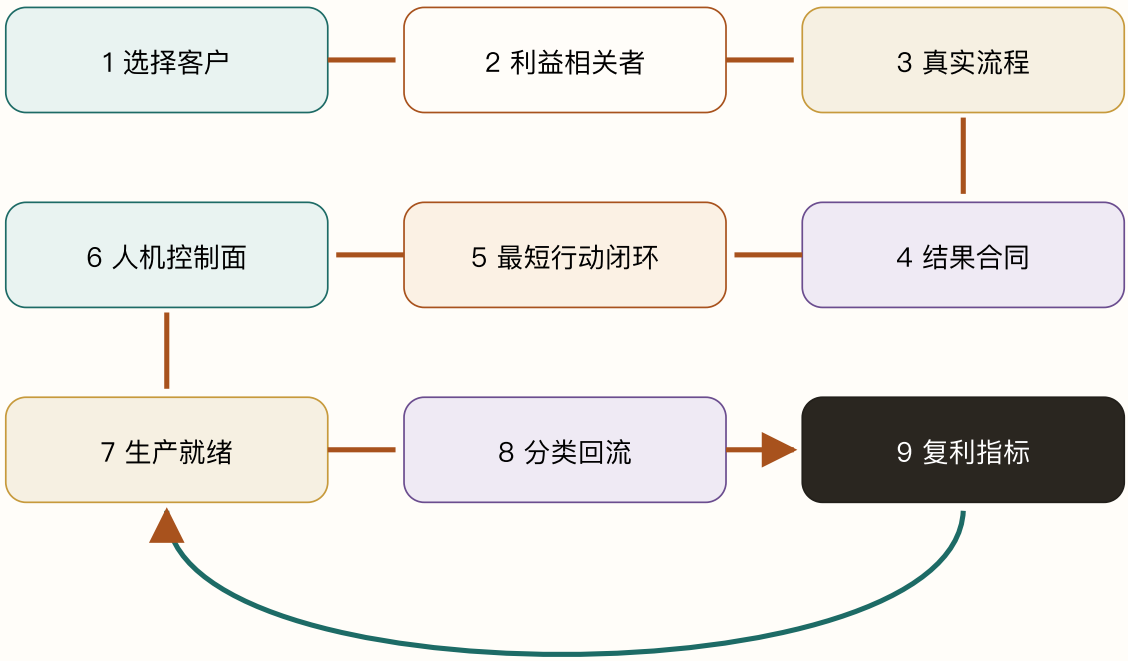
第九道：用复利指标审视团队

FDE不能只看签约额、工单关闭数或代码行。至少同时看四组指标：

- **客户结果**：采用、业务指标、稳定性、风险与满意度；
- **交付速度**：从问题确认到首个价值、从首版到生产的时间；
- **产品回流**：多少重复需求成为平台、自助或标准评测；
- **组织韧性**：一名成员离开后能否接手，知识是否成对和共享。

一个很有力量的总问题是：**第十次部署是否比第一次更快、更稳、更少依赖特殊人物？** 如果答案长期是否定的，增长只是项目数量增加，不是能力复利。

FDE操作系统的九道关口



复盘结果回到客户选择与方法设计，形成下一轮复利

每道关口都需要可审阅产物。流程的作用不是增加官僚手续，而是让判断不再只存在于个人脑中。

三种团队形态

不同阶段可以选择不同形态。

创始人型。 客户少、产品未定型，最强工程师或创始人直接部署。优点是学习极快，风险是英雄依赖和无边界承诺。

专门FDE团队。 客户与模式增多，需要明确负责人、方法和平台接口。优点是连续责任，风险是与产品工程分裂成“接需求”和“做正事”两类人。

全工程前向化。 自助能力成熟，产品工程师轮转或直接参与客户，专门FDE保留在最复杂场景。优点是产品反馈直接，风险是普通路线图被高声量客户绑架。

团队可以在三种形态间移动。选择标准不是潮流，而是产品成熟度、客户复杂度和知识回流速度。

招什么样的人

八场演讲合起来，理想画像有五项：能写生产代码；能把模糊问题拆成最小结果；能与技术和非技术人沟通；能在压力下承担端到端责任；能从单个案例看出可复用模式。

不必要求每个人五项满分。可以搭配工程尖峰、产品尖峰、行业尖峰和客户尖峰，再用结对与统一工程门槛保持质量。最危险的招聘描述，是列出十年经验的“独角兽”，却不解释公司用什么平台、流程和团队支撑他。

实践手册小结

FDE操作系统把英雄式经验变成九道可审阅关口：选择客户、识别利益相关者、画真实流程、签结果合同、压缩首个价值、设计人机控制、进入生产、分类回流、衡量复利。最终标准不是做了多少项目，而是部署能力是否随项目增长。

附录A　FDE术语表

本表按英文名称和缩写大致排序。它既收录基础技术名词，也收录本书反复使用的交付方法、组织概念和产品名称，方便跳读时集中查询。

Agent / AI Agent

能接收目标、调用工具、根据环境反馈继续执行任务的软件系统。与普通聊天模型相比，它更强调行动、状态和多步循环。本书保留英文Agent，不译作“代理”。

Agent Readiness / Agent准备度

一项任务或一个代码库支持Agent安全自主工作的程度，取决于上下文、工具、权限、边界和验证循环，而不只取决于模型能力。

Agent Builder / Agent Software Engineer

Decagon对两类现场工作的区分：前者主要在产品内配置Agent的意图、语气、动作和人工交接；后者处理需要进入通用产品的工程能力。名称不是行业统一标准。

AIP / AIP Bootcamp

AIP是Palantir将生成式AI连接到企业数据、工具与工作流的平台；AIP Bootcamp是与Palantir工程师一起，在小时或数天内将真实场景做成可运行用例的实战工作坊。

AP / AR / FP&A

AP是应付账款，处理企业应该付出的款项；AR是应收账款，处理客户应该支付的款项；FP&A是财务规划与分析，负责预算、预测和经营分析。

API

应用程序编程接口。一套软件按预先约定的方式向另一套软件请求数据或动作，例如查订单或发起退款。

AWS / EC2 / S3 / DynamoDB

AWS是Amazon的云计算平台；EC2实例是云中的虚拟服务器；S3是以存储桶组织文件的对象存储；DynamoDB是由AWS管理服务器与扩容的NoSQL数据库。

BI / Business Intelligence

商业智能。用查询、报表、图表和仪表板帮组织分析经营数据的工具与方法。

Cassandra / Keyspace

Cassandra是一种分布式NoSQL数据库；keyspace是它组织表、复制策略等数据定义的顶层命名空间。本书案例中，空日期导致系统错误地产生数百万个keyspace。

Change Management / 变革管理

让新的流程、工具和责任方式被组织真正接受并持续使用的工作，包括沟通、培训、控制点设计、分阶段切换和反馈处理。

CI / Continuous Integration

持续集成。代码被频繁合并到共享仓库，然后自动构建和测试，以尽早暴露冲突与缺陷。

CLI / IDE / Sandbox

CLI是输入文本命令的命令行界面；IDE是集编辑、运行和调试于一体的开发环境；Sandbox是与生产系统隔离、用来限制风险的运行环境。

COBOL / JCL

COBOL是银行、保险和政府大型机中仍常见的业务编程语言；JCL是IBM大型机上描述批处理作业如何运行的控制语言。它们代表企业软件现代化中常见的遗留技术。

Continuity of Customer Accountability / 客户责任连续性

同一个人或团队持续理解客户为什么购买、系统怎样构建、用户为何采用或拒绝，以及问题如何回到产品，避免责任在销售、实施和支持之间丢失。

CRM / ERP

CRM管理客户、销售和服务关系；ERP管理财务、采购、库存等企业资源。两者常是Agent必须读写、又不能破坏的核心系统。

Custom-to-self-serve / 从定制到自助

把反复由工程师手工完成的客户配置或集成，逐步变成产品中的标准能力，让客户、实施人员或Agent Builder可以自行完成。

Design partnership / 设计合作

供应商与少数客户共同发现并塑造产品的深度合作。FDE模式希望把它扩展到企业场景，同时保留可复用性。

Deterministic validation / 确定性验证

对同一输入给出清晰、可复现判定的检查，如编译、类型检查、单元测试、规则校验。它为Agent提供比主观评价更稳定的反馈。

DevOps

不是某一款工具，而是打通软件开发与运维，让团队共同负责构建、发布、监控和故障恢复的文化、方法与工具集合。

Eval / 评测

系统化检验模型或Agent在特定任务上表现的方法。企业评测不仅看答案质量，还要包含工具调用、风险、延迟、成本和业务结果。

ETL

Extract、Transform、Load，即抽取、转换、加载。它把数据从源系统取出、清理转换，再送入数据仓库或其他目标平台。

Forward Deployed Engineer, FDE / 前向部署工程师

直接与客户合作，将复杂技术部署到真实环境，并把现场反馈带回产品的工程角色。本书强调这是能力组合，不是统一职位标准。

Foundry

Palantir的核心数据运营平台。它把数据集成、业务对象、分析、应用和工作流放在同一套安全与治理体系中，不是一个单独数据库。

Generalization / 泛化

从一个客户的特殊需求中识别可跨客户复用的稳定结构，并把它迁回平台、接口或方法。它不是把所有客户强行做成一样。

Golden path / 黄金路径

流程在所有条件都正常时的标准路线。自动化常在黄金路径上演示成功，却在例外和错误恢复中失败。

GTM / DevRel

GTM（go-to-market）是产品如何被发现、购买、部署和采用的进入市场体系；DevRel（developer relations）通过文档、示例、社区与技术沟通服务开发者。

Harness / Agent Harness

围绕模型提供上下文、工具、权限、记忆、轨迹、验证和治理的运行框架。模型负责推理，Harness决定它在真实环境中怎样行动。

Human in the loop / 人在回路

Agent执行部分工作，人类在关键节点审查、批准、纠正或接管。它不是过渡阶段，也可以是高风险流程的长期控制设计。

MCP / Model Context Protocol

让AI应用以较统一的方式发现并使用外部数据源和工具的开放协议。MCP解决“怎样连”，不替Agent决定“什么时候应该调用”。

Ontology / 本体

把业务中的对象、关系和动作组织成统一模型。例如“客户”“账户”“计费实体”分别是什么，以及它们之间怎样关联。

Observability / Audit / Rollback

可观测性让团队知道系统正在发生什么；审计记录回答谁在何时依据什么执行了动作；回滚让错误变更可以撤销。三者是Agent从演示进入生产的基本控制。

Outcome-based Pricing / 结果定价

按已经定义并确认的业务结果收费，而不是只按席位或使用量收费。它会迫使供应商明确结果口径、归因方式和未达成时的责任。

Parquet / CSV

CSV是以文本逗号分隔表格字段的简单格式，便于直接打开；Parquet是面向高效存储与分析的开源列式文件格式，对机器更友好，对人却不如CSV直观。

PR / Pull Request

向代码库提交一组变更，请其他开发者审查并合并。“生成了代码”不等于“PR已被接受”，后者更接近组织认可的产出。

Rubric / 评分规则

把一项好结果应具备的条件写成明确检查项。例如需求规范是否说明用户、目标、非目标、权限和成功指标。Rubric帮助评测判断质量，不等于替代人的判断。

Primitive / 原语

平台提供的可复用基本能力，如认证、连接器、工作流节点、策略、评测或界面组件。客户方案由原语组合，而非每次从零开发。

Productization / 产品化

把一次性客户解法抽象成可被更多客户重复使用、由产品团队维护的能力。

Product-led Growth, PLG / 产品主导增长

主要依靠产品本身的试用、使用和自助购买推动获客与扩张。复杂企业AI常因数据、权限和流程集成而无法完全依赖这种模式。

Production Adoption / 生产采用

真实用户在正式环境中持续使用系统，并产生可测工作流或业务结果。它比“原型能运行”更严格，包含权限、支持、监控、回滚和责任。

SaaS

Software as a Service，软件即服务。用户通常通过网络按订阅使用由供应商持续运维和更新的软件，而不是买断一个静态版本。

SAP S/4HANA / NetSuite / Salesforce

S/4HANA是SAP的企业资源计划套件；NetSuite是Oracle旗下云ERP；Salesforce以CRM平台著称。它们常是企业Agent必须连接并写回的核心记录系统。

SAST / Lint / Type Checking

SAST在不运行程序时扫描已知安全风险；lint检查可疑写法和风格问题；类型检查验证数据是否按代码约定流动。它们都能给Agent较明确的自动反馈。

Scoping / 范围界定

在构建前澄清目标、用户、边界、依赖、优先级和成功标准，并主动删除与结果无关的实现。

Skill / 技能

供Agent重复调用的一组能力封装，可能包含指令、工具和执行步骤。它比一次性提示词更稳定，但仍需要权限、输入边界和结果验证。

Skywise

Airbus与Palantir合作推出的航空数据平台，将飞机、机队、维修、备件和航班等数据集成起来，用于航空运营与分析。

Software Factory / 软件工厂

把需求信号、计划、Agent执行、验证、发布和治理连成可观测反馈循环的软件生产系统。重点不是“无人写代码”，而是每一步都有状态和验证。

SRE / Site Reliability Engineering

站点可靠性工程。用软件工程、自动化和可量化目标提高线上系统的稳定性、容量与故障恢复能力。

Stakeholder Map / 利益相关者地图

记录购买者、流程负责人、最终用户、IT、安全、数据、法务和管理者各自目标、权限、担忧与成功标准的工作表。

SWE-bench

一套软件工程基准，使用真实GitHub问题及其修复评估模型能否生成通过项目测试的补丁。它比纯代码生成更接近修复真实仓库，但仍不等于完整企业交付。

System of record / 记录系统

组织用来保存权威业务数据和审计记录的核心系统，如ERP、CRM或财务系统。企业AI通常必须与之集成，而不是绕开。

Time to value / 价值实现时间

从项目开始到客户第一次得到可验证业务价值的时间。它比“代码开始写得多快”更接近部署质量。

Token / Token-maxxing

Token是模型处理文本和代码时使用的计量单位。Token-maxxing是本书引用的讽刺说法，指把大量模型调用和代码生成误当成价值，而不检查有效结果。

Trace / Trajectory / 执行轨迹

Agent一次任务中收到的上下文、采取的步骤、工具调用、结果和关键判断记录。轨迹是故障复盘、评测、审计和后续改进的重要材料。

Workflow Agent / 工作流Agent

围绕一条业务或工程流程持续执行多步任务的Agent。它必须理解状态、依赖、异常和人工交接，而不只是调用单个工具。

附录B　八位讲者与演讲

讲者	2026大会职位	本书所用演讲	对应课程
Eno Reyes	Factory CTO兼联合创始人	How Forward Deployed Engineering is done at Factory	第八课
Vasuman Moza	Varick Agents创始人兼CEO	AI tools for Forward Deployed Engineering	第四课
Jia Wu	Cognition Deployed Engineering Lead	How Forward Deployed Engineering is done at Cognition	第七课
Leo Mehr	Ramp Director of Engineering	How Forward Deployed Engineering is done at Ramp	第五课
Natalie Meurer	Sierra Agent Engineering负责人	The Dirty Secret of Forward Deployed Engineering	第一课
Sunny Rekhi	Decagon CTO of Forward Deployed Engineering	How Forward Deployed Engineering is done at Decagon	第六课
Vinoo Ganesh	Kepler CEO兼联合创始人	How Forward Deployed Engineering is done at Kepler	第三课
Kevin Bai	Anthropic Member of Technical Staff	Forward Deployed Engineering 101	第二课

注：Varick演讲后半段另由一位名为JD的工程负责人介绍FDE Agent的技术实现；字幕未给出可由公开资料可靠核验的完整姓名。大会FDE track另有其他场次；本书只整理用户提供字幕的八场。

附录C　思考题参考思路

开放题没有唯一答案。下面给出一套检查角度，方便自学或团队讨论。

1. **定位试点瓶颈**：把问题放进事实、判断、工程、产品、组织五层；优先找阻塞下一层的信息，而不是立刻补模型能力。
2. **给需求排序**：同时看客户业务价值、时间敏感性、可验证性、实现代价、跨客户复用和机会成本。销售紧迫感与客户紧迫感分开记录。
3. **判断是否产品化**：比较需求在不同客户中的“形状”，寻找稳定对象、动作和约束；重复不等于完全相同，适合抽象的是变化背后的结构。
4. **设计人在回路**：错误代价越高、结果越难自动验证，人工控制越靠前；频率高且规则清楚的步骤更适合自主。
5. **选择首个价值**：选真实用户、短反馈周期、可安全失败的一条行动闭环；宁可缩渠道和覆盖面，不要删除日志、权限与回滚。
6. **防关键人风险**：重要客户至少两人理解；关键会议、架构和变更进入共享记录；用代码审查、轮换和接管演练测试知识是否可转移。
7. **审计AI指标**：把token、session、生成行数归为活动指标；把采用、周期、有效产出、错误和业务变化归为结果指标，两类都看但不混用。
8. **检查样板能否复制**：列出试点使用的特殊权限、手工数据、明星人员和例外预算；每项都要有普通环境的替代方案。

附录D　资料来源与准确性说明

原始材料

本书正文主要改写自 AI Engineer YouTube频道于2026年7月28日发布的八段自动字幕。演讲来自 **AI Engineer World's Fair 2026** 的 Forward Deployed Engineering track。大会官方日程记录活动于2026年6月29日至7月2日在旧金山 Moscone West 举行，相关场次位于 Track 8、Room 2020。

内容来源致谢

感谢 AI Engineer World's Fair 与各位讲者公开分享这些演讲。点击标识可访问大会官网。AI Engineer World's Fair 的名称、标识，以及原始演讲、视频与字幕的相关权利归主办方、讲者及各自权利方所有；本项目只作非官方、非商业的学习性整理与来源识别，不主张对这些第三方材料的权利。

外部核验资料

- AI Engineer World's Fair 2026 官方日程：大会日期、场次、讲者与职位。
- AI Engineer World's Fair 2026 完整机器可读议程：FDE track 场次说明。
- Palantir Careers：Deltas、Echos与Devs的公开角色定义，以及现场解决方案向产品扩展的关系。
- Palantir架构概览：Foundry、AIP与Apollo的平台边界，以及Ontology对数据、逻辑、动作和安全策略的组织方式。
- Palantir Getting Started：AIP Bootcamp与“在小时或数天内从零到用例”的官方说明。
- Palantir：AI FDE Overview：AI FDE通过自然语言执行数据转换、代码仓库和ontology操作的产品边界。
- Airbus：Skywise发布说明：Skywise与Palantir的合作背景，以及飞机、机队、维修、备件和航班数据的集成范围。
- AWS：What is DevOps?、Amazon EC2、Amazon S3 与 DynamoDB：DevOps方法以及书中AWS基础设施术语的官方定义。
- Apache Cassandra数据定义 与 Apache Parquet：keyspace在Cassandra中的作用，以及Parquet列式文件格式的设计目标。
- SWE-bench FAQ：基准如何使用GitHub问题、对应修复和项目测试判定模型生成的补丁。
- OpenAI：Forward Deployed Engineer：2026年FDE职责与生产采用、工作流影响、eval反馈等成功标准。

- Ramp Builders：Forward Deployed Engineering：Leo Mehr对Ramp FDE历史、客户生命周期、scoping与generalization的完整阐述。
- Sierra：Natalie Meurer：讲者背景与Agent Engineer角色。
- Sierra产品概览：客服Agent的渠道、测试监控与按结果定价等产品背景。
- Decagon：How Decagon is redefining forward deployment：从英雄式定制向系统设计、客户自助与交付复利的转变。
- Decagon：Understanding agentic workflows 与 Introducing Duet：客服Agent的动作、生产纪律、持续评测与自助构建背景。
- Kepler About 与 Kepler Blog：Vinoo Ganesh经历、产品背景与可追溯验证主张。
- Factory 2.0：From coding agents to software factories：软件工厂循环、模型无关与组织级自主的产品背景。
- Factory：Software Factory：Agent编排、质量关口、治理、部署方式和结果指标的产品结构。
- Cognition：Introducing Devin 与 Cognition Blog：Devin的基本系统形态与2026年企业部署背景。
- Varick Agents：从业务流程出发，把Agent接入企业工具、数据与运营逻辑的公司定位。
- Varick：Solving the AI Adoption Problem 与 AI for Enterprise Finance：在NetSuite等记录系统之上部署Agent、从黄金路径走向异常处理的公司方法。

如何理解书中的数字

演讲者展示的效率、投资回报、人数、迁移规模和客户案例，大多来自公司内部项目或个人回忆。除非本书明确写明外部来源，否则它们只代表演讲口径，不能直接推广到其他公司。字幕简介与正文冲突时，以连续的演讲正文和官方专名为准；无法可靠识别的具体模型版本未强行写入。

非官方改写声明

本书是基于公开视频字幕的中文学习性改写，不是 AI Engineer、大会主办方或相关公司的官方出版物，也不代表讲者对中文表述逐字审定。为了阅读连贯，口语重复与招聘信息已删除，论证顺序、案例归属和重要限定尽量保留。

结语　真正被部署到现场的，不只是AI

FDE听起来像一种把技术送进客户环境的职业。读完八场演讲，会发现被重新部署的其实还有软件公司的组织方式。

产品不再能把客户问题简单交给售后；工程不再只对代码仓库负责；销售不能用“客户需要”替代证据；客户也不只是被动接收成品，而是参与定义工作怎样重构。AI越能快速执行，这些边界越必须被重新协商。

FDE不会解决所有问题。它可能成为昂贵定制的借口，也可能制造英雄依赖，让最大客户绑架路线图。防止这些失败的办法，不是远离现场，而是让现场工作可验证、可回流、可接手：问题被写成结果，结果进入生产，例外变成评测，重复变成平台，个人经验变成组织记忆。

最终，判断一家公司是否真正掌握FDE，不必看它招了多少人，也不必看职位名称多新。只要问一句：

> 每完成一次部署，这家公司有没有变得更懂客户，也更会做产品？

如果答案是肯定的，现场就不再只是成本中心。它是产品学习最快的地方。

本书由公开视频字幕改写整理，用于学习；并非大会、讲者或相关公司的官方出版物。
术语、人名与公开职位经核对；公司案例与数字保留演讲口径，详见附录“资料来源与准确性说明”。