氛围转变之后的软件工程

肖恩·戈德克

2025年4月

内容

前言

我读过很多关于软件工程的书，但我对这份工作的所有最重要的认识，都是从口口相传或痛苦的亲身经历中学到的。本书是我尝试把这些经验教训记录下来的成果。鉴于此，我想先提出一些注意事项。

我觉得我相当成功。我被提升为员工级工程师两次，参与过一些非常高优先级的项目，并且通常在我工作过的每个地方都与管理层保持良好关系。但我当然不是世界上最成功的工程师。如果你做得比我好，你应该给我发邮件告诉我你的建议。

我在这个行业已经工作了十年。我曾在两家大型科技公司任职：在 Zendesk 工作了五年，在 GitHub 工作了五年。这些公司相似之处多于不同之处，因此我对类似的大型、基于 SaaS 的 Web 开发科技公司的工程实践得出了一些结论。不过，你工作的地方可能非常不同，所以我的建议未必适用。

最终，我的大部分职业生涯都在高级职位或更高职位上度过，所以我的建议主要是面向这个层级的。特别是，我没有什么好的面试建议——我一生中只在两家科技公司面试过，而且两次都得到了工作，所以我就停止了面试。虽然我在2010年代曾经站在面试过程的另一方，但此后就没有再参与过。因此，本书中的大部分建议将会是关于如何在高级+层级上取得成功，而不是关于找工作或刚进入行业的建议。

危险的建议

我非常喜欢“锋利的工具”。这些工具足够强大，取决于使用方式，既可能带来巨大的帮助，也可能造成巨大的伤害。大多数形式的直接生产环境访问都属于这一类：比如 ssh 或 kubectl 访问、一个可读写的生产 SQL 控制台。也可以提供“危险的建议”。危险的建议之所以危险，是因为（就像锋利的工具一样）要把它用好需要能力和判断力。把危险的建议给错了人，就像把生产环境的 SQL 访问权限给错了人——他们可能会跑去做出极其破坏性的事情。

这是一本充满危险建议的书。例如，我建议你:

- 做出关于工作内容的决定
- 有时故意违反公司的书面规章制度
- 即使你不确定，也要采取坚定的立场
- 自认是个“骗子”
- 故意避免所有非运输活动

我担心一些读者在选择工作方向时做出错误的决定，或者以错误的方式违反规则，等等。我不喜欢自己可能为伤害某人的职业生涯做出贡献的想法。但我认为写下危险的建议是很重要的 *somewhere*。

大多数职业建议都是假的

主要原因是，强大的工程师渴望危险的建议，就像他们渴望锋利的工具一样。大多数职业建议都是假的：为了避免责任或为了给别人留下深刻印象而写的。如果你认真对待完成任务，你最终会做其中一些事情。

因为它们显然很有帮助。

当你*know*事情并没有按照官方所说的那样运转，却又无人可以倾诉时，这会让人感到深深的疏离。当我还是一名更初级的工程师时，我非常感激那几个愿意和我坦诚相待（私下里）的同事。

经理无法告诉你

另一个原因是，经理几乎从不向你提供危险的建议，即使那正是你需要听到的。如果一位经理告诉你无视公司政策，而你又把事情做错了——例如，你在 Slack 里发帖说你正在这么做，而且你的经理说过这是可以的——那对他们不利。事实上，这对他们的影响远比对你更糟。科技公司的领导层常常把工程师视为“有用的傻瓜”。而管理者则被期望是专业人士。

然而，很多管理者都希望自己能给你这样的建议。当你遵循这些建议时，他们当然会非常感激。我从未当过管理者，但管理那些能力很强的工程师一定令人极其沮丧——如果他们在工作中采取更具战术性的方式（而不是过于拘泥于书面岗位描述），效率会高得多。

危险的建议需要勇气去遵循。它是高风险、高回报的，因此对强大的工程师特别有用（对较弱的工程师则有害）。如果你不觉得舒服去遵循它，你绝对不应该。但如果你已经有时是这样操作，并且在想自己是否在做一些可怕的长期错误，我不这么认为。或者如果你真的是，我也在这么做！我们开始吧。

2010年代之后的软件工程

好时光结束了

在过去十年的大部分时间里，做一名软件工程师一直非常有趣。每家公司都提供大量福利，裁员和解雇几乎闻所未闻，总体而言，我们被当作需要被精心呵护的特殊小天才，好让我们施展魔法般的才能。但在过去两年里，这一切发生了变化。2023 年的第一轮科技行业裁员令人震惊，不过至少各家公司还争先恐后地提供丰厚的遣散费，并由 CEO 发表含泪的公开信，对裁员的必要性表示遗憾。两年之后，Meta 却公然将其裁员定性为“这些都是我们表现最差的人，走了正好”。这到底发生了什么？这对我们意味着什么？

为什么氛围发生了变化？

在2010年代，利率为零或接近零。因此，投资者可以借到一笔 *lot* 的钱。很多钱都花在了科技公司上，希望能获得超额回报。因此，科技公司有动力去（a）疯狂招聘，和（b）做很多低风险高回报的事情，即使最终浪费了钱。科技公司确实 *not* 不需要盈利。事实上，他们甚至不需要赚钱——他们只需要获取用户，或者至少制造炒作，来提高公司自身的估值。在这种环境下，把钱投入到软件工程师身上（通过提供带薪旅行、内部厨师和丰厚的薪酬包）是一个明智的商业决策。

在2023年，这种潜在的经济形势发生了逆转：利率上升至约5%。科技公司激励措施完全发生了变化：现在，盈利变得至关重要，或者至少要赚很多钱。这意味着，对于大多数公司来说，疯狂招聘或继续无限制地投入大量资金并不是明智之举。

他们的软件工程师。

我认为，这本身就足以解释气氛的变化。那COVID呢？它起到了作用，但并不是根本原因。两年（或差不多）的时间里，人们更多待在家里，这意味着与科技产品的互动增多，也意味着更多的钱流入了科技公司。*Everyone* 在COVID期间有招聘。短期的繁荣一旦结束，公司自然会想要裁掉一些工程师，这也是导致初期裁员的原因。然而，我确实认为即便没有COVID，我们也许依然会处于类似现在的状况。公司在2020年之前也一直在不断招聘。

这种认为人工智能正在取代软件工程职位或导致裁员的观点——就我所知——目前纯粹是幻想。我确实相信人工智能的力量，也不会对它在未来某个时刻取代软件工程职位感到惊讶，但它显然不是当前软件工程领域气氛变化的原因。

这对我们意味着什么？

我认为现在很多软件工程师都在固守自己的立场，拒绝改变。在过去十年里，他们的意见一直被咨询用于大公司的决策，现在他们试图保持这种权力。我尊重那些在个人付出代价的情况下，坚持自己认为正确的人的做法。我只是想强调，不跟随趋势变化会有个人代价，特别是对于较初级或更脆弱的工程师。作为一个生活在澳大利亚的人，我自己也感到相当脆弱。

最重要的是要内化的一点是，现在的公司实际上在努力专注。在2015年，许多公司都有同时做所有事情的愿望：建立新的产品线，从产品转向平台，做出重大开源贡献，

致力于打造一流的开发者体验，等等。到2025年，这些举措中的大多数被突然撤资，以便将更多资源投入到公司高管真正关心的少数几个押注上。

在2010年代，似乎公司*were*它们的软件工程师，并且对和工程师一样的事情感兴趣。许多工程师因此被误导，强烈认同自己的雇主。但这只是一个海市蜃楼：部分原因是公司希望吸引和留住人才，部分原因是公司没有真正的压力去对任何事情说“不”。现在，那个海市蜃楼已经消失。公司就是它们的执行领导层，而它们的执行领导层只对一小部分事情感兴趣。

如果你是一个热衷于公司开源库工作的工程师，可能需要正视一个事实：公司其实并没有那么在乎这件事。当利率为零时，做这些事是值得的，因为大多数事情都是值得去做的。但当利率为5%时，大多数开源工作无法达到这个标准。换句话说，你的兴趣现在与公司的利益发生了冲突。

没关系，您的个人兴趣与公司利益相冲突。您有权决定自己关心什么，以及愿意为之奋斗的事物。但是，当您的行为不符合公司利益时，您有可能被视为无效或不可靠。到2025年，这使您面临被裁员的风险。

有没有一线希望?

好消息是，科技公司现在生活在（或者至少更接近）“现实世界”。曾经被宠爱是一件愉快的事，但即便在当时，这其中有一种根本的荒谬感。我认识很多觉得这一点令人反感的工程师，包括我自己。这也是为什么许多

工程师们觉得电视节目*Silicon Valley*看不下去——讽刺过于真实，令人笑不出来。主要是让人尴尬。

如果我必须选择，我一定会选择回到2010年代的就业市场，这样我可以通过更少的工作获得更多的报酬，并且拥有更高的工作保障。我不是傻瓜。但*actually having to ship*的一个积极方面是，你不再生活在幻想中。如果你对事情是现实的，软件工程的工作就变得更容易理解了：

1. 为公司提供价值会得到奖励 2. 不为公司提供价值会受到惩罚 3. “为公司提供价值”意味着推动公司高层的明确计划

这并不算什么使命声明！当然也没有什么“让世界变得更美好”之类的内容。但它有着让人安心的真实感。音乐终于停下来的好处在于，你不再需要担心它什么时候会停。

了解你的薪资来源

随着最近美国联邦解雇潮的掀起，许多人都在指责和嘲笑那些投票支持特朗普的联邦雇员，他们现在才发现自己投票选举的人是将自己解雇的人。你怎么可能有如此糟糕的 *what your job even is* 心理模型？嗯。在我看来，许多软件工程师也在操作一个同样糟糕的心理模型，常常做出等同于投票支持一个承诺解雇他们的人的行为。

我不会引用那些推文，但我经常看到这样的故事：“我说服了我那群白痴老板*finally*让我只做技术债工作，你敢信他们几个月后就把我解雇了吗？”或者“我在这个资金不足的项目上拼命干，结果还是拿到了糟糕的绩效评估”。或者“太不公平了，我还没被升职——看看我做了这么多出色的无障碍/标准/开源工作！”基本结构大致是这样的：

1. 一位眼睛闪亮的工程师加入了一家科技公司，兴奋地想去改变世界，让它变得更好 2. 他们投入到各种不赚钱的工作中（提升FCP性能、改善屏幕阅读器支持、重构代码） 3. 他们的经理们拼命地试图引导他们做些能赚钱的工作，导致了一场漫长而令人沮丧的权力斗争 4. 最终，这位眼睛闪亮的工程师放弃了，不情愿地专注于盈利产品X，或者 5. 这位眼睛闪亮的工程师离开了公司或被解雇，去Twitter上抱怨他们的重要工作没有得到重视

这个故事中的抱怨基本等同于“我居然被特朗普从国税局开除，而我还是投票给那个人”。它代表了对科技公司 *are* 的根本误解。

科技公司是什么?

那么正确的理解是什么呢？我们尽可能简单地开始。科技公司由小团队运营，目标是赚钱。成功的科技公司根据定义会赚很多钱。它们雇佣软件工程师，以便继续做那些赚钱的事情，或做一些能够赚更多钱的新事情。

在成功的科技公司中，工程工作是根据它为公司创造的利润（直接或间接）来衡量价值的。Patrick McKenzie 对此有一篇很棒的文章：

> 利润中心是组织中带来利润的部分：律师事务所的合伙人、企业软件公司的销售、华尔街的“宇宙主宰”等等。成本中心则是，嗯，其他所有人。你真的想要与利润中心挂钩，因为它会带来更高的工资、更多的尊重和更多对你有价值的机会。这并不难：一名聪明的高中生，在得到一段关于企业的描述后，通常能识别出利润中心在哪里。如果你想在那里工作，就为此努力。如果做不到，要么a) 去其他地方工作，要么b) 在加入公司后争取转岗。

公司越接近利润中心，就越重视工作。我不认为你必须要在 *in* 利润中心工作（在大多数科技公司中，这意味着放弃你的工程师头衔和角色）。但你需要向利润中心展示你的价值，这样你的工作才能被重视。我不是说你必须这样做才能保住工作。公司会支付给许多没有创造价值的人，有时甚至是多年。我想说的是：

如果你你的工作与公司利润没有明确联系，你 r posi-

翻译是 ***unstable***

换句话说，你可能依赖于一位善良的经理（或首席执行官），他个人重视你的工作。当他们离开时，你就麻烦了。或者你依赖于一个大公司，这个公司根本不关心是否有一个小团队在带来利润。当他们关注时，你就麻烦了。或者你依赖于一种文化氛围，在这种氛围中，你的工作有短暂的文化影响力（例如，2000年代初的生物燃料）。当这种情况发生变化时，你就麻烦了。拥有稳定职位的唯一方法是与公司赚钱的方式相连接。

将你的工作与利润联系起来

为了了解你的工作是否与公司利润相关，你需要知道两件事：

1. 你们公司的商业模式是什么？他们如何赚钱？ 2. 你的工作如何支持这个商业模式？

上市公司每年必须发布其商业模式和财务状况，这意味着你可以直接阅读这些内容，或者阅读人们在商业博客、杂志等上写的关于它的文章。（如果你在一家知名公司工作，可能只需询问一个LLM）。如果你在一家私人公司工作，可能会更难一些，但通常了解大致情况并不难。例如，Valve 赚钱的主要来源是很清楚的：是 Steam，而不是他们的第一方游戏。

在公司担任工程师将使你更清楚地了解商业模式。例如，你可以运行分析查询，找出前十大客户。通常你不需要直接运行这些查询——它们会在产品和业务团队之间共享，在大多数工程师并不感兴趣的频道中。了解商业模式是值得尝试的。例如，如果我从事工作——

为了Valve，我希望得到比“它是Steam”更清晰的答案：我希望了解哪些类型的游戏带来了最多的收入，新用户和现有用户之间的分布，等等。

一旦你了解了公司如何盈利，你就可以评估你的工作如何支持这一点。如果你开发了一款许多人购买的产品，这很容易：计算你的产品占公司利润的百分比。如果你不开发产品呢？假设你在无障碍团队或德语本地化团队。那么，你应该弄清楚公司为什么会投资这些事情。例如，致力于无障碍可能很有价值，因为：

- 它使我们能够向（例如）视觉障碍客户销售，扩大总可服务客户群体 X%
- 这使我们能够满足特定的监管要求，从而可以向大型企业客户（例如政府）销售。
- 它让我们看起来很好，或者至少避免我们看起来很糟。
- 这只是值得做的好工作，因为这是正确的事情。

有些原因只有在时光顺利时才显得重要。如果公司发展良好，拥有比知道如何使用更多的钱，那么最后两点可能值得花钱去做。（如果公司发展*really*良好，就像2019年很多公司那样，“几乎任何事情”都值得花钱去积累工程师）。当利率上升时，这些原因就会消失。

这些原因对一些公司比其他公司更为适用。例如，如果你在谷歌工作，第一个原因很重要，因为将价值 ~270B 的客户群增长 2% 可以解锁 ~5.5B 的新收入。如果谷歌支付给你的团队的总薪资少于 5B，你的团队可能已经赚取足够的收入来证明其存在的价值。但如果你在一家收入以百万计的小公司工作，那么

数学朝相反方向发展。

显然，这有一个后果：如果你希望你的工作被重视（即你不想被重组或解雇），并且你希望从事个人满意的工作，如可访问性、UI优化，或任何与利润无直接关联的其他事务——你需要去一家非常有利润的公司工作。

提供边际价值

当我写到这个想法——即规模极大的科技公司通过交付边际性的功能来略微扩大其庞大的可服务客户群——时，一些读者觉得这个想法令人沮丧。也许吧！但至少，这为如何在这些类型的功能上工作并因此获得报酬提供了一种理论。另一种理论大概是这样的：

1. 可访问性、清晰的代码、良好的性能等都是好的特性 2. 好公司关注好的特性 3. 我只需要继续寻找，直到找到一个好公司，而不是一个坏公司

我不认为一个聪明的工程师在思考这个问题后会相信这一点。但很多聪明的工程师不喜欢思考他们的工作如何与公司利润连接，因此他们的隐性信念通常会加起来形成这样的观点。这些工程师通常会经历我在本文开头提到的五步过程。我不忍看到技术能力强、积极进取、心地善良的工程师因为完全可以预见的原因而陷入倦怠。

总结

- 人们很容易陷入这样的误区：认为你之所以获得报酬，是因为这份工作很重要。

- 你之所以因工作而获得报酬，是因为它能创造收入。如果你的工作没有为此作出贡献，那么你的职位本质上是不稳定的。
- 如果你想要一个稳定的职位，你应该尝试弄清楚你的工作如何与公司利润关联，并在可能的情况下加强这种关联。

- 各种看似不赚钱的工作其实都能赚钱，尤其是在大型公司里，小百分比也意味着可观的收益。
- 如果你想从事看似不盈利的工作，你可能更适合在大型且成功的科技公司工作

运输

在任何科技公司取得成功的核心是*shipping*。如果你不交付，其他一切都不重要。如果你确实交付了，它能掩盖你可能遇到的许多其他问题。

如何在科技公司运送项目

我在过去的~10年里推出了很多不同的项目。我经常被选中领导新项目，因为在关键时刻需要确保做对，我擅长这一点。在大公司发布项目是一项与写代码完全不同的技能，很多擅长写代码的人在发布项目时却很糟糕。

这是我在领导一个项目时所思考的，以及我看到人们常犯的错误。

运输很难

我看到的最常见错误，是以为发布很容易。项目的默认状态是*not ship*：被无限期推迟、被取消，或者以半成品发布并引发灾难。项目不会在所有代码写完或所有 Jira 工单都关闭之后就自动发布。它们之所以能够发布，是因为有人承担起了发布这一艰难而精细的工作。

这意味着在几乎所有情况下，交付必须放在第一位。你不能把其他任何事情作为你的首要优先级。如果你把所有时间都花在担心打磨客户体验（例如）上，你就无法交付！当你是团队中的一名工程师时，沉迷于 UX 是值得称赞的行为；但如果你在领导项目，这就是一个严重失误。你应该珍惜团队中那些在做这项工作的工程师，并尽你所能给予他们支持。但你的首要关注点必须是把项目交付出去。这项工作太艰难了，根本不可能在业余时间完成。

根据我的经验，项目几乎总是能够交付，是因为某一个人促成了它的交付。需要说明的是，那个人并不会写完所有代码或完成所有工作，我也不是说没有整个团队的支持项目就能交付。关键在于 *really important* 项目中有那么一个人，对整体有端到端的理解：如何

它在技术上是连贯的，并且它服务于什么样的产品或业务目的。优秀的团队和公司理解这一点，并确保每个项目都有一个负责的工程师（通常这个职位被称为“技术负责人”或“DRI”角色）。不优秀的团队和公司没有做到这一点，项目的成败取决于是否有工程师自愿承担这个角色。

什么是运输?

为什么这么多工程师认为交付很容易？我知道这听起来很极端，但我认为许多工程师甚至不理解在一家大型科技公司里“交付”到底是什么。什么叫做交付？它*not*意味着部署代码，甚至也不只是让某个功能对用户可用。交付是公司内部的一种社会性建构。具体来说，这意味着：当你公司里的重要人物相信它已经交付时，一个项目才算交付了。如果你部署了你的系统，但你的经理、VP 或 CEO 对它非常不满意，*you did not ship*。（也许你交付了某个东西，但你并没有交付真正的项目。）只有当公司的领导层承认你已经交付时，你才知道自己真的交付了。来自 VP 在 Slack 上的一句祝贺是一个好兆头，内部博客发布的胜利宣言也是如此。对于较小的交付，经理给你一句表扬就足够了。

这听起来可能有点循环论证，但我认为这是一个非常重要的观点。当然，如果你发布了用户喜爱并且赚了很多钱的东西，那你就算是交付了。但之所以如此，只是因为让用户满意并赚钱这件事会让你的领导团队开心。如果你发布了用户讨厌、也赚不到钱的东西，但你的领导团队很开心，*you still shipped*。你对此可以有任何感受，但事实就是如此。如果你不喜欢这样，你也许应该去为那些真正关心用户有多开心的公司工作。

把“发布”理解为交付规格说明或部署代码的工程师，往往会一次又一次地用工程手段把发布做成失败。

沟通

所以，如果你在交付某个东西时的首要工作是让公司领导对这个项目感到满意，那么这在实践中意味着什么？首先，你必须弄清楚公司希望从这个项目中获得什么。有时候是从一小部分用户身上榨取更多收入（例如企业级功能）。有时候是花钱来扩大整体用户规模（例如吸引眼球的免费层功能）。有时候是通过专门为某个非常大的客户构建功能来安抚他们。有时候这只是某位有影响力的副总裁或 CEO 的“心头好”项目，而你需要与他们的愿景保持一致。可能的原因有很多，如果你想把项目交付出去，就必须知道在这个案例中哪些原因适用。相应地调整你的工作方式和沟通方式！例如，企业级功能通常不需要炫目的 UI，但在需求上完全不能变通；面向终端用户的功能需要打磨得非常精致；而“心头好”项目则意味着你需要与拥有这个项目的那位关键决策者保持积极而频繁的沟通，等等。

第二，无论项目目标是什么，相比你而言，你的领导团队（在你的汇报链路中、关心该项目的人）对该项目几乎始终没有任何技术背景。这意味着他们会在估算、回答技术问题以及预判技术风险方面信任你。维护这种信任应当是你的首要任务。如果他们不相信你有能力把事情做好并持续向他们同步信息，你就无法交付。他们会通过取消项目来“降低风险”，或者让项目在几乎没有关注或庆祝的情况下上线（记住：没有被庆祝的发布不算交付！）。或者，他们会把你边缘化，转而找另一位工程师，而那个人将正式或非正式地成为真正交付该项目的人。

不管怎样，到了评审的时候你都会有所体会，而下一次他们就会转而找别人。

你如何与领导团队保持信任？这个话题本身就可以写成一整篇文章（甚至一本书），但下面是我的总结：

- 最好的情况是，如果你能拿到的话，拥有一份过去成功交付的业绩记录
- 项目自信（如果你看起来担心，他们也会担心）
- 项目能力。你希望追求类似NASA任务控制中心的氛围

- 专业且简明地沟通，不要让他们追着你要更新：在某处发布每日或每周的主题帖。

做这些事情比确保项目按时发布且没有任何漏洞要重要得多。如果一个项目因为技术原因必须延迟，根据我的经验，只要你清晰、自信地沟通（最好能提前一些警告），你不会遭受后果。事实上，恰恰相反，如果有某种问题导致了延迟，对你来说往往是有利的，就像那个修复了突发事件的英雄值班工程师比那个避免事件发生的细心工程师更受赞赏一样。

进入生产阶段

即便如此，你通常仍然必须将项目投入生产环境。这里最常见的问题是遗漏一个关键细节。有时这是一个技术细节：也许我们依赖将用户文档存储在 Memcached 中，但许多文档有数兆字节大小，会超过 Memcached 的块大小。有时这是一个协调方面的细节：也许负责 Memcached 的平台团队原本预计只会收到我们项目发送流量的十分之一，因此他们召集副总裁开会，从而推迟了项目。有时这是一个

法律细节：也许用户数据出乎意料地敏感，而我们的系统没有我们需要的控制措施来安全地处理它。这些问题可能来自任何地方，且很难预见。应对这些问题需要对系统有深入的技术理解，并具备快速调整的能力。

例如，您可能已经阅读了第一个示例，现在正在思考“好吧，您可以将文档分割到多个 Memcached 键中，或者增加块大小，或者迁移到 Redis 等等……”。这些都是潜在的解决方案！但知道哪些解决方案有效——更重要的是，哪些解决方案不会拖延项目时间表——是无法做到的，除非您对此有深入的理解。

这一点尤为重要，因为所讨论的问题甚至不一定是真实存在的。在项目上线前的准备阶段，其他团队或工程师提出 *potential* 问题是非常常见的（例如：“嘿，我们确定用户数据能放进 Memcached 吗？”）。如果没有人站出来解释为什么这不是一个问题（或者如果确实是问题，在上线前是如何被解决的），项目就会被延迟，而责任将落在你身上。为什么？因为你的经理（或他们的经理）并不知道这是否是一个严重的问题。这正是他们付钱让你来做的事情！如果你没有主动站出来处理，他们自然会采取更为谨慎的做法并 *not ship*。

你需要保持行动灵活，这样当这些问题出现时，你就不会被其他工作淹没。这通常意味着不要把全部精力都埋头于实现细节（也就是把任务委派给项目中的其他工程师）。理想情况下，在项目的早期阶段，你至少应当有 20% 的时间不用于实现工作，并在最后几天逐步提升到 90–100%。如果你能做到这一点，那么当问题真正出现时，你就能给予它们充分的关注。

我们现在可以发货吗?

功能标志是我看到的最好的方法，但阶段环境也能发挥作用，等等。关键是将你所构建的东西展示给尽可能多的人：你自己，但也包括其他工程师，理想情况下还包括领导层、产品、设计等。即使是在非常粗糙的状态下，实际体验该功能五分钟，也能发现没人预料到的问题。能够直接看到它本身，也有助于让领导层放心，确保你已经掌控了局面。

预见问题的最佳方式是尽早部署。一般来说，一个有用的问题是：我现在就能发布吗？不是这周，也不是今天：就是此时此刻。如果不能，需要改变什么才能让我发布 *something*？如果发布需要一次部署，这件事能否现在就在功能开关的保护下完成？如果我们在等待其他团队在他们那边做出改动，是否可以让系统其实并不严格依赖他们的改动？例如，如果平台团队正在搭建一个缓存层，我可以让我的功能在找不到缓存时仍然能够工作（只是会稍微慢一些）。

记住，你的首要任务是与领导团队保持信任。没有什么比拥有后备计划更能建立信任，因为在紧急情况下，后备计划表明你对局势的控制。如果最坏的情况发生，且你无法按计划发布，经理会更高兴向他们的上级报告此事，如果他们能说出类似“我们的选择是延迟四天，或者通过牺牲X来明天发布”这样的话 —— 即使牺牲X是无法接受的。这样，他们更可能将延迟解释为一个你有效处理的不可避免的问题，而不是你犯的错误，意味着他们无法依赖你。

我认为很多工程师基本上是出于恐惧而推迟部署。如果你想发布，你需要做完全相反的事情：你需要

尽早部署尽可能多的内容，并且你需要尽早做出最可怕的变更。记住，你对项目拥有最完整的端到端上下文，这意味着你应该是最不怕可怕变更的人。其他人都在处理更多的未知因素，并且会更加不愿意拉动大杠杆。（如果有其他工程师对这一切都很了解，而你在等他们，坏消息是：他们很可能是实际在交付你项目的人）。

总结

- 运输真的很困难，你必须把它作为你的首要任务。
- 交付并不意味着部署代码，而是意味着让你的领导团队满意

- 你需要你的领导团队信任你，才能交付
- 大多数关键的技术工作在于预见问题并制定备用方案

- 在接近上线时，减少实施工作量，这样你就能有时间处理临时出现的问题。
- 你应该不断问自己：“我现在能发货吗？”

- 要有勇气！

胶水作业被认为有害

“Glue work（胶水工作）”是 Tanya Reilly 在 2019 年提出的一个概念。其核心观点是：为了高效运作，每个团队都需要大量不那么光鲜的工作，例如更新文档和路线图、处理技术债务、为新工程师做入职引导、确保人与其他团队的对口人员沟通、留意哪些事项正在被遗漏，等等。务实而朴素的工程师往往会被这类工作吸引，因为它显而易见地有用；但在晋升或发奖金的时候，他们却常常被忽视，转而更青睐那些做了更显眼工作的工程师（比如交付新功能）。

我认为这个概念非常出色。这也是我一直在说“交付项目为什么这么难”的原因——如果你是那种只习惯埋头写代码的工程师，你就不会具备完成真正把事情成功交付所必需的那些“胶水式”工作的工具。纯粹的黑客是不会交付成果的。你需要能够真正应对大型组织中的摩擦，才能交付价值。

那么，既然胶水工作对交付项目如此关键，为什么它却不能让你获得晋升？公司很蠢吗？他们是在故意白白浪费价值吗？不，我不这么认为。公司不奖励胶水工作，因为他们不希望你把它放在优先级上。而他们不希望你优先做它，是因为他们希望你交付功能。胶水工作很难。如果你有能力把胶水工作做好，他们更希望你把这种能力用在交付项目上，而不是提升整体效率。

核心问题在于，你在替公司决定需要什么，而不是在做好你的本职工作。让你的团队运转得更顺畅不是你的工作吗？不！你的工作是执行公司领导层的使命。以 60% 的效率执行这一使命，也比把所有时间都花在提升总体效率上要好（或者更糟的是，以 100% 的效率去执行另一个使命）。

为什么？主要有两个原因：第一，你不可避免地会精疲力竭，这对所有人都不利；第二，与其在短时间内人为地消除摩擦，不如让你的团队习惯在公司的基础效率水平下运作。

你是不是永远都不应该做胶水工作？不是的，你应该做胶水工作 *tactically*。也就是说，对于你所领导的项目——那些你对其成功负有责任的项目——你应该做这种额外的工作，以确保它们取得成功。你不会因为胶水工作本身而获得奖励，但你会因为项目的成功而获得回报。对于其他项目，你只需要做好你的本职工作即可。

这是对如何在办公室政治中取得成功的一种极度犬儒的看法吗？我其实并不这么认为。大型科技公司在任何给定时间的运作效率大约只有 20%–60%（而且规模越大，效率越低）。即便明知如此，增长仍然是一种有意为之的选择：公司之所以扩张，是为了覆盖更多的“表面积”，因为即使在较低效率下，这也是创造更多价值的一种方式。如果个别员工愿意通过把时间消耗在“胶水工作”上，将所在小团队的效率提升到 80% 或 90%，公司会接受这份免费的价值，但它们并没有真正的兴趣把这种状态长期固定下来（因为这依赖于少数杰出的人以难以获得回报的方式自愿投入时间，因此并不可持续）。

如果你是那些出类拔萃的人之一，恭喜你！你可以策略性地运用这种能力，成为一名更高效的工程师。但你不应该一直这样做。

像一名优秀工程师一样思考

到目前为止，我已经写了如何从商业视角来思考你的工作（也就是如何管理项目、你应该选择做什么样的工作，以及如何与公司的领导层保持一致）。但当然，你还必须是一名强大的 *engineer*。什么是强大的工程师?

强大的工程师

在我的经验中，衡量才能的真正标准不是产出的速度或数量，而是完成其他工程师无法完成之任务的能力。换句话说，优秀的工程师能够做到较弱的工程师即使拥有无限的时间也做不到的事情。因此，最强的工程师比人们想象的还要强：不是比中位工程师强 10 倍，甚至也不是 100 倍，而是在某些问题上达到无穷倍。最弱的工程师也比人们想象的还要弱：不是 0.1 倍，而是 0 倍。他们几乎无法完成大型软件组织中所需要完成的任何任务。

例如，能完成复杂项目的工程师和不能完成的工程师之间有明显的分界。这并不是说较弱的工程师做得更慢——他们只是似乎做不到 *at all*。要么是附近的一位强工程师在“幽灵式”地领导项目，要么项目失败。还有一些类似的能力示例：

- 解决非常困难的 bug（例如跨多个服务的竞态条件）
- 在遗留代码库中提供对最棘手部分的有意义改进
- 成功实施需要大规模架构重构的变更

对于最顶尖的工程师，他们的能力变成了诸如“提升大语言模型的SOTA”和“让自动驾驶汽车实现功能”这样的事情。

并不是每个优秀的工程师都能完成所有这些任务。有人可能非常擅长解决困难的 bug，却无法交付项目；或者非常擅长处理遗留代码库，却无法快速推进。然而，如果某人在其中一件事上很出色，他很可能在其他大多数方面也很出色。我并不真正知道原因：也许只是纯粹的智力，或者擅长一件事能帮助你学习其他事情，或者这些能力是

比看起来更相似，或者这些类型的工程师在每件事上都非常努力。但根据我的经验，这绝对是真的。

我想明确一点，虽然能够完成单一任务是相对明确的，但强/常规/弱这三个类别是存在一个光谱上的。可能会有一个“常规工程师”在修复难题时表现出色，或者有一个“弱工程师”在保持开发环境正常运行方面非常擅长。你可能处于两个类别之间的模糊边界。

常规工程师

紧跟在优秀工程师之下的是常规工程师，他们构成了大多数公司的主体。以下是你可以预期这些工程师具备的一些能力：

- 解决95%的漏洞（例如正常的、非诅咒的漏洞）
- 接手并交付大多数 JIRA 工单
- 大多数时候，他们将自己从开发环境问题中解脱出来

很久以前，一位同事曾称这种类型的工程师为“缓步者”：他们的工作速度可能不快，但在一个正常难度的工程任务上，他们会稳步前进。我现在认为“缓步者”是一个不必要的贬义词，因为随着经验的积累，我越来越喜欢这些同事。他们 *help*。他们 *do the work*。他们只是没有强烈的野心去在下一个晋升周期中脱颖而出，或者以令人印象深刻的成果让同龄人刮目相看。可能他们的生活中有其他事情正在发生！

我对这个小组几乎没有什么可说的，除了警告不要把它与最后一个小组混淆：弱工程师。

能力较弱的工程师

另一类是真正薄弱的工程师。这些人几乎没有任何能力。换句话说，一个正常到简单的软件任务的基线难度就已经高于他们所能适应的水平。我猜其中少数人是多重就职或以某种类似方式行骗，但我认为更多的是能力不足。我想明确说明，我并没有在夸大其词：薄弱的工程师 *cannot complete almost any engineering task*。我曾在完全没有这类人的团队中工作过，但只要你在这个行业待得足够久，就一定会遇到他们。

讽刺的是，虽然这样的人几乎存在于所有资历层级，但你更有可能在高级岗位上遇到能力较弱的工程师。我认为这大概有两个原因。第一，招聘初级工程师的门槛明确是以能力为导向的，因此更难混过去。在面试中，高级工程师可以谈论他们只是边缘性参与的工作，这很难与他们 *did* 的工作区分开来。第二，能力较弱的初级工程师往往会获得更多学习机会，因为他们不知道一些事情在社会上是可以被接受的。能力较弱的高级工程师则必须掩饰自己知识的不足并私下学习，这要困难得多。

我对弱小的后辈没有太多要说的。你应该帮助他们，指引他们解决具有挑战性的问题，看他们是否能够迎难而上并学习。然而，弱小的前辈则要有趣得多。

能力较弱的资深工程师是如何生存的?

弱工程师如何在高级+级别生存？他们做很多单向配对。如果你曾和这些工程师配对过，那将是一次非常不愉快的经历，因为你必须做所有的工作，无论是驱动还是导航。配对通常是隐秘的——他们会悄悄地通过私信联系团队中的其他工程师，请求帮助解决每一个任务。有时会有一个不幸的受害者，他们的时间就这样被利用：例如，团队中的一名有效的初级工程师。

谁很乐意提供帮助，但又太缺乏经验，无法知道更好的做法。更精明的弱工程师会在团队中轮流配对，这样每个成员可能每周只需要贡献一次。当每个人交换笔记时，才会发现这个弱工程师正在处理自己 100% 的任务。

弱工程师在与工作相关的讨论中常常出奇地活跃。这部分是因为他们有很多时间——除非他们能找到一个“搭档”，否则他们实际上并没有在工作。这也是一种有效的防御机制。如果有人提出关于他们个人产出的质疑，他们可以拿他们的公开沟通作为证据，证明他们在提升团队的能力，而不是在埋头做具体的工作。在讨论某个问题时，识别一个弱工程师的一个方法是看谁能带来有关系统当前工作方式的具体事实，谁则是提出纯粹的、可以适用于任何系统的通用建议。如果他们的消息可以都作为公开的推文，那他们可能并没有增加太多价值。

与能力较弱的资深工程师合作的一些建议

首先，最重要的一点是记住这只是工作。某人工作做得不好并不意味着他们是坏人。不要做混蛋！人们可能因懒惰和缺乏天赋而变得软弱，但也可能因各种私人原因，而这些与你无关。以你希望他人在你经历个人悲剧、无法专注于工作时给予的慷慨态度来行事。即使是缺乏天赋也可能是特定情境下的表现：例如，也许他们是一个嵌入式系统专家，正在尝试不同的领域，但没有什么成功，或者是一个大公司员工，正在努力适应创业公司文化。

然而，你应该尽量保护你的时间。不要默默地将你的工作时间献给帮助他们维持生计。一种策略是避免时间不对称的帮助。不要为他们做那些需要花费更多时间的工作。

所花的时间比他们询问这件事本身还要多。比如，如果有人问你：“嘿，我遇到这个问题，你会怎么处理？”，不要花时间把实际解决方案完整算出来再交给他们。迅速给出一个 *quick* 的回应，把他们指向下一个最直接的步骤（例如：“哦，对，看起来是计费代码里的问题，你应该看看服务 X 是怎么处理的”）。这样一来，他们就无法用自己几分钟的时间，占用你数小时的精力。

相关地，你也应该努力保护团队中初级成员的时间。不要让他们被能力较弱的资深成员利用——这些人会让初级成员替他们解决问题，然后再向管理层包装成这是在帮助初级成员提升水平。最好的做法是（以专业的方式）确保你的经理了解实际情况。在一些公司，把不合适的人“管理出局”可能是一个极其困难的过程（而且可能存在你并不知情的其他因素），所以不要指望这个人会被解雇、被 PIP，或被明确要求改进。但你仍然有责任确保你的经理对此有所了解。

结论

工程人才并不等同于更快的速度或更高的产出，而是完成其他工程师无法完成任务的能力。这就是为什么最弱的工程师产出如此之低，以及为什么科技公司的 CEO 对招聘最强的工程师格外执着。

大多数工程师都能完成一系列常规的工作任务，但有些人能够完成非常困难的任务，而有些人几乎什么都做不了。你可能应该努力扩大自己熟练掌握的任务范围。如果你正在与一名能力较弱的工程师合作，要保持友善，但也要保护好自己的时间。

是什么让优秀的工程师变得强大?

正如我上面所说，强工程师的定义在于能够完成弱工程师即便拥有近乎无限的时间也做不到的任务。但构成这种能力的具体技能或特质是什么呢？强工程师身上究竟有什么，使他们能够完成范围广得多的任务？按重要性排序，我认为依次是自信、务实、速度，以及技术能力。

自信

强大的工程师相信自己能够成功，即使面对困难或不熟悉的问题。许多技术能力强的人却是弱工程师，仅仅因为他们缺乏自信。为什么？因为软件问题都是不熟悉的问题。软件工程师在黑暗中工作。几乎每一项实际工作，在完成之前都无法预见它有多么棘手。这就是为什么技术公司中的估算工作非常困难的原因：你无法预知会遇到什么问题。

有许多聪明的人无法在这种环境中工作。谈论项目时，如果他们不确定技术细节，或者在很多细节尚不清楚的情况下开始工作，这会违反他们的核心工程自我认知。要成为一名强大的工程师，你需要拥有足够的自信，相信无论遇到什么问题，你都能解决它（或者如果问题真的太难，那它一定是如此棘手，以至于无法解决也不丢人）。

这不仅仅影响工程师从事的项目，还影响工作的质量。如果你接手一个困难的任务，直接正面解决最难的部分和推迟处理、试图绕过它，区别非常大。我曾看到工程师们花费数周时间回避一个本可以在一天集中努力下完成的困难任务。

它还创造了一个正反馈循环。具有高度自信的工程师倾向于解决更具挑战性的问题，这增强了他们的自信心，进而推动他们去面对更困难的问题，依此类推。描述这种类型的工程师的一个常见方式是“主动型”。正如常说的那样，“你可以做到事情”。

实用主义

强大的工程师能够完成任务。他们倾向于选择可行的解决方案。根据我的经验，强大的工程师都是无情的实用主义者：每一个设计决策都是通过其效果来判断，而不是看它有多简洁或优雅。这并不是说强大的工程师不会创造优雅的解决方案。优雅的解决方案往往是最直接的。但是，强大的工程师会抵制为了整洁而添加抽象层次。他们通常愿意为了按时交付做出妥协。

因为这个原因，强大的工程师常常与聪明、技术能力强但较弱的工程师发生冲突。典型的导火索可能是“我们是否需要在发布前两周进行这次重构”或“这个设计模式是否对我们的用例来说过度”。作为外部观察者，很难判断这些冲突，因为双方在技术上都很强。如果你处于这个位置，我建议在决策时根据哪一方在交付方面有更好的记录来打破僵局。

速度

强大的工程师工作迅速。我从未见过一个不是高效的工程师。我认为这有两个原因。首先，许多工程师工作慢是因为他们拖延艰难的工作。强大的工程师不会这样做（参见上文关于自信的观点）。其次，快速工作的人通常随着时间的推移成为强大的工程师，因为他们迅速积累经验。我本来会写更多内容，但丹·卢

已经在这里说得最好。我全力支持那篇文章，特别是以下几点：

- 快速的执行速度让你能够足够迅速地进行实验，从而找到正确的解决方案
- 快速执行使得低概率但高回报的想法值得尝试，最终产生高回报

- 快速执行在质的层面上不同于慢速执行。完全不同的任务因此成为可能

我不认为优秀的工程师必须是特别勤奋的人，这里的勤奋指的是长时间工作。根据我的经验，好的工作状态更像是短暂而高强度的生产力爆发，中间夹杂着较长时间的相对低效。当然，可能确实存在一些绝对的“怪物级”人物，既极度高产又工作时间很长（据说约翰·卡马克就是其中之一）。但我不认为要在你的科技公司里成为一名非常出色的工程师，就必须做到那样。

技术能力

确实，成为一名优秀工程师需要具备一定的技术基础，而更强的技术能力始终是一个优势。深入的技术人才能够看到其他人忽视的简单解决方案，并能够解决其他人无法解决的整类问题。

一名优秀的工程师需要具备多强的技术能力？这实际上是一个关于你的具体技术优势与需要完成的工作之间匹配程度的问题。例如，我非常擅长在庞大的 Rails 代码库中进行代码追踪。这让我很适合做调试以及构建业务线功能，但在最近的一项工作中，当我需要给一个内部的 C 库添加一些功能时，这些优势几乎没有派上用场。总体而言，从事一个全新工程领域的公司将比一家公司需要更多的技术技能

应用已建立的工程实践。

优秀的工程师不需要是天才。事实上，天才往往与成为一名优秀工程师所需的技能背道而驰。我合作过的一些最聪明的人——就纯粹的智力而言——并不是特别高效的工程师，因为他们在务实性和速度方面存在困难。我更愿意与一位智力水平中等、但异常自信且务实的工程师共事。

总结

- 强大的工程师相信自己能够解决几乎所有问题。他们直接迎接最可怕的未知，而不是拖延。
- 强大的工程师是务实的，这常常让他们与聪明但较弱的工程师发生争执。
- 强大的工程师总是高效的工作者，但不一定是苦干者
- 强大的工程师在技术上很强，但通常是他们的优势与工作非常契合，而不是“这个人是天才”。

替代价值

有两种评估你作为工程师提供了多少价值的方法。第一种方法是将你所发布的所有代码以及这些代码所提供的所有价值（例如，它所带来的收入）加总起来。第二种方法是尝试弄清楚*you specifically*做了哪些事情，而一个替代水平的工程师在你的岗位上是做不到的。换句话说，你可以看你的实际价值，也可以看你的替代价值。

我认为一种方式不一定比另一种更好。但重要的是要清楚自己正在使用哪种方法以及为什么使用它。例如，如果你在想自己是否在为自己的薪酬找理由，你应该关注价值。这是一个直接的计算，涉及公司为你花了多少钱，以及它从你身上获得了多少价值。但如果你在想自己是否值得晋升，你应该关注替代价值。你开发的某些代码带来了1000万的年经常性收入，这很酷，但如果那是任何同行都会做的正常工作（例如，如果你生病了），这不一定意味着你表现超出了自己的水平。

如何衡量价值？知道公司花费了多少钱在你身上是很容易的。困难的部分是衡量公司从你身上得到多少回报。如果你知道公司赚取了多少利润——而你应该知道——你可以尝试估算公司中你所在部分产生的利润，以及你的代码为此做出了多少贡献。例如，如果你为一个新的客户群体编写了注册流程，而这些客户最终成为了每年 ~5百万美元 的收入来源，你可以合理地说你的价值是这 5百万美元/年的某个合理百分比。

替代价值如何？那需要猜测你做的哪些事情是其他工程师不会做的。

在你的职位上（或者做得更慢）。你必须发展一个“替代级工程师”的概念：在你的水平和公司中可以雇佣的中等水平工程师。在这里，很容易自欺欺人——要么认为你的替代者会完全无能，要么认为你的替代者会非常强。一般来说，这比仅仅衡量价值要困难得多。

如果你习惯于主动识别新工作（例如：开发新功能或深入分析指标以发现性能问题），估算你的超越替代价值就会容易得多。替代水平的工程师至少会尝试完成你分配的所有工作，但他们不会有你拥有的相同想法。只要你识别的工作确实增加了价值，那就是超越替代的价值。

然而，即使只是做常规工作，你也完全可以创造出超越替代者的价值，哪怕这种价值更难被察觉。识别这一点的一种方法是，留意你（或其他工程师）是否被明确点名参与某个特定项目：这可能是因为你的领域专长、异常的效率，或类似的原因。这就是对你超越替代者价值的直接认可。

理论上，你不一定非得在替代价值上做出贡献。作为一个在所有指标上都处于标准水平的工程师，按定义来说已经足够了。但没有人能在所有指标上都处于标准水平。每个人在某些任务上表现较差，在其他任务上表现较好。实际上，你最好至少在某些领域提供正的替代价值，因为你很可能在其他领域提供负的替代价值。

强大的工程师往往是对的

亚马逊有一条臭名昭著的领导力原则，他们说“优秀的领导者经常是对的”。我不太清楚这条原则对领导者有多大用处，但对工程师来说，它绝对成立。优秀的工程师确实经常是对的。

布莱恩·坎特里尔在这里对亚马逊原则有一番很好的吐槽。我同意布莱恩的看法，觉得这个原则听起来有点傻，而且它不是那种能够指导艰难决策的根本原则（没问题，只要做出正确选择就行，像一个好领导者那样！）但我确实认为这只是一个简单的事实。好领导者*are*做对的事情的次数很多。当然，这源自其他的品质：他们很聪明，或者他们让自己身边有好人，或者出于其他一百种原因。我认为亚马逊原则的核心在于，评估这些其他品质的总和是很困难的。与其这样，不如问自己：这个人做对的事情的次数多吗？

好的，这更容易了，但仍然不容易。你可以根据用户是否蜂拥购买他们的产品、是否赚了大笔钱、他们的公司是否被认为是令人惊叹的工作场所等等，来判断一个领导者 *wins* 是否成功。然而，人们赢得成功有很多原因。你也可以通过在某个特别关键的事情上既错误又幸运来获胜。糟糕的领导者有时也能偶然获得惊人的产品。商业世界实在是太混乱，难以准确评估。

在工程领域并非如此。优秀的工程师通常是对的，很多时候。他们对软件系统如何运作的具体事实陈述是正确的（例如，端点X的速率限制是Y）。他们对具体的计划是正确的（例如，我们不能在这里添加这个检查，但在那里可以）。他们对编写代码时做出的成千上万的小决策是正确的，这使得他们的代码通常能正常工作，并且导致更少的错误。

你可以根据工程师在交谈中听起来有多聪明、他们用的是浅色模式的 VSCode 还是深色模式的 vim、他们的头衔等等来评估一个工程师。但如果你有时间，更可靠的方法是看他们是否经常是对的。对于技术同事，这种评估会自然而然地发生：你只需问问自己，当他们提出主张时，你是默认感到放松还是心存怀疑，或者在评审他们的 PR 时你本能地会带着多强的批判眼光。否则，你也可以留意他们自信地说了什么，以及事实证明是否正确。如果你是一名工程师，很有可能你的经理正在悄悄地跟踪这一点。

一个警告：一些工程师通过从不做出自信的技术陈述来避免犯错。我认为这是失职。如果你是房间里最具技术能力的人，那么提供信息和建议是你的责任——这是你的工作。总是说“嗯，我不确定”或者给所有的陈述加上限制词会让那些技术能力较弱的人很难与你合作。无论如何，原则是“要做得*right*很多”，而不是“永远不犯错”。如果你不主动提出自己的观点，就不可能经常是对的。

为了延续这一点，“对的，很多”与“总是对”之间，甚至“经常对”之间是有区别的。在那些技术上非常不明确的领域（例如腐化的遗留代码或前沿技术问题），即使只有一部分时间是对的，也极其有价值。根据我的经验，犯错是可以被原谅的，特别是如果你是唯一一个敢于提出答案或计划的人。

所有这些对那些想要正确地 *become*（很多次）的工程师来说并没有什么帮助。这是一个更复杂的话题。但它是衡量自己表现的一个很好的标准：

1. 你是否在自我推销并做出技术性陈述（无论是用英语还是代码）？ 2. 这些陈述是否正确，很多?

优秀的工程师会采取立场

一些工程师认为在技术讨论中保持不表态是一种美德。我们的团队应该以事件驱动方式还是同步方式来构建一个新功能？嗯，这取决于：每一方都有许多强有力的技术理由，所以最好保持开放的心态，不偏向任何一方。当你是初级工程师时，这种策略是可以的，但到了某个时刻，你将是房间里最有背景（或技术能力，或机构权力）的人。到那时，你需要采取一个立场，无论你是否特别自信。

如果你不这样做，你是在迫使那些比你技术背景更薄弱的人自己去弄明白。通常这意味着有人会做出随机的猜测。在最坏的情况下，团队中最弱但最吵的工程师会借机推动一个非常糟糕的想法。如果你是一个强大的工程师，你有责任采取立场，以防止这种情况发生，即使你只有 55% 或 60% 的信心。

为什么保持不表态是懦弱的

像大多数形式的怯懦一样，保持不表态从内心感受上看起来像是明智的谨慎。毕竟，技术问题很复杂。总是有理由表达不确定性，或在陈述中加上保留。如果正确的前进方向确实不清楚，那么（他们说）表达不确定性在严格意义上是正确的。

我认为，促使这种态度的原因通常是许多工程师（包括我自己）非常、非常、病态地讨厌犯错。当我在某件事上犯错，尤其是在公众面前时，我会感到胸口一阵恶心。我会在事后长时间思考这个问题。这是有用的，因为它促使我付出努力去做对的事情。但它也使得在会议中做出一个可能错误的有根据的猜测 emotionally 困难。

最终完全错误。我已经努力去接受这一点，所以我能理解那些做不到的人。但我也看得很清楚：这是一种懦弱。当人们依赖你做决定时，你应该挺身而出并做出决定。

如果你错了怎么办?

当工程师过度使用警告和限定语时，管理者通常不会想到“哇，我很高兴这个人如此小心和准确”。他们会想“唉，为什么你要逼我自己做决定？”

根据我的经验，当你做出一个技术判断而结果证明是错误的时，管理者通常会非常宽容。这是因为他们的工作本身也涉及大量基于经验的猜测，所以他们早已内化了这样一个事实：有些猜测并不会奏效。当你所做的判断确实非常困难时，这种宽容会加倍——例如，在会议中突然出现一个技术问题，所有人都陷入沉默。如果只有你一个人站出来回答，即使答案是错的，也仍然可能很有价值。走错方向至少常常能为你带来信息，或者提供一个可以继续迭代的基础。

当然，如果你犯错太多，人们就不会再信任你的估计了。或者，如果你在某个特定情况下错得太离谱——例如，你提供的解决方案导致了一个更严重的事件——你也会失去信誉。我建议通过经常正确来避免这种情况。

有时候避免承诺是聪明的

估算是一个有趣的例子。许多工程师对“嗯，这取决于，难以说，可能几天，也可能需要一个月”这种回答是默认的，除非是最明显的一行代码更改。但你的经理并不是出于好奇问的，他们问是因为他们

需要一个大致的估算用于规划。如果你给出一个不明确的回答，他们只会在心里叹息，然后自己猜测估算值。

然而，有时避免估算并非出于懦弱。在一些公司，工程师避免做出确切的估算，因为当这些估算未能实现时，他们将面临真实且不公平的后果。在这种情况下，工程与产品之间的信任已经完全破裂。工程师被激励低调行事，永远不做任何承诺（至少在管理层面前）。

我确信有些公司环境中，每一个技术承诺都如此风险重重。我对这些环境中的工程师没有任何批评。

总结

我想以重复我自己的一个警告来结束。我并不是说你应该强迫自己做出承诺 *when you're the person in the room best positioned to know the answer*。当你和一个技术同行——例如你团队中的另一个工程师——交流时，凭借你的上下文水平，你可以不做任何承诺。尽管如此：

- 如果你不表明立场，你实际上是在默许最终做出的决定。
- 在极端情况下，这迫使你的经理做出艰难的技术决策，这些决策是 *your responsibility*
- 决策越困难，你应该愿意接受的 uncertainty 越多。
- 我只是在谈论功能性环境。如果你的经理因为错过了估算而对你进行绩效改进计划，那真糟糕——对于那些在这种情况下保持沉默的人，我没有任何批评。
- 提出一个你不确定的主张确实可能让人害怕。但你仍然应该去做。

优秀的工程师了解聚光灯

想象一家科技公司是一座巨大的、昏暗的工厂。工厂里各个部分的工作不断进行，组件来回传送，成品不断被搬走，但各个部分的协调性和效率相当低。工厂的某些部分会发生堵塞，几天什么也不做。其他部分则疯狂生产出损坏的部件，这些部件最终被丢弃。 然而，工厂经理——即科技公司 CEO——有一个工具来应对这个问题。那个工具就是聚光灯。

有一个巨大的聚光灯安装在屋顶梁上，像蝙蝠信号一样，随时指向工厂的某个部分。当工厂的某个部分被照亮时，它的运作就会更加符合工厂经理的意图。员工们工作更加努力且高效。但聚光灯一次只能照亮一个区域。当一个区域被照亮时，其他区域则处于黑暗中。因此，巧妙地使用聚光灯是厂长工作中最重要的一部分。保持聚光灯指向最关键的工作（同时避免任何部分长时间处于黑暗中）是一项微妙的平衡艺术。

科技公司关注有限

聚光灯代表公司当前正在积极关注的事情。科技公司一次只能专注一到两件事，因为“专注”意味着“CEO 正在亲自监督这件事”。其他一切则只是勉强推进。在运作良好的公司中，聚光灯决定了高度专注的工作与常规但仍然成功的工作之间的差别。在运作失灵的公司中，聚光灯决定了事情是否会真正发生——有的工作会发生，而有的工作则完全不会发生（换句话说，只有在聚光灯下，工作 *only* 才会发生）。具体来说，处在聚光灯下意味着：

- 首席执行官正在要求每日更新，这带来了一种 rea l 感觉

紧急程度在组织结构图中的下移

- 阻碍者将在必要时被强制移除（例如，如果Y队拖延，预计他们的上级的上级的上级会下来处理此事）

- 敏捷流程（估算、规划会议、回顾）常常被简化，甚至被完全移除

对软件工程师来说，处在聚光灯下意味着高压力与高回报。你的直属经理和跳级经理，只有在你的团队处于聚光灯下时，才会给予你前所未有的关注。在聚光灯下完成的项目——前提是你在其中的出色表现是可见的——在你的晋升材料中，其影响力会放大五到十倍。如果你从未站在聚光灯下，就很难获得晋升所需的那种曝光度（尤其是晋升到高级或资深岗位）。

在聚光灯下工作时优化是一个非常好的想法：例如，将你的基础工作量设定为80%或90%的努力，这样当聚光灯照在你身上时，你可以在短时间内将工作量提升到110%或120%。这不仅有助于你展示自己的能力，还能帮助你避免犯下严重错误。在聚光灯下出错可能会永久损害你在公司中的声誉。

追逐聚光灯

一些工程师将他们的职业生涯奉献给追逐聚光灯。这可以是一个良性循环：如果你在聚光灯下脱颖而出，你很可能会被转移到聚光灯指向的下一个领域。实际上，你可以通过成为公司领导用来聚焦特定领域的工具之一，成为聚光灯的一部分。当高级领导层认为“我们需要做好这个项目”时，他们也会想到“我们应该把工程师X调到这个项目，他们可靠”。显然，这是一个获得晋升的好方式。

然而，追逐聚光灯是令人疲惫的。它不仅需要

工作更辛苦了，但频繁的调岗和项目切换本身就很难适应。你往往需要与许多不同的人合作，而不是长期待在同一个团队里，这使得更难建立那种能让工作愉快得多的长期同事关系。坦率地说，作为一名资深+工程师，你的薪酬确实不错，但肯定无法与你所支持其优先事项、并为其确保奖金的 C 级高管和高级副总裁（SVPs）处在同一水平！全职选择接受 CEO 的工作节奏未必是一笔好交易。

避开聚光灯

其他工程师将他们的职业生涯藏在聚光灯之外。有时，这是因为他们不称职，不想承受压力，但也有很多技术熟练的工程师更喜欢做离高层越远越好的技术工作。这些工程师通常从编写代码本身中获得很多内在动力（而不是为股东创造价值）。他们通常也不具备雄心壮志——既不是积极的雄心，想要争取更多的权力和影响力，也不是消极的雄心，害怕被边缘化或被解雇。

这没什么错，但这确实意味着你可能会因为你的努力而得不到应得的回报。你的辛勤工作可能不会得到足够的回报。一个月在聚光灯下的努力，相当于在黑暗中努力一整年。这是有充分理由的！公司会奖励那些为实现公司目标而努力的工程师。如果你没有在公司最优先的事务上工作，公司（可能）会给你更少的回报。如果你真的喜欢在聚光灯之外工作，这可能是值得做的交易——但你不应该抱怨其后果。

总结

- 科技公司一次只能专注于一两件事情；这就是“聚光灯”
- 在聚光灯下表现出色的工程师会获得更多的关注、奖励和职业成长。
- 一些工程师追求聚光灯，成为领导层熟知的量化人物，但以牺牲稳定性为代价。
- 有些人避开聚光灯，默默地做着良好的技术工作，但往往得不到应有的认可。
- 任何一种策略都可以奏效，但你应该诚实地面对其中的权衡。

强大的工程师受到管理链的信任

从事关键任务既有趣又有成就感。但重要的工作数量终究有限。更糟糕的是，参与这些项目的机会分配并不均衡，因为这些工作往往由被精心挑选的团队来完成。我也参与过不少这样的团队。是什么让一名工程师被列入那份名单？

信任圈

随着你在公司管理层中的认可度不断提高，你会沿着一系列同心圆向上提升。中心位置是在你自己团队中成为首选工程师。你从这里开始，只需把本职工作踏踏实实地做好。在这个阶段，你的经理会（明确或隐含地）把重要的工作引导并交到你手中。

接下来是成为你所在组织的首选工程师。此时，你经理的同事们正在要求你的经理让你负责关键工作。你可能会进入跨团队项目的候选名单，领导任何涉及你团队的项目，或者至少会被战术性地安排参与必须成功的跨团队项目。你所在组织的领导（通常是总监或副总裁）知道你的名字，偶尔会直接联系你。

最终，你是公司内圈的工程师之一。所有与工程相关的董事和副总裁都知道你的名字，你经常被召集到跨部门的临时团队中，执行关键任务。你被视为关键资源，因此通常不能在某个单一项目上待得太久——公司会根据当时的优先事项，将你调派到任何它认为最重要的地方。

这些同心圆是*trust*的圈层：首先获得你自己的经理的信任，其次获得你经理的同级同事和直接上级的信任，最后

从公司里真正的关键人物，一直到 CEO。除非你已经和副总裁是一起滑雪的老友，否则你必须按部就班地逐级通过。你老板的老板之所以愿意考虑信任你，只是因为你的老板信任你，依此类推。

你是如何在这些圈子中逐步上升的？以我的经验来看，是在每一步都被邀请。不同于在你自己的团队里，你不能主动去要求被安排到某个特定的工作上。相反，你会通过参与重要项目而获得与高级管理者接触的机会，而这（如果你做得很好）会促使这些管理者把你点名安排到更重要的项目上，从而让你接触到更高层的管理者，如此循环。你对这一切几乎没有直接的控制权——你能做的只是让自己被看见，并努力成为领导层愿意安排到其他项目上的那种工程师。

所以问题“我如何获得好的机会”归根结底就是这个：成为公司可以信任的那种工程师需要什么？

信任为何如此难以建立

很少有工程师真正意识到自己与最高层管理者之间的关系究竟有多么微妙。随着你与管理链条中更高层的人交谈，他们拥有的技术背景越来越少，也没有直接影响产品的权力。但他们拥有大量的制度性权力，可以要求工程师去做事，而且还有大量的 *respon- sibility*：如果你参与的产品没有成功，对你来说只是有些不便；但这会让你的领导团队在奖金损失上付出真金白银（甚至可能因此被解雇）。因此，他们对成功投入甚深，但几乎完全依赖工程师去实际完成需要完成的工作，并告诉他们在可用的时间内能够做到什么。

这就是为什么工程师与领导之间的关系既奇怪又微妙。

去维持。这些是公司中非常有权势的人，他们可以动员数百甚至数千人来实现他们的目标，但他们自己*need your help*直接做任何事情。对他们来说，这是一个尴尬的处境。即使是凯撒坐在牙医的椅子上也会感到无助。如果你能在这种情况下真正提供帮助，你将从拥有凯撒作为盟友中获益。但当然，如果你背叛了他的信任，你可能会被砍掉脑袋。

大型科技公司之所以有许多管理层级，其中一个原因是为了避免不得不依赖这种棘手的关系。副总裁（VP）理应能够与向其直接汇报的经理沟通，这些经理再与向他们直接汇报的经理沟通，而这些经理再从工程师那里获取估算和计划。于是，信息在上下传递的过程中被“安全地”经手了三遍。现实中，非常资深的领导层往往对这种间接性心生不满，转而私下依赖少数资深工程师作为讨论对象或理性校验。有时这种做法甚至会在组织结构图中被正式化（例如，设立一名“浮动”的资深员工工程师，直接向 VP 或 CEO 汇报）。

维护信任

如何才能赢得并保持非常资深管理者的信任？首先，要谨慎守口如瓶。如果你老板的老板的老板私下向你提了一个问题，而你却立刻在 Slack 里说诸如“就像我前几天和大老板谈到的那样”之类的话，那你已经证明了自己管不住嘴。获得高层领导的信任，往往意味着被托付机密信息（例如公司的计划）。要避免给人留下你不懂得如何保密的印象。

第二，要站在他们的层面交流。你和你的 CTO 几乎处在完全不同的语境中：不同的工作领域、不同的问题、不同的工具。当他们来找你并询问某个特定功能（例如）运行得如何时，这个问题对你来说简直就像是在用一门外语。

在你没有他们具体语境的情况下使用语言。如果你一开始就立即就 SLO 或代码质量展开回复，你会让他们感到无聊，且无法提供任何价值。他们几乎肯定是带着某个具体议程来的，或者听到了某个想要核实的说法。在你承诺给出回应之前，你需要先引导他们表达出这个议程或说法的大致内容。

最后，你必须把技术问题弄对。如果你告诉他们某件事是以某种方式运作的但事实并非如此，或者说做 X 不可能而结果却是可行的，信任就会消失。非常资深的管理者与具体工作有足够的距离，以至于他们无法在细节上对你进行复核，但正因为如此，他们会密切关注那些他们*can*能检查的事项，比如项目面向客户的结果。

如果你真的破坏了信任，会发生什么？与凯撒不同，你那位“友好的邻里高管”并不会在你把互动搞砸时真的砍掉你的脑袋。但你会悄无声息且立刻被移出那个信任圈。他们将不再直接向你提问，不再在秘密项目的早期规划讨论中把你拉进来，也不会在私下讨论由谁来领导哪个项目时提到你的名字。这并不是世界末日，甚至也不意味着你在公司里失去了晋升的可能。但这确实意味着你与那位特定高管的密切关系就此结束。

将其纳入你的管理链

提醒一句。在公司里被管理者信任是件好事。但你必须尽量把这种信任限制在你的管理链条之内。我见过工程师陷入一个陷阱：成了来自不同组织的经理或产品人员的首选联系人，却以牺牲本职工作为代价。你所在组织之外并不缺少掠夺型的经理，他们非常乐意“免费”占用更多工程师的时间（即，

他们不必与实际需要对其负责的人讨价还价）。一个“快速的请求”可以是一个明智的主意。两个或三个则是一个错误。

感到自己有帮助是令人愉快的，对于许多工程师来说，解决他人的问题的机会足够具有诱惑力。但像这样参与其中真的不是个好主意。这些人会非常有说服力——这是他们的工作！——但在你下一次晋升讨论时，他们不会在场，而且一旦对他们有利，他们会立刻离开你。把精力留给你所在组织真正付薪水让你做的工作。

如何应对这些请求？直接涉及到你实际的管理链。可以说类似这样的话：“嘿，我很乐意帮忙，让我把这个请求跟我的经理沟通一下，让她知道我在做什么。”你的经理可能会因为她的工程师被挖走而大发雷霆。

总结

- 信任的基础是擅长你的工作。做一个擅长你工作的专业人士！
- 你通过参与重要项目而被注意到，这会让更资深的人注意到你，依此类推
- 你试图建立信任的管理者越资深，这段关系就越微妙。
- 这是与直接经理建立信任所需的技能集不同。
- 直接信任关系非常有用，但违反了“正常”的指挥链，因此必须保持相对低调。
- 不要疏远你自己的直属经理
- 不要投资于自己管理链之外的这些关系！也许有可能，但我从未见过它有效。

我如何决定要做什么

在科技行业中，最重要的职业技能之一是学会识别哪些工作才真正重要。许多工程师整个职业生涯都没有真正做出这个判断。他们也许会在冲刺规划中偶尔就某个问题发声，但并不会在心中维护一份团队中正在进行的最重要工作的清单。把这个决定推给你的经理很容易。毕竟，决定哪些项目重要并不真的是你的工作——你的工作是执行项目，并就技术层面的事情发声。

这是一个巨大的错误！大多数被优先处理的工作其实是低优先级的。在科技公司，工作并不像流水线那样，每个小时都同等重要。它更像当一名消防员：长时间相当清闲的工作，间或被短暂而紧张的阶段打断，在这些时候把事情做对至关重要。

为什么重要的工作很稀缺

这份工作的本质就是如此，因为科技公司在任何一个时间点只能专注于一两件事情。“专注”意味着“CEO 正在亲自跟进这件事”，而一个人能够跟进的项目数量是有限的。当聚光灯照在你们团队的某个项目上时，你最好愿意放下不那么关键的工作，全身心投入到这个项目中。但聚光灯一次只能照亮一件事——最终，它会移开。

对哪些工作重要有良好判断力的工程师会识别这一过程的早期迹象，并确保自己处在可以随时介入的位置。那些没有这种判断力的工程师在聚光灯打来时会正忙于其他工作，因此无法立即转向高优先级项目。这很糟糕：

- 对工程师而言，因为他们失去了从事那些在公司最高层可见的工作的机会。

- 对于团队来说，因为这意味着团队在重要项目上的资源较少。
- 对公司而言，因为它将无法在其所专注的项目上执行得同样出色。

做错了事情

在最坏的情况下，工程师们不会意识到重要的工作正在进行，并且在全员待命的情况下，明显地做着其他事情。例如，他们会在 Slack 上发布关于某个无关任务的问题或更新。这比什么都不做还要糟糕。经理们非常讨厌这样，因为这向所有在场的人展示了他们未能集中团队的注意力。这真的是一个非常糟糕的形象。

跳级经理及以上的管理者会关注在某个团队被聚光灯照着的时候，哪些工程师在工作（或至少看起来在工作）。但他们也会注意到那些在那段时间犯下明显错误的工程师，或者看起来在做完全不同工作的工程师。你可以用一整年稳步建立起声誉，却在一周内因为显得毫无用处而失去一切。或者，更乐观地说，你也可以通过在恰当的一周里表现得极其出色，弥补一整年的低迷。

当聚光灯不在你身上时

在你不参与高可见度项目的时段，我建议划出属于你自己的实验室时间或20%时间。（也许从10%时间开始，逐渐增加。）这是积累快速、简洁成果的好方法，这些成果可以放入晋升资料包或简历中。但这也是一个实践“投入其中”的方式。如果你花一个下午在自己设定的目标上，而没有带来价值，那么那段浪费的时间就是*on you*。就像你花了一下午玩电子游戏一样。一旦你意识到这一点，它迫使你变得更有行动力，这对你的职业生涯有各种好处。

一个项目怎么会不提供价值？例如：

- 低价值的重构：工程师毫无察觉，只是满足你对整洁的追求。
- 永远不会有任何进展的功能演示（例如，因为它与公司战略不一致）。
- 那些引入新缺陷的缺陷修复，因此在投入的工作量上净为零。
- 在不关心性能的端点上的性能改进（管理员面板，低频率 API）。

这些是不会带来价值的项目的例子。但*reason*一个项目不带来价值的原因在于，它既不会(a)为你在经理面前建立信誉，也不会(b)为你在同事工程师面前建立信誉。

实验日的价值

作为一名初级工程师，我做过的最有用的事情之一就是参与Zendesk每周的“实验日”。这是我们版的Google“20%时间”计划——这个想法是你每周花一天时间做自己选择的事情。唯一的规则是必须至少与工作相关：比如一个新功能的演示，或者一个不属于冲刺任务的代码清理，等等。

我不记得在实验室时期我交付了什么。可能是一系列小的性能改进，因为那时我对这些最感兴趣。但真正的价值在于定期练习选择我工作的技能。这确实是一项技能：一开始你做得很差。需要时间去学习哪些任务会产生实际影响，哪些则不会。

总结

- 选择工作内容是一项大多数工程师不常练习的技能

- 大多数被优先处理的工作其实是低优先级的，因此真正的影响来自于知道何时以及在何处集中精力
- 科技公司以一种“聚光灯”模式运作，在这种模式下，领导层一次只能专注于一两件事
- 当聚光灯照在你的团队身上时，你需要做好准备，因为高影响力的工作往往发生在短暂而关键的时期
- 在错误的时间做错误的事情比什么都不做更糟，因为这表明缺乏觉察，并可能损害你的声誉

- 当聚光灯不在你身上时，主动创造机会能帮助你保持投入并建立信誉
- 并非所有副项目都有价值。好的项目要么能在管理者那里建立信誉，要么能在工程师同行中建立信誉

不要做一个 JIRA 工单机器人

别做一个 JIRA 工单僵尸！我觉得很多有抱负的初级工程师都会有这样的经历——我自己也曾经这样——对团队推进缓慢感到沮丧，于是决定“算了，我就把这些工单全都刷完”。在很多团队里，要做到比下一个最高产的人多完成 2 倍甚至 3 倍的工单并不难。但这是一条死路。你最多只会被摸摸头，听到一句“干得不错，别把自己累坏了”，却对晋升高级毫无帮助。

你从一家公司能得到的一切——晋升、奖金、内部认可——都来自于交付*projects*，而不是把工单关掉。经历一段大量处理工单的阶段仍然是有用的：了解自己的工作速度，快速熟悉代码库，并在同事中积累一些信誉。但在某个时刻，你需要从“手头有什么就做什么”转变为优先处理你认为最重要的工作。

你是如何处理重要的事情的?

什么决定重要性？在一家科技公司里，重要性就是你的董事／副总裁／C 级高管所说的重要性。我是非常字面地这么说的。作为一名工程师，你的工作是执行公司的战略，而既然战略由高层制定，你的工作就是确保与你的领导认为重要的事情保持一致。在任何哪怕稍微运转正常的公司里，他们都会（一遍又一遍地）告诉你什么是重要的。等他们这么做的时候，你应该认真听。

要毫不留情地放弃那些不再重要的工作。如果你已经交付了一个项目，而你的管理层开始谈论下一件事，*stop improving that project*。以我的经验，在已经交付的项目上继续工作是一个非常常见的错误。宣布胜利，然后转身离开！

用于确定优先级的公式真的就这么简单：

1. 我现在在做最重要的事情吗？ 2. 如果不是，放下我正在做的事情，去做那个。

你应该每天至少问自己这个问题两次。这是我看到很多工程师犯的第一个错误：甚至没有真正尝试去优先排序，只是每天进来继续做前一天的工作，或者处理一个新的 JIRA 任务。

第二个错误是不清楚什么才是最重要的事情。以下是我在实践中如何弄清这一点的：

1. 是否有正在进行的高可见度事件（例如，董事/副总裁在询问）？如果有，我能帮忙吗？ 2. 在Slack/JIRA等平台上是否有我能回答的未解答问题？是否有人在等待我团队的回复？ 3. 对于我负责的每个项目，按重要性排序：它现在能发布吗？如果不能，我现在能做些什么来加速进度（需要部署的代码或需要审查的PR）？ 4. 对于我团队正在进行的每个项目，按重要性排序：是否有可用的工作可以接手，以推动项目进展？ 5. JIRA看板上是否有“准备工作”状态的任务？

请注意，访问JIRA看板是*last*步骤，而不是第一步。记住，公司关心的是项目，而不是票据。

如果你做对了这一点——如果你只把时间花在最重要的事情上——你每周工作不到40小时也能产生巨大的影响力。公司里充满了那些没有明确方向的工程师，每天做一两件有用的事情就能立刻让你变得异常高效。

这不是你经理的错。

这是我在工作中曾经最生气的一次经历。刚开始我的职业生涯时，我非常投入于保持一套集成测试的稳定性。那真的是一项艰苦的工作：随着应用程序的变化，保持测试的更新，确保有状态的部分（例如账户池）是健康的，尽可能提高测试对临时问题（比如慢连接）的抗压能力，等等。我为此付出了*months*的努力。我们从90%的失败率提升到了接近100%的成功率。然后在一次会议中，我的经理随口提了一句，暗示我不应该花这么多时间在这个上面，因为这并不那么重要。

我中途起身离开了会议——那是我前后唯一一次这么做。我去了一个空的会议室，坐下来，更新了我的简历。我感觉自己一直以来辛苦而重要的工作被完全否定了。事后看来，这是一件挺好笑的事。我当时真的很努力！但我的经理否定我所做的那些工作是对的，因为不管我在做什么，那都不是我的本职工作。

我已经见过这个故事上演很多次了。一名工程师认为某个特定任务很重要（比如，为某个冷门的输入格式添加支持，或者清理某些技术债务），于是花了数周甚至数月的时间去做这件事。他们做了所有正确的敏捷实践：尽早与经理沟通、创建问题、把工作拆分成模块化的 PR。然后，当这个项目第一次与某个 *real* 公司价值发生冲突时，他们被告知要放弃它，这让他们对所有被浪费的努力感到愤怒（或难过）。

这是管理失误吗？是，也不是。理论上，他们的经理本可以在早期就提醒他们，这项工作优先级很低，可能不值得他们投入时间。但当那些工程师告诉经理技术工作很重要时，经理往往会信任他们的高级工程师。经理在沟通优先级时也往往比较委婉：例如，他们会给出不温不火的回应，比如“当然，如果你认为这

值得去做就放手去做”而不是明确地说“我看不出这项工作的价值”。

还有一个原因使得经理并不是可靠的指引者：公司的宣称价值观往往与其真实价值观相悖。当我还是一名初级工程师、痴迷于集成测试时，我的经理并不会告诉我“其实，公司并没有那么在意 *that* 我们的集成测试是否不稳定”，因为公司在内部大量宣传集成测试的重要性。由我自己来决定要多认真地对待这些信息，并将其与我手头可做的其他任务进行权衡。

要警惕试图说服你的公司去做事情

到目前为止，交付价值最简单的方法就是找出你的组织正在做的最重要的事情，并在这方面提供帮助。判断公司所宣称的第十优先级是否真的有价值可能很困难，但公司所宣称的最高优先级几乎总是有价值的。你通常可以直接问你的经理（如果你有跨级的一对一，也可以问你的跨级经理）他们的首要优先事项是什么。或者，留意你的跨级层级及更高层在 Slack 或会议中都在询问和关注什么。

交付价值最困难的方式是找出你所在组织正在做的重要事情 *isn't*，并试图说服他们继续做这件事。这基本上是在打赌，你公司高层在识别他们需要什么方面做得不好，而你能够做得更好。

也许你可以！尤其是如果你的公司从事开发者工具（比如 GitHub），你可能会有一个有用的视角。但这是一项巨大的风险。即使你成功说服他们值得一看，你也不太可能把它放到他们优先级列表的前列。下一次风向改变时，你辛苦推动的项目很可能就不再重要了。然后你就会和我处在同样的位置，与

自动化测试：深度投入于公司不重视的艰难技术任务。

不要因为我而打消你去说服公司接受一个想法的念头。这是你能做的最有成就感的工作之一。只是如果你真的走上这条路，要保持谨慎——如果结果不如人意，你投入的时间可能会毫无回报。

总结

- 快速刷完 JIRA 工单只是个很酷的派对把戏，并非通往真正影响力的道路
- 要开展重要的工作，就要刻意关注你的管理链条认为重要的事情

- 对不重要的工作要果断放弃
- 不要依赖你的直属经理来告诉你什么值得去做
- 说服公司认为某项任务很重要可能行得通，但风险不小。

编写和阅读代码

这一切的基础技能是 *being good at the actual job*：也就是说，精通编写和阅读代码、系统设计等。如果你不能经常做对，就无法被信任；而如果没有真正的技术能力，你也不可能经常做对。那具体是什么样子？

优秀的软件设计看起来平淡无奇

多年前，我花了大量时间审阅编程挑战。这个挑战本身非常直接——构建一个 CLI 工具，调用一个 API，并允许用户对数据进行分页浏览和检查。我们允许使用任何语言，因此我看到了各种各样的实现方式。有一次，我遇到了一份我认为几乎完美的作品。那是一个单一的 Python 文件（总共大概三十行代码），以一种非常朴实、务实的风格编写：用最简单、最直接的方式满足了挑战的所有要求。

当我把它发给另一个评审，建议我们把它作为10/10的参考标准时，我真心震惊地听到他们说他们不会把这个挑战通过面试。根据他们的说法，这并没有展示出足够的对复杂语言特征的理解。它太*too*简单了。

多年后，我更加确信我当时是对的，那位评审是错的。优秀的软件设计应该是过于简单的。我现在想我终于能够开始阐明为什么。

消除风险

每个软件系统都有很多可能出错的地方。有时这些被称为系统的“故障模式”。以下是一个示例：

- SSL 证书会过期，并且不会自动续期
- 数据库填满并变得太慢或内存不足
- 用户数据被覆盖或损坏
- 用户会看到损坏的 UI 体验
- 核心用户流程（例如保存记录）无法运行

有两种设计潜在故障模式的方式。第一种是反应式的：在有风险的代码块周围添加救援条款，确保失败的 API 请求会被重试，设置优雅的

降级处理，以免错误导致整个体验崩溃，添加日志记录和度量指标，以便可以轻松识别bug，等等。这是值得做的。事实上，我相信这种（坦率地说，有点多疑的）态度是经验丰富的软件工程师的标志。但这样工作并不是良好设计的标志。它往往意味着你在掩盖糟糕设计中的缺陷。

应对潜在失效模式的第二种方法是从设计上将其消除。这在实践中意味着什么？

保护热路径

有时这意味着将组件移出热点路径。我曾经处理过一个目录端点（由于其他设计选择）非常低效，达到了每条记录~200毫秒。这使我们暴露于一些糟糕的故障模式：应用程序其他部分的资源饥饿、索引请求的代理超时，以及用户在等待十秒钟没有响应后直接放弃。最后我们将端点构建代码移到一个定时任务中，把结果存放在 blob 存储里，然后让目录端点提供这个 blob。我们仍然有那段每条记录 200 毫秒的低效代码，但现在它在我们的控制之下：它不会被用户操作触发，如果失败，最糟糕的情况是我们只会提供一个过时的 blob。

移除组件

有时这意味着干脆使用更少的组件。我参与过的另一个服务是一个文档 CRM，它有一套非常定制化的系统，用于从不同的代码仓库中拉取各类文档片段，并将它们拼接成数据库条目（有时甚至直接从代码注释中提取文档）。这在最初是一个不错的决定——当时，很难让各个团队写任何形式的文档，因此系统必须具备最大的灵活性。但随着公司的发展，它已经明显显得老旧了。同步任务

将一些状态存储在数据库中，另一些状态存储在磁盘上，当磁盘上的状态不同步或底层主机内存不足时，经常触发奇怪的 git 错误。我们最终完全移除了数据库，将所有文档移入一个中央存储库，并将文档页面重新设计为一个普通的静态网站。各种可能的运行时和操作错误就这样被消除了。

集中化状态

有时候这意味着规范化你的状态。最糟糕的故障模式之一是会导致你的状态（例如，你的数据库行）处于不一致或损坏的状态的错误：一个表说一个东西，但另一个表却说不同的内容。这很糟糕，因为修复错误只是工作的开始。你必须进去修复所有损坏的记录，这可能需要一些侦探工作，以弄清楚正确的值应该是什么（或者在最坏的情况下，只能猜测）。设计时确保你状态中关键部分有单一的真实来源，通常是值得承受其他很多痛苦的。

使用鲁棒系统

有时候，这意味着依赖经过战斗考验的系统。我最喜欢的例子是 Ruby Web 服务器 Unicorn。这是你在 Linux 上构建 Web 服务器的最直接、最简单的方式。首先，你需要一个监听套接字并一次处理一个请求的服务器进程。一次处理一个请求无法扩展：传入的请求会比服务器处理它们的速度更快地排队在套接字上。那么，你该怎么做呢？你需要多次分叉这个服务器进程。由于 fork 的工作方式，每个子进程已经在原始套接字上监听，因此标准的 Linux 套接字逻辑会将请求均匀地分配到各个服务器进程。如果出现问题，你可以终止子进程，并立即分叉另一个进程。

有些人认为喜欢Unicorn有点傻，因为显然它比线程服务器更不具备可扩展性。但我喜欢它有两个原因。首先，因为它将大量工作交给了进程和套接字的Linux原语。这很聪明，因为它们是极其可靠的。第二，因为Unicorn的工作进程很难对其他Unicorn工作进程做出任何恶意操作。进程隔离比线程隔离更可靠。这就是为什么Unicorn是大多数大型Rails公司（如Shopify、GitHub、Zendesk等）首选的Web服务器：伟大的软件设计并不意味着你的软件具有超高性能，而是意味着它非常适合任务。

总结

伟大的软件设计看起来很简单，因为它在设计阶段尽可能消除了许多故障模式。消除故障模式的最佳方法是*not*做一些令人兴奋的事情（或者如果可以的话，什么都不做）。

并非所有的故障模式都是一样的。你需要尽力消除那些真正可怕的故障模式（例如数据不一致），即使这意味着在其他地方做出稍显笨拙的选择。

这些都是相对无聊、缺乏吸引力的想法。但伟大的软件设计就是无聊和缺乏吸引力的。人们很容易对像CQRS、微服务或服务网格这样的大想法感到兴奋。伟大的软件设计看起来不像那些激动人心的大想法。大多数时候，它根本看起来像什么都没有。

在脑中设计软件

每当有人向我描述一款软件时，我都会思考如果是我会如何构建它。软件工程师经常这么做，但其中许多人做得并不好。我之所以这么认为，是因为我看到很多技术讨论纠缠于总体方案中的具体细节 *that could not pos- sibly work*。例如，争论是否使用 prop-drilling 或 context-passing 来向前端提供一份我们并不且永远不可能访问到的数据，或者在必须保持无状态的后端服务中实现何种精确的持久化数据存储策略。

为了避免这种情况，我认为在脑海中全程跟踪一个重要的用户流程是一个好主意。这在实践中意味着什么？

两种常见的反模式

我经常看到的第一个错误是停留在过于高层次。假设你正在为一个博客构建评论系统。一些工程师会停留在“哦，我会把评论放在某个关系型数据库里，然后把它们取出来放到页面上”。关系型数据库最终可能是正确的选择，但这种层次的设计对真正构建该功能并不太有用。你需要再深入一层：评论是如何从用户的浏览器传输到关系型数据库的？

第二个错误是过度纠结于错误的具体细节。有些工程师在开始设计评论系统时会说：“哦，酷，我要用 React”，然后就一头扎进无数微小的决策中，比如是否使用 RSC，或者通过 fetch 还是 TSQ 来获取数据，或者是否通过 GraphQL 暴露评论数据，等等。第一次听到这个问题的时候，并不是做出这类决策的合适时机。你最终可能不得不做这些决定，但不应该在一开始就做。

如何正确地做

那么正确的方法是什么呢？正确的方法是选择最重要的用户流程，并在脑海中追踪最简单的实现过程。你只能追踪一个用户流程，否则你会感到混乱。你必须把实现过程从头到尾追踪一遍，否则你会错过关键细节。当我说“追踪”时，我指的是伪代码级别的追踪。你不需要在脑海中想象整个代码，但你确实需要想象每一个逻辑步骤。

这相当于实用程序员著名的“追踪子弹”规则的心理等价物。追踪子弹规则是，你的第一个原型应该是构建所需的最小部分，以便让一个用户流程从头到尾正常工作。对于单纯思考写软件也是如此：你的目标应该是思考一个用户流程从头到尾。

在那样的细节层面把流程想清楚的好处在于，你会被迫直面重要的问题（就像你在用真实代码构建原型时一样）。你不必在这里设计最干净或最好的解决方案，但你确实需要设计 *something that could possibly work*。如果你从一个能工作的东西开始，通常可以通过迭代得到一个 *good* 能工作的东西。如果你从一个不能工作的东西开始，就很难通过迭代回到可行解的空间。

通过一个示例逐步讲解

例如，在实现评论系统的情况下：

用户访问我的博客文章之一时，应看到一个输入评论的表单。这可以通过向我的文章模板添加一个 `<form>` 元素轻松实现。

当他们提交表单时，他们的评论应该被存储在某个地方。好吧，我需要在我的后台创建一个端点和某种数据存储。我的添加评论端点代码大概是这样的：

```
comment = params['comment_body']
post = params['post_id']
user = ???

Comment.new(comment: comment, post: post, user: user).save!

redirect_to(post)
```

我如何设置评论的用户？如果人们匿名评论，解决方案很简单（为name添加一个可选的表单字段），但如果不是这样，我需要在我之前的静态网站上支持某种登录功能。这个端点可能会比较慢。用户提交评论的频率远低于查看页面的频率，如果花费一秒钟也没关系。

重定向之后，用户就应该能够查看他们的评论。这意味着我需要在帖子页面上加入类似这样的逻辑：

```
comments = Comment.where(post: current_post)
render(post, comments: comments)
```

然后在帖子页面进行一些HTML模板渲染每个评论。这个端点必须非常快。查看帖子是网站上的主要用户活动，添加几百毫秒的延迟会显著影响体验。这意味着可能需要缓存或延迟加载，并且一旦评论数量增加，就有必要进行分页。

即便是这样一个简单的例子，你也可以看到它如何暴露出我缺失的基础设施组件（在后端运行代码的能力、数据存储），以及我需要回答哪些问题（如何

用户是否可以登录或设置他们的身份）。当你在一家大型科技公司为一个系统执行此操作时，通常会遇到一些有趣的问题：

- 我们的系统需要数据X，但它只能从服务Y中的一个慢速端点获得。
- 我们的系统需要一些我们目前并未收集的数据（例如，我的静态博客不会收集有关用户身份的数据）
- 我们需要在每篇帖子上显示新的评论，但这些帖子目前在 CDN 上被长期缓存，因此我们需要找到一种在每条新评论出现时使该缓存失效的方式。
- 我们需要考虑一个或多个棘手的特性——例如，如果有人在运行博客的本地部署版本，我们就需要弄清楚可以在哪里存储评论（或者隐藏/禁用评论表单）

请注意，这些要点在很大程度上与所选择的具体技术无关（例如后端使用 Rails 或 Express，数据存储使用 MySQL 或 MongoDB 等）。它们所做的假设是一些直接源自需求的通用假设：无论如何，评论都必须被存储 *somewhere* 并与用户关联 *somehow*。

如何就心理计划进行沟通

你制定的计划大多应该留在你的脑海里。以我的经验，你无法将它有用地向产品经理，甚至其他工程师解释清楚。计划的价值在于它如何帮助你进行估算并提出问题。例如，为我的博客添加评论功能，最困难的部分将是切换到一种并非完全静态的基础设施（因此能够在用户请求中存储数据并运行我自己的代码）。对这部分工作的估算将为整个项目提供一个粗略的指引，而其中涉及的问题（例如我应该把博客切换到什么平台，或者是否应该使用第三方托管的评论服务）将是其中最重要的问题

规划项目。

一旦最初的讨论结束，把你的计划写下来会非常有用。我喜欢一种松散的“盒子和连线”结构，通常用 Mermaid 图来表示，但其实用什么方式并不重要。一小段文字可能就足够了。书面的计划可以作为进入更具体实现细节的良好起点。如果团队中的每个人都同意评论应由有状态的后端应用来管理，那么我们就可以开始讨论应该使用哪些技术以及如何实现。

我认为，如果你的团队有一个更明确的设计流程（例如协作式设计会议，或某种由架构师主导的方式），这种方法仍然有效。如果你带着一个关于该功能如何运作的具体想法进入这些流程，成功的可能性会大大提高。但有一个注意事项：不要对这个想法过于执着。你应该保持开放，愿意对计划进行大幅调整，只要新的方案同样可行即可。你最初在脑海中产生的那个粗略想法，整体来看不太可能是最佳选择。

总结

这就是那种看起来几乎太明显而不需要写下的观点——当你在规划工作时，当然应该考虑系统如何可能运行。但是我认为许多工程师低估了让任何东西都能正常工作的难度，因此忽视了他们应该最关注的具体细节。

如果你直接跳入那些具体的细节——哪些数据是可用的，它需要到达哪里，以及如何到达——你很可能会浪费更少的时间去争论一个本来就无法奏效的策略的实施细节。

在大型已有代码库中工作

在大型成熟的代码库中工作是大科技公司中最重要的技能，也是软件工程师最难学习的事情之一。你无法事先练习它（很少有开源项目足够大，能给你相同的经验）。个人项目永远无法教会你如何做到这一点，因为它们必然是小规模的，且是从零开始的。为了澄清，当我说“大型成熟的代码库”时，我指的是：

- 单一数字百万行代码 (~5M, 假设如此)
- 大约有100到1000名工程师在同一代码库上工作
- 该代码库的第一个可用版本至少已有十年历史

我已经在这些代码库中工作了十年。以下是我希望在一开始就知道的事情。

最根本的错误是不一致性

我看到的一个错误比其他错误更常见，而且它是致命的：忽视其余的代码库，仅以最合理的方式实现你的功能。换句话说，就是限制与现有代码库的接触点，以保持你干净的代码不受遗留代码的污染。对于主要在小型代码库上工作的工程师来说，这个做法是非常难以抗拒的。但你必须抵制它！实际上，你必须尽可能深入到遗留代码库中，以保持一致性。

为什么在大型代码库中一致性如此重要？因为它能保护你免受意外的困扰，减缓代码库进展成混乱的速度，并且让你能够利用未来的改进。

假设你正在为某种特定类型的用户构建一个 API 端点。你可以在端点中加入一些“如果当前用户不是该类型，则返回 403”的逻辑。但你应该首先查看一下代码库中其他 API 端点是如何处理身份验证的。如果它们使用了一些特定的辅助工具，你也应该使用那些工具（即使它们很丑陋、难以集成，或者看起来对你的用例来说是多余的）。你必须抵制将你代码库的这一小部分做得比其他部分更漂亮的冲动。

这样做的主要原因是因为大型代码库中有很多地雷。例如，您可能不知道代码库中有一个“机器人”的概念，机器人类似于用户，但又不完全是用户，并且需要特殊的身份验证处理。您可能不知道代码库中的内部支持工具允许工程师有时代表用户进行身份验证，这也需要特殊的身份验证处理。肯定还有其他一百件您可能不知道的事情。现有的功能代表了通过雷区的安全路径。如果您像其他长期存在的 API 端点一样进行身份验证，您可以沿着这条路径走，而不必知道代码库中所有令人惊讶的功能。

此外，缺乏一致性是大型代码库的主要长期杀手，因为它使得任何通用的改进变得不可能。以我们的认证示例为例，如果你希望引入一种新的用户类型，一致的代码库让你能够更新现有的认证助手集以适应这一变化。在一个不一致的代码库中，由于某些API端点的实现方式不同，你将不得不去更新和测试每一个实现。实际上，这意味着通用的更改不会发生，或者最难更新的5%的端点就被排除在外——这反过来又进一步降低了一致性，因为现在你有一个只适用于大部分（但不是全部）API端点的用户类型。

所以，当你坐下来在一个大型代码库中实现任何东西时 se，你

应该始终首先去查看现有技术，并尽可能遵循它。

还有其他重要的事情吗?

一致性是最重要的。不过，我也快速过一下其他一些关注点：

你需要培养一种对服务在实际使用中（即由用户使用）如何运作的良好感知。哪些端点被访问得最频繁？哪些端点最为关键（即被付费客户使用，并且不能优雅地降级）？服务必须遵守哪些延迟保证，哪些代码在热路径中执行？一个常见的大型代码库错误是进行“微小调整”，而这个调整意外地进入了关键流程的热路径，从而引发了大问题。

你不能像在小型项目中那样，依赖在开发阶段测试代码的能力。任何大型项目都会随着时间的推移不断积累状态（例如，你觉得 GMail 支持多少种类型的用户？）到了一定阶段，即使借助自动化，你也无法测试所有状态组合。相反，你必须测试关键路径，采取防御式编码，并依赖缓慢的发布和监控来发现问题。

要非常、非常谨慎地引入新的依赖。在大型代码库中，代码往往会长期存在。依赖会带来持续的成本，包括安全漏洞和包更新，而这些几乎肯定会比你在公司的任期还要长。如果确实必须引入依赖，务必选择被广泛使用且可靠的，或者在需要时易于分叉维护的依赖。

基于相关的原因，如果你有机会删除代码，一定要牢牢抓住。这是在大型代码库中风险最高的工作之一，所以千万别敷衍了事：首先对代码进行插桩，在生产环境中识别调用方，并把它们逐步降到零，这样你才能百分之百确定可以安全删除。但这仍然值得去做。在一个……中，几乎没有什么事情能比得上这一点。

以小规模的 PR 工作，并将影响其他团队代码的改动前置。在小项目中这一点也很重要，但在大型项目中则至关重要。原因在于，你往往需要依赖其他团队的领域专家来预见你所遗漏的事项（因为大型项目过于复杂，不可能由你一个人全部预见）。如果你将对高风险区域的改动控制得小而且易于阅读，这些领域专家就更有可能发现问题，从而避免一次事故。

为什么要烦恼?

最后，我想花一点时间来为这些代码库辩护。一个我常听到的观点是这样的:

> 为什么你会决定在遗留的烂摊子中工作？花时间在乱七八糟的代码中可能很难，但这不是好的工程实践。当面对一个庞大的现有代码库时，我们的工作应该是通过拆分出小而优雅的服务来缩小它，而不是陷入其中并让烂摊子变得更大。

我认为这种观点完全是错误的。主要原因是，作为一个普遍的规则，大型已建立的代码库产生了 90% 的价值。在任何大型科技公司中，绝大多数的收入生成活动（即真正支付你工程师工资的工作）都来自于一个大型已建立的代码库。如果你在一家大型科技公司工作，并且认为这不成立，也许你是对的，但我只有在你对你认为没有提供价值的大型已建立代码库有深入了解的情况下，才会认真考虑这个观点。我见过多次情况，其中一个小而优雅的服务驱动了高收入产品的核心功能，但所有实际的产品化代码（设置、用户管理、计费、企业报告等）仍然存在于大型代码库中。

已建立的代码库。

所以你应该知道如何在“遗留的烂摊子”中工作，因为这正是你公司实际上在做的事情。无论工程做得好不好，*it's your job*。

另一个原因是，在没有首先理解一个大型现有代码库的情况下，你无法将其拆分。我见过成功拆分大型代码库的例子，但我从未见过没有已经熟练于在大型代码库中交付功能的团队做到这一点。你根本无法从零开始重新设计任何非琐碎的项目（即，能带来实际收入的项目）。有太多意外的细节支撑着数千万美元的收入。

总结

- 大型代码库值得从事，因为它们通常会支付你的薪水

- 迄今为止，最重要的是一致性
- 在开始一个功能之前，务必先研究代码库中的前期工作

- 如果你不遵循现有的模式，你最好有一个非常充分的理由。

- 了解代码库在生产环境中的足迹
- 不要指望能够测试每一种情况——相反，应依赖监控

- 尽量去除代码，但要非常小心。
- 让领域专家尽可能容易地发现你的错误

绕过棘手的特性

为什么在大型科技公司工作如此艰难?

这是因为少数“恶性特征”主导了其他一切。如果你正在构建一个待办事项应用，添加将图片附加到待办事项的功能可能是一个大功能，但它不是一个恶性特征。然而，将你的待办事项应用提供为网络应用和独立可执行文件就是一个恶性特征。有什么区别？恶性特征是必须考虑的功能 *every time you build any other feature*。

以下是一些强大的特性示例:

- 添加新用户类型
- 添加您的SaaS的本地版本
- 将客户分片到多个不同的数据库中
- 支持强数据局部性
- 支持客户在不同区域之间迁移账户的功能

- I18n（将面向客户的文本翻译成他们的母语）

假设你已经完成了所有这些工作，现在正在构建图像附件功能。新用户能否添加图像？假设你通常将图像存储在 S3 中——在 S3 不可用的本地环境中，图像存储在哪里？如果客户数据是分片的，你是否也在适当地分片 `images` 表？你是否确保为每个用户区域都准备了一个 S3 存储桶？如果客户将数据迁移到其他区域，你是否有自动化系统来将 S3 图像一同迁移？你是否已经提取了所有与图像附件相关的新字符串，并且已经预估了翻译它们所需的时间？

为什么棘手的特性很难

邪恶特性就像密码游戏。在密码游戏中，新的规则——例如“你的密码必须包含其长度作为一个数字”，或“密码中的所有数字之和必须为200”——不能单独考虑和解决。它们必须作为一个整体来解决，因为为了适应一个规则而改变解决方案往往会破坏其他几个规则。事实上，密码游戏非常慷慨，它会立即告诉你当前哪些规则被破坏了以及为什么。在大型技术项目中，你通常会从用户的工单或事件中得知。

这是工程师低估任务的一个常见原因。人们很容易忘记一个或多个会使实现复杂化的棘手特性，随后当有人问起“那 X 呢？”时就会猝不及防。对于那些在公司待的时间不长、可能干脆就不知道某些棘手特性的工程师来说，这一点尤为明显。公司的“老兵”之所以有价值，很大程度上是因为他们熟悉所有这些棘手的特性。

邪恶特性是技能问题吗?

恶劣的特性仅仅是糟糕的设计吗？难道你不能更好地将程序进行模块化，以满足需求而不添加恶劣的特性吗？当然，有时可以。我相信你可以通过足够笨拙的实现让任何特性变得恶劣。但我认为有些需求本身就是恶劣的。

以“让这个SaaS可以在本地运行”为例。无论你多么小心，这都无关紧要。即使你确保你的SaaS构建管道完全适应本地环境，以便你不必维护两个版本，然而你必须小心做到这一点，本身就是一个需要牢记的复杂特性，必须在未来对构建管道的所有更改中时刻考虑。

或者以“添加一种可以做 X 但不能做 Y 的新用户类型”为例。假设你

做好重构用户能力的工作，这样你就不必再做 `isUserTypeX(user)`。新功能必须适应你自制的用户能力框架，这本身就是一个棘手的特性。

这些功能的棘手之处不在于实现，而在于根本性的领域模型。问题在用户流程图的层面就已经很棘手了。无论你的代码结构分解得多么好，你仍然必须回答我上面列出的那些问题（例如：“我正在构建的这个新能力是否能被所有用户类型访问？”）。

为什么要构建邪恶的功能？

如果公司能够避免构建不良功能，它们会这么做。问题在于，付费最高的用户*love*不良功能。本地部署的 SaaS 产品通常极其盈利，因为它们吸引的是一群财务状况非常良好、并且愿意支付高昂企业软件合同价格（而不是低廉的 SaaS 订阅价格）的用户。同样，数据本地化和分片也对财力雄厚的企业客户具有吸引力。

其他一些有害的功能则是由懒惰或无能的开发者构建出来的。比如，有些公司只有五个用户，却仍然为其数据搭建了一整套完整的分片系统，仅仅因为相关工程师觉得这样做听起来很有趣。我也见过工程师试图构建有害的功能，因为他们看不到其他做事方式（或者只是觉得那样做才是“正确”的方式）。对于一个只支持一种语言的应用来说，把所有面向用户的字符串都抽取出来，就是一个经典的低风险例子。

作为一名工程师，你能做的最有价值的事情之一，是尽可能防止团队构建有害的功能；而在这些功能必须构建时，通过合理的模块划分来限制其造成的损害，并着眼于“这将如何影响在同一系统中试图构建完全不相关功能的开发者？”

总结

- 邪恶特性是每次构建其他任何东西时必须考虑的需求
- 它们大幅增加了实施复杂性和协调成本。
- 某些恶劣特性是不可避免的，尤其是在向高付费企业用户销售时
- 另一些则是由过度工程化、教条主义或品味不佳自我造成的
- 优秀的工程师限制爆炸半径：他们避免不必要的恶劣特性，并将必要的特性进行隔离，以免污染整个系统。

玩政治

假设你是一个出色的工程师，公司信任你运用你的能力来支持公司的目标。你正处于一个有利的位置：晋升、奖金和高曝光度的项目应该会纷至沓来。但随着这种增加的曝光度，也会带来一系列新的问题——政治问题。

保护你的时间不被掠夺者侵占

如果你是一家大型科技公司的称职软件工程师，你的时间需求会非常高。很多人都会希望你去做各种事情。你应该对如何处理这些请求非常慎重，并且一定要避免对所有人都说“是”。

帮助他人感觉很好。当这些人来自公司其他部门时，这种感觉更加强烈。你会觉得自己正在发挥一种跨部门的影响力，这是一个员工+工程师应该拥有的。然而，帮助公司其他部门并不是你的主要工作。交付项目才是你的主要工作。一个常见的陷阱是过于慷慨地投入时间，忽视那些实际上是你责任范围内的项目。为了避免这种情况，你应该识别那些试图占用你时间的人。

识别捕食者

大型科技公司充满了掠夺者：那些想要从乐于付出的工程师那里提取未经报酬的工作的人。一旦掠夺者识别到一个合适的目标，他们会通过私信而不是通过正常渠道向该人发送工作。“未经报酬”是这里的关键词。当你的经理要求你做工作时，这不是掠夺行为，因为你为此得到了报酬，并且（希望）在评估时会得到奖励。当同事要求你做工作时，这也不是掠夺行为，因为他们也有可能帮助你。掠夺者要求你做的工作对他们来说有很大价值，但对你没有任何好处（甚至可能有害）。让我们看一些例子。

一种常见的“掠食者”类型是来自组织中不同部门的产品经理。我想明确一点，大多数产品经理并不会这样做。但在任何大型组织中，产品团队里通常都会有一两个这种类型的人，因为他们往往是以结果为导向的人。

他们习惯于从工程师身上榨取工作。这种掠食者的请求在 Slack 私信中看起来像这样：

- 我知道你从技术上来说并不拥有这部分代码库，但你能帮我快速做个小改动吗？
- 我从另一个PM那里得到了这个（与您的工作无关的）技术问题，答案是什么？
- 你在使用我们的数据工具方面真棒，等你有空时能帮我拉取一些统计数据吗？
- 我的团队正在做一个与您代码库相关的项目，您能在这里做个更改以解锁他们吗？

这对产品经理来说很棒，因为他们能迅速解除阻碍，而且不必花费任何政治资本来推动工作的优先级。而对工程师来说，这不好，因为他们忽视了自己的实际项目，并疏远了直接经理（经理会对工程师被直接接触感到不满）。来自其他组织的那位产品经理不会出现在你的评审会议上，一旦你停止提供免费的工作，他们的感激之情可能会迅速消失。

还有另一类常见的掠食者：那些寻找你为他们做工作的人。来自他们的请求通常看起来像是源源不断的这样：

- “我刚拿到这张票，不知道从哪里开始”
- “救命，我的测试都红了 [PR 链接，没有更多上下文]”
- 我们能在我正在处理的这个票上配对吗？[配对完全是你开车，他们观看]

在极端情况下，弱工程师几乎没有自己完成任何工作，而是依靠一系列强工程师朋友来完成每个任务。对于每个强工程师来说，看起来他们只是稍微帮了一下忙。这对强工程师来说是一个不划算的交易，因为弱工程师 (a) 不想公开他们依赖强工程师的程度，因此不会共享功劳，且 (b) 不会

以后能够在工作上帮助那位能力很强的工程师。

为了明确，我并不是在说一个正在学习的初级工程师——他们在获得相关技能后会帮助你。我是在说一个依赖他人帮助作为避免实际学习工作的人。你应该对正在学习的工程师非常慷慨地提供时间，而对其他人则要非常谨慎。

并非所有求助请求都是掠夺性的

并非所有的求助请求都是掠夺性的。帮助团队中的工程师是你工作的一部分，而跨部门的影响有时确实涉及帮助他人，即使你没有得到任何回报。掠夺性行为是指在没有回报的情况下消耗你的时间。一个警告信号是，当请求本身非常低效，但却要求你付出大量的努力。另一个辨别方法是，掠夺者的请求通常通过非正式渠道（例如 Slack 私信）传达。公开请求会（a）明确他们的目的，（b）至少让你得到公开的帮助认可，且（c）给你的经理一个机会介入，保护你的时间。

我想要特别强调的是，根据定义，来自你的经理或任何管理链中的人的请求并不是掠夺性的。这些人将参与你的评审会议，因此处于直接奖励你工作的位置。帮助他们直接而明确地是你的工作，你应该优先考虑它，除非是最关键的任务。

避免时间非对称请求

一个有效的经验法则来避免成为猎物是警惕针对你时间的非对称DoS攻击。对那些比回答更容易提问的问题保持怀疑。换句话说，要保持警觉。

当某人只投入了很少的精力来提出一个问题，却需要你付出大量工作来回答时。“我刚接手这个工单就迷失了方向”是一个低投入的问题，但需要你付出很多精力才能回答。“我刚接手这个工单，已经尝试了这个（这是我的 PR 草稿），但因为这些原因我不喜欢它”则是一个高投入的问题。

当有人向你提出一个低投入的问题时，就给一个低投入的回答。这并不意味着给出错误的答案或不礼貌的回答——只是别去做一大堆工作。有时这意味着回复“抱歉，我一时想不起来”，或者“是的，我记得我们在某处有这方面的数据，但想不起在哪”，而不是实际去查找。有时这意味着保持模糊，比如回复“是的，我觉得你需要更新控制器里的认证逻辑”，而不是去找到并链接具体的文件。

这样做有几个好处。首先，它会打消那些寻找容易下手目标的掠食者。其次，它会立刻腾出你的时间。第三，它会鼓励与你共事的人提出更费心思的问题，这对每个人都有好处。

总结

- 如果你是一名优秀的软件工程师，你的时间将不断受到侵扰
- 帮助他人不是你的主要工作，把工作做好才是
- 大型科技公司充满了猎人，他们寻找那些能够利用时间的优秀工程师。
- 当心来自不同组织的产品经理和薄弱的工程师
- 掠夺者没有权力奖励你（也就是说，他们不在你的管理链条中）
- 不要让你的时间被不成比例地消耗

与冷酷的经理合作

工程经理主要有两种：富有同理心的和铁腕的。出于几个原因，我认为铁腕型经理被低估了。

富有同理心的管理者在乎员工。他们在情感上把员工当作人来投入，并积极为支持员工的需求而奔走。冷酷的管理者只是来把工作做好。他们并不是*necessarily*混蛋，但他们将自己的主要角色视为在公司与工程师之间传达需求、并在两者之间沟通。他们几乎从不为了员工的利益而冒险出头。

总体来说，拥有一位富有同情心的经理是件好事。经理对他们的工程师有一定的权力——虽然比许多工程师想象的要少——如果能以宽容和善良来行使这种权力，那是很好的。但我其实不介意无情的经理。

首先，如果经理和工程师都称职，那么即便是冷酷无情的经理也仍然希望工程师快乐。在其他条件相同的情况下，快乐的工程师工作得更好，也更容易管理。一个纯粹自利的经理，只要不需要付出重大的个人代价，就会做出让工程师感到快乐的事情。出于同样的原因，冷酷无情的*companies* 也希望他们（称职的）工程师快乐。在大型科技公司里，工程师需要时间才能变得真正有用。如果你让工程师保持快乐，他们就会待得更久，产出也更高效。所以，“冷酷无情”并不总是意味着“残忍”。

第二，富有同理心的管理者往往会与自己的上级发生冲突，这使他们几乎没有政治资本。大型公司让管理者承担大量与绩效管理、解雇员工、施加紧迫期限等相关的脏活累活。富有同理心的管理者会不断对此进行反对。如果这种反对是有效的，那当然是好事，但往往并非如此。如果你的经理总是在和他们的上级对抗，那么他们可能已经没有剩余的政治资本去为你真正需要的少数事情争取（例如晋升）。然而，

一个冷酷无情的经理通常会在其上级那里积累大量的政治资本。说服他们去推动你所需要的事情会更困难，但如果你成功了，他们也极有可能真的把事情办成。

第三，由于相同的原因，同理心强的经理通常不太快乐。假设你团队里有一个非常喜欢写 Python 的工程师，但上级管理层要求他们开始使用 C# 编程。你的经理将不得不在两者之间做出选择，要么迫使工程师写 C#，要么与老板争论这个政策有多愚蠢。一位无情的经理会 (a) 不会对决定感到痛苦，且 (b) 会毫不犹豫地迫使工程师去做，而不会感到内疚。如果你不是那位工程师，这对你来说是更好的，因为你的经理不会感到难过和分心。

第四，冷酷的管理者往往更擅长沟通。富有同理心的管理者不喜欢传达坏消息，可能会过度粉饰。在我的经验中，他们有时也会被公司“洗脑”，给你的只是官方口径的回答，而不是更有用（或更犬儒的）真相。然而，冷酷的管理者也可能因不同原因在沟通上失败：通常是因为他们更乐于故意对你撒谎，或者因为担心说出真相会反噬自己而不愿完全坦诚。

第五，冷酷的经理更容易预测。冷酷的经理总是做他们自己的上级——以及公司整体——所重视的事情。如果公司高管设定了优先级，冷酷的经理一定会遵循。有同理心的经理有自己的优先级，因此更难知道他们想要什么，或者他们会奖励什么样的行为。为冷酷的经理工作有时也挺不错，因为他们更容易理解（至少在工作中的形象是这样）。

当然，我说的是在相对健康的公司中的*competent*经理（即不是一家濒临死亡或严重失衡的公司）

如果经理的激励与工程师完全不一致，那么拥有一个冷酷的经理是没有任何好处的。如果你的经理不称职，你也不希望他们冷酷无情——那样只会造成不必要的破坏。如果必须在冷酷和富有同情心之间做选择，你可能仍然会选择富有同情心。但是既然你无法选择，意识到这两种方式的优缺点也是件好事。

大型语言模型

到目前为止，本书一直在讨论由2010年代末零利率时代的终结所引发的工程工作中的重大转变。但工程工作中正在发生另一场重大转变，它可能对作为工程师的工作体验产生更大的影响。我指的是大型语言模型。

软件工程师没有理由对大型语言模型的前景持怀疑态度。我不认为它们现在已经能写出非常好的诗歌或文章。但它们确实能够写出可运行的代码。它们不需要变得好很多，就能独立完成工作中很大一部分。因此，现在就弄清楚如何使用它们是个好主意。

我如何使用 LLMs

软件工程师在大型语言模型这个话题上意见分歧。许多人认为它们是有史以来对行业影响最大的技术。另一些人则认为它们是一个又一个仅仅是炒作的产品中的最新者：虽然令人兴奋，但最终对那些试图做严肃工作的专业人士来说并没有实际用处。

就我个人而言，我觉得我从 AI 中获得了很大的价值。我认为许多没有这种感觉的人是“用错了方式”：也就是说，他们没有以最有帮助的方式使用语言模型。在这篇文章中，我将列出作为一名资深工程师，我在日常工作中经常使用 AI 的多种方式。

编写生产代码

我每次写代码都会使用 Copilot 的补全。我接受的补全几乎全部都是完整的样板代码（例如补齐函数参数或类型）。我很少让 Copilot 为我生成业务逻辑，但偶尔也会发生。在我擅长的领域（例如 Ruby on Rails），我有信心能比 LLM 做得更好。它只是一个（非常好的）自动补全。

然而，我并不总是在自己的专业领域内工作。我经常发现自己会在不太熟悉的领域里做一些小的战术性改动（例如一个 Golang 服务或一个 C 库）。我了解这些语言的语法，也用它们写过个人项目，但对什么是地道的写法不太有把握。在这种情况下，我会更多地依赖 Copilot。通常我会使用启用了 o1 模型的 Copilot Chat，把我的代码粘贴进去，然后直接问：“这样写算是地道的 C 吗？”

像这样更多地依赖 LLM 是有风险的，因为我不知道自己遗漏了什么。它基本上让我在各个方面都以一个聪明实习生的基线水平来运作。我也必须像一个理智的实习生那样行事，并确保该领域的主题专家审查该变更，以

我。但是即便有这个警告，我认为能够迅速做出这种战术性调整是非常高杠杆的。

编写临时代码

在编写那些永远不会进入生产环境的代码时，我使用 LLM 要大胆得多。比如，我最近做了一项研究，需要从一个 API 拉取公共数据片段，对其进行分类，并用一系列快速的正则表达式来近似这种分类。所有这些代码都只在我的笔记本电脑上运行，而我基本上用 LLM 写了其中的全部内容：用于拉取数据的代码、用于运行另一个 LLM 进行分类的代码、用于对其进行分词并统计词元频率和打分的代码，等等。

LLM 非常擅长编写那些可以正常运行但不需要维护的代码。只运行一次的非生产代码（例如用于研究）与此完美契合。我会说，在这里使用 LLM 让我完成这项工作的速度比在没有辅助的情况下快了 2x-4x。

学习新领域

我使用 LLMs 做的事情中，可能最有用的一件就是把它当作一个按需的导师，用来学习新的领域。比如，上周末我学习了 Unity 的基础，主要依赖 ChatGPT-4o。使用 LLM 学习的魔力在于，你可以 *ask questions*：不仅仅是“X 是如何工作的”，还可以提出诸如“X 与 Y 有什么关系”这样的追问。更有用的是，你可以问“这样理解对吗”之类的问题。我经常把自己认为学到的东西整理成文字再反馈给 LLM，它会指出哪些地方我是对的，哪些地方仍然存在误解。我会向 LLM 提出 *a lot* 的问题。

我在学习新东西时会记很多笔记。能够直接把我所有的笔记复制粘贴进去，让 LLM 帮我审阅，真是太棒了。

那幻觉怎么样？老实说，自从GPT-3.5以来，我没有注意到ChatGPT或Claude出现很多幻觉。我尝试学习的大多数领域都已被很好地理解（只是我还不懂），而根据我的经验，这意味着幻觉的概率非常低。我从未遇到过通过LLM学到的东西最终证明是根本错误或是幻觉的情况。

最后的救命补丁

我不常这样做，但有时候当我在一个 bug 上卡住时，我会把整个文件或文件们附加到 Copilot 聊天中，粘贴错误信息，然后直接问：“你能帮忙吗？”

我之所以不这么做，是因为我认为目前我在找bug方面比当前的AI模型要强得多。几乎每次，Copilot（或者在一些个人项目中使用Claude）都会搞错。但如果我真的卡住了，还是值得尝试一下，毕竟它的投入非常低。我记得有两三次，我只是错过了一些微妙的行为，而LLM捕捉到了，帮我节省了大量时间。

因为大型语言模型（LLMs）在这方面还不够成熟，我不会花很多时间进行迭代或试图解锁LLM。我只是尝试一次，看看它能否完成。

校对拼写错误和逻辑错误

我写了不少英语文件：ADR、技术总结、内部帖子等等。我从不允许LLM为我写这些。部分原因是我认为我能写得比当前的LLM更清晰；另一部分原因是我对ChatGPT的写作风格普遍不喜欢。

我偶尔会将草稿输入到LLM中并请求反馈。LLM在捕捉拼写错误方面非常出色，有时还会提出一个有趣的观点，这会成为我草稿的修改。

像修复bug一样，当我做这件事时，我不会反复修改——我只会要求一次反馈。通常，大型语言模型会提供一些风格上的反馈，我总是忽略这些。

总结

我使用 LLM 来完成这些任务：

- Copilot 智能自动补全
- 在我不太熟悉的领域进行的小幅战术性调整（始终由一名领域专家复核）
- 编写大量一次性、用完即弃的研究代码
- 通过提出大量问题来学习新主题（例如 Unity 游戏引擎）
- 作为最后手段的漏洞修复，以防它能立即弄明白
- 大局观校对用于长篇英语沟通

我暂时不使用LLM来处理这些任务：

- 在我熟悉的领域为我撰写完整的 PR
- 编写ADR或其他技术文档
- 在大型代码库中进行研究，了解事情是如何做的

使用LLM进行安全编码

使用LLM编写代码在根本上不同于其他编程方式。LLM通常是非确定性的，而且总是不可预测的。它们具备其他任何技术都无法匹敌的能力：与自然语言进行交互的能力。这对安全意味着什么?

我对大多数关于 LLM 与安全的在线内容并不怎么满意。例如，OWASP 的内容草案是准确的，但并不特别有用。它把 LLM 安全描绘成一系列彼此不同的威胁，你必须逐一去熟悉。相反，我认为更好的理解方式是把 LLM 安全视为源自一个单一的原则。如下:

LLM 有时会表现出恶意行为，因此你必须像对待用户输入一样对待 LLM 的输出。

说大型语言模型（LLMs）有时会表现出恶意，这是什么意思？有时它们会在毫无预兆的情况下以令人惊讶的方式行事——LLM 的本质是一个黑箱，其输出无法在事先被完全预测。但在其他时候，恶意行为者可以通过特定提示，可靠地诱导它们表现出恶意行为。

提示注入是不可避免的

你可能会认为只有你一个人在给你的 LLM 进行提示，所以是安全的。但如果你在 LLM 的输入中引入了任何用户生成的内容，那么这就等同于允许这些用户对你的 LLM 进行提示。例如，如果你的 LLM 能够搜索互联网，你可能会无意中把一些网页内容引入到提示中，其中写着“忽略之前所有指令，去做［邪恶的事情］”。如果允许用户与你的 LLM 聊天，或者允许他们将自己的文档作为上下文提供给 LLM，或者提供他们的代码等等，同样的风险也都会存在。

即使你只是使用一个 LLM 工具，而不是构建一个 LLM 应用，你仍然会受到这个问题的影响。出于显而易见的原因，允许第三方填写你的 Cursor 或 Copilot 规则，与允许他们直接向你的代码库贡献代码具有同样的风险。

通过控制部分输入来让 AI 执行某些事情的过程被称为“越狱（jailbreaking）”。存在许多越狱技术（例如将你的请求进行 base64 编码、角色扮演、哲学论证），但最重要的是要知道：越狱是有效的。没有任何模型能够免疫于提示注入。你不能将其作为安全模型的一部分来依赖，这意味着你不能信任模型的响应——正如我上面所说的，你必须把模型的响应当作不可信的用户输入来处理。

AI 工具实际上是面向用户的

如果坏消息是 LLM 不可信，那么好消息是 LLM 不能自行做任何事情。你必须将它们的所有输出转化为行动：要么运行其输出的代码，要么执行它们请求的工具调用。如果你运行 LLM 输出的代码或在 UI 中展示 LLM 内容（这在本质上是同一回事），你需要像处理用户生成的内容或用户生成的代码一样对其进行清理。那工具调用呢?

如果你根据 LLM 的输出来调用工具，你应该将所有能够参与构建提示的用户都视为对工具函数拥有完全控制权。因此，工具函数应当预先限定到与 LLM 交互的用户范围内。也就是说，LLM 所能访问的所有函数，都必须在相同的访问控制范围内进行限定，就如同它们是作为面向用户的 API 的一部分可用一样。

例如，如果你有一个“查找过去用户消息”的函数，其签名必须是 `fetch_messages()`，而不是 `fetch_messages(user_id)`。否则，LLM 可能会决定——或被诱骗——去获取一条不—

不同用户的消息和泄露他们的数据。如果工具的范围设置得当，用户可以使用他们想要的任何输入来调用工具。在最坏的情况下，这意味着用户只会泄露或删除他们自己的数据。

影响多个用户的操作工具（例如 `send_message`、`make_transaction`）更加危险。对于这些工具，您应该让用户手动批准操作，或者确保没有其他用户可以向提示提供上下文。如果模型能够执行网页搜索并根据搜索结果采取行动，您就有可能面临搜索结果返回一个页面，指示模型以不恰当的方式调用 `make_transaction`（例如，可能会耗尽当前用户的余额）。

你应该对通常功能非常强大的 LLM 工具格外谨慎，例如那些可以执行任意 Python 或 shell 命令的工具。如果你不会将这种功能作为面向用户的 API 暴露出来，那么你也不应该将其作为 LLM 工具暴露。如果你确信自己已经足够谨慎地对其进行了沙箱隔离——就像我相信 OpenAI 和其他 AI 实验室为其代码执行工具所做的那样——那就放手去做吧！但你最好真的有足够的把握。

MCP 服务器会让你暴露于供应链风险之中

如果你依赖任何与 LLM 接口的第三方代码，你就在信任这些代码不会恶意地向模型注入提示。库是这种攻击向量中一个众所周知的例子：当然，如果你引入了不可信的第三方 AI 库，你就麻烦了。模型上下文协议（Model Context Protocol）服务器是这种风险中一个较少被理解的例子。如果你连接到一个 MCP 服务器，你实际上是在引入一个第三方库。最坏的情况下，它是一个第三方库，其中每个函数都封装了对其自有服务器的 API 调用，因此你引入的是一个库 *where the implementation is unauditable and can change at any moment*。

许多关于“为什么MCP本质上不安全”的文章列出了大量的例子，说明连接到一个恶意的MCP服务器可能会完全搞砸你的系统。大致有两类情况：一是引入恶意的提示，指示模型做坏事；二是引入恶意工具，这些工具在执行其主要功能的同时，作为副作用做坏事（例如，一个`web_search`工具，同时将你的`./ssh`文件夹内容发送给第三方）。

我并不完全相信这代表了 MCP 规范的问题，就像能够导入恶意 Python 库并不代表 Python `requirements.txt` 规范存在问题一样。供应链安全问题并不是 LLM 独有的。所有你通常用来降低库导入风险的措施——尽可能锁定版本、阅读源代码、限制对最敏感操作的暴露——同样适用于 MCP 服务器。

AI 可以自行变得恶意。

即使你只是为个人用途编写一个程序，在赋予 LLM 访问强大工具的权限时也仍然需要保持一定的谨慎。LLM 往往高度以结果为导向，有时会不惜一切代价去实现你设定的目标。假设你正在“氛围式编程”一个工具，让你的工作笔记可以从所有设备上访问。你足够聪明，知道通过公共服务器暴露整个本地文件系统在技术上*technically*可行，但并不是一个好的解决方案。然而，一个在解决问题时陷入困境的 AI 代理，可能还是会这么做。

换句话说，不需要恶意的第三方提示注入就能说服您的LLM做一些不安全的事情。它可以独立完成这一切。如果您仍然希望使用强大的工具，一种明智的策略是建立一个“人类在环”步骤，您需要批准任何 shell 命令或 Python 代码。确保将这个步骤构建到您的工具 *code* 中，而不是工具 *prompt*。

也值得记住，LLMs 有时会直接产生幻觉：也就是说，它们会毫无理由地错得离谱。因此，即使提示没有恶意，且 LLM 也不试图像学徒魔法师那样完成目标，你仍然不能安全地信任输出结果。

未对齐且不安全的模型

一些LLM模型比其他模型更安全。在极端情况下，可能会训练出一个对齐不良的模型（即，它的行为是妨碍人类目标的实现，而不是实现这些目标）。这样的模型可能故意暴露你的数据或破坏你的应用程序。人工智能实验室投入了大量的精力来确保他们的模型不会落入这一类别，因此通过使用流行且值得信赖的模型，你可以在很大程度上避免这种风险。如果你在训练自己的模型，那么这是你的责任。

模型不安全的另一种方式是它们可能（通常是偶然地）包含私人信息。如果你在敏感用户数据上训练或微调一个模型，那么你的模型就不能安全地公开暴露——它可能在任何回应中泄露这些敏感数据。

训练模型以确保安全性的问题是一个独立的研究领域。它甚至有自己的学派。我在这篇文章中提到它，是因为AI开发者必须意识到使用未经验证的模型的风险，但对这个话题的充分探讨将需要一篇完整的文章（或一本书）。

性能和拒绝服务攻击

最后，我想谈谈一个与LLMs不可信这一事实无关的安全问题。与您基础设施中的大多数部分相比，LLMs是*slow*响应的。即使是快速模型，也可能需要数秒钟甚至数分钟才能完成生成响应。这就是为什么LLM应用程序会流式传输响应——等待整个

事情会是一个不愉快的用户体验。

这意味着LLM应用程序特别容易受到拒绝服务攻击。LLM需要运行的GPU是有限的资源。因此，您可以通过相对较少的并发请求使许多AI应用程序瘫痪。如果您正在开发这些应用程序，必须小心对待故意滥用（即谨慎控制免费层）和意外滥用（即不要允许用户运行多个并发会话）。

如果可以，请确保限制提示词和回复中的 token 数量。无意中允许用户生成庞大的回复，可能在资金和可用性方面都代价高昂。

总结

- LLMs 是不可预测的，有时会表现出恶意行为，这意味着你必须像对待用户输入一样对待它们的输出。
- 提示注入是真实的且无法解决的
- LLM 工具必须像用户面对面的 API 一样进行访问控制。绝不能将一个不应暴露给当前用户的工具暴露给 LLM。

- MCP 服务器实际上是远程库。你信任的是服务器拥有者，而不仅仅是接口。
- 即使单独使用，如果你让大语言模型（LLMs）也可能做出愚蠢或鲁莽的事情。
- 如果你正在使用自己训练的模型，你就承担了另外一系列的风险。
- 拒绝服务攻击既简单又廉价。限制并发会话、令牌长度和工具访问

Vibe 编码

在过去几个月里，利用大型语言模型（LLM）为你构建整个程序的做法（通过Cursor、Copilot，或直接请求ChatGPT）被称为“vibe coding”。这个方法非常有效。对于那些在过去几年里使用LLM的人来说，这并不令人惊讶（尽管对于GPT-3.5之前的工程师来说，这将是一个类似AGI的突破）。这种做法有两个大且众所周知的问题。

氛围编码的明显问题

vibe coding 的第一个重大问题是，在某个时刻你会碰到代码库规模的上限，之后就无法再添加新的功能。如果你的代码库足够模块化，可以把这个问题推迟一段时间，但最终你会到达这样一个点：合理的代码变更所需的上下文信息超过了大型语言模型所能容纳的范围。

另一个大问题是你没有编写代码，因此你不理解它。如果你需要回答关于代码如何工作的一个问题，或者自己在其上构建一个功能，你必须花费大量时间去弄明白。换句话说：

代码不是最有价值的产物。***Your understanding of the codebase*** 才是。

LLM 可以以极低的成本生成大量代码。但它无法自动补全你对代码库的理解——这部分必须由你自己完成——而且当代码库的规模超过上下文大小时，它自身的理解能力就会碰到硬性上限。到某个时候，你终究需要亲自深入并理解代码。

为理解而重构

在过去的几年里，我发起的一个项目从一个四人的“虚拟团队”扩展成了一个正规的工程组织：三

是时候增派工程师、多个经理和跳级管理人员，等等。在那段时间里，我做出的最佳决策之一就是热情地接受几乎任何提议的重构。即使我个人认为这个重构并没有带来太多价值（或者甚至认为它会稍微让事情变得更糟），我仍然支持它。这背后有几个原因：

1. 对项目新来的工程师提出重构建议的工程师是充满热情的工程师，我希望鼓励他们，而不是让他们气馁 2. “新工程师”对什么是可读或“简单”代码的看法可能比我更准确，因此优先考虑这种看法是可以的 3. 让工程师重新编写代码库的一部分是让他们快速了解代码库的最简单方法

第三个原因是最重要的。我再说一遍：如果你想让某人熟悉一个新的代码库，让他们重写其中的一部分。他们会从这个过程中学到很多，但关键是他们会立刻成为他们重写部分的领域专家。通过一些重构，你可以从一个只有你是唯一依赖的工程师的情况，变成一个多个团队工程师都能承担责任的情况。这是管理一个大型代码库的唯一可持续方式。

总结

Vibe 编程在一定程度上非常有效。当你达到那个点时，你必须进行自己的“重构以理解”，才能对代码库进行任何进一步的有用修改。随着大语言模型（LLMs）能力的提升，这个点可能会越来越远，但它始终会存在。

有效地使用 LLM 并不只是提示的问题

当人们谈论如何有效地使用语言模型时，他们主要谈论的是提示：分享优秀的提示语、提示的技巧清单，或者成为“提示工程师”的课程。确实，提示是一种出乎意料的有效方式，可以从大型语言模型（LLM）中获取更多信息。提示的微小变化可以对LLM的输出产生巨大影响。确实存在一些通用规则（例如，将问题放在前面，将背景放在最后）。然而，当我使用LLM时，我很少考虑提示。

使用大型语言模型（LLMs）得当涉及从它们擅长的领域中提取价值，并避免在它们不擅长的领域浪费时间。

最重要的是要对强大的 LLM 的优势和劣势有清晰的认识。这比听起来要难，原因有几个：

1. 与大多数工具不同，LLMs 会主动试图欺骗你关于它们的能力 2. LLMs 在一些对人类来说很难的任务中表现得非常好（例如广泛召回），而在一些对人类来说很简单的任务中表现得非常差（例如拼写） 3. LLMs 擅长或不擅长的任务会变化 *rapidly*

要很好地把握 LLM 的优势和劣势，有两种方法。第一种是从技术层面理解它们是如何工作的，这样你的直觉就有依据。第二种方法就是花大量时间使用 LLM，通过反复试错来体会。实际上这两种方法都必须结合使用，因为 LLM 有时甚至会让最了解它们的工程师感到意外，而且它们变化得如此之快，仅凭经验本身也可能具有误导性。

具体建议

好的，这是一些具体的建议（与提示无关）：

在学习新主题时，要充分利用聊天的特点。如果你把大型语言模型当作更好的谷歌搜索来使用，那么你只是在捕捉它们价值的一小部分。我最好的体验是在提问跟进问题时——只是直接、简单、天真的问题，例如“所以你是说X是真的”或“这会有Y的后果吗”？

信任广泛的理论性陈述，但对偶然的细节保持怀疑。根据我的经验，LLMs 在解释诸如“天空为什么是蓝色的”或“OAuth 是如何工作的”这样的问题时非常出色。但它们偶尔会搞错细节（例如日期、姓名、引用的论文）。如果你有一个细节密集的任务，LLMs 可能不是最佳选择。

如果LLM陷入循环或似乎感到困惑，请退出。你无法拯救它——至少在没有足够理解问题的情况下，无法像喂饭一样逐步给出答案，这时你倒不如根本不使用LLM。

与此相关的是，跨多个大语言模型提问困难问题。与其在单一对话中花费十五分钟反复推敲，不如将初始问题问四五个最先进的模型。有时它们中的一个会立即给出一个好的答案。

如果你在提问（而不是试图学习一个新话题），而大型语言模型给出了一个很长的答案并包含大量的前言，可以快速浏览前言。这些前言更多是针对大型语言模型的，而不是针对你的——你可以跳到最后看答案，然后如果需要再回头阅读。

充分利用这样一个事实：一次性代码实际上几乎可以零成本生成。如果你对某个主题感兴趣（例如，GitHub 仓库中有多少次提交

触摸多个文件），您可以生成一个快速脚本立即回答这个问题。它将十分钟的问题转化为一分钟的问题，让您可以提问十倍的问题。

不要害怕编写使用 LLM API 的脚本。你可能能够编写自己的代码来解决上述 git 问题，但编写代码来回答“有多少次提交使用了正确的标点符号”要难得多。使用 LLM，你可以轻松编写脚本来回答这个问题（或任何其他难题）。现在有许多便宜或免费的 LLM API 选项可供选择：例如，Google 和 GitHub 都提供慷慨的免费套餐。

总结

- 尽量了解哪些模型擅长，哪些模型不擅长
- 提出后续问题
- 信任理论，对细节持怀疑态度
- 如果LLM似乎困惑，完全退出
- 在多个大型语言模型（LLM）中提出严肃问题
- 生成一次性代码
- 生成一次性代码 *that uses LLM APIs*

提示不值得关注吗？那是夸大其词了。它仍然值得仔细思考如何与模型交互。特别是推理模型会奖励仔细的提示。如果你正在编写使用LLM的代码（而不仅仅是通过聊天与它们交互），你应该花很多精力来设计一个好的提示。但如果你正在使用LLM来回答问题、学习主题，或出于任何其他个人原因，你应该更关注了解LLM能做什么和不能做什么。

为了避免被大型语言模型取代，做它们做不到的事

当软件工程师，这是一个奇怪的时代。大型语言模型非常擅长编写代码，而且能力正在迅速提升。目前正有多项耗资数十亿美元的尝试，旨在开发一个纯 AI 的软件工程师。大致的策略——将一个推理模型与工具放入一个循环中——众所周知，并且（在我看来）似乎很可能奏效。我们软件工程师应该做些什么来为即将到来的变化做好准备？

在短期内，学习 AI 并获得晋升 d

有一些明显的短期答案：

- 从AI工具中获得你能得到的优势
- 了解语言模型背后的技术原理，以便您能够参与日益增长的人工智能工作量
- 获取状态，因为似乎更初级的角色将首先被替代。

没人真正知道“短期”会持续多久。我不认为在未来五年内，AI 软件工程师会取代大量工作岗位——大型企业的行动速度没那么快。一家大公司必须率先在这项新技术上承担风险，而在几年过去且没有发生灾难之后，其他公司才会跟进。如果技术问题比看起来更难，或者 AI 的进展停滞，这个时间可能会更长。

在中期，积极利用遗留代码

那中期怎么样？LLMs 最后可能发展哪些软件工程技能？回答这个问题的一种方法是思考它们可能先发展哪些技能 *first*。目前，LLMs 能够展现的最令人印象深刻的编程成就是在竞赛编程中的卓越表现。这项工作的显著特点是：

- 在技术上是困难的（在某种程度上，在数学上也很困难）
- 问题定义清晰，范围明确
- 解决方案是显然可验证的
- 所涉及的代码总量非常低

那么，哪一种编程工作会与此相反呢？

- 问题定义不清，范围模糊
- 解决方案难以验证
- 涉及的代码总量是巨大的

在我看来，这是在描述 *legacy code*：在大型已建立的代码库中进行特性开发。将需求转化为这些代码库中的所需更改是困难的。由于特性交互的组合爆炸式增长，要确信自己没有引入另一个 bug 更加困难。而且你必须阅读和编写的代码量巨大：数百万或数千万行。

我认为大型语言模型最终能够完成这类工作。但这需要一段时间，原因有几个。首先，它需要更好的大上下文问题解决方案：要么显著改善RAG，要么找到一种快速有效的方法来拥有一个数百万标记的上下文窗口。如果我们非常幸运，这个问题可能会证明是无法解决的。第二，编写一个真正好的评估来处理遗留代码的调整非常困难。当前的软件工程评估范围相对较小。第三，相关数据分布在许多非常私密的孤岛中。Facebook或Google可以在自己的内部PR或变更请求上进行训练，但没有任何一个AI实验室能访问多个公司的代码库和PR/JIRA堆栈。

所以，如果你目前在 LeetCode 和明确定义的困难技术问题上非常强，考虑花更多时间在你公司中的单体代码库上。能够做好这种工作的工程师，可能会因此延长五到十年的职业生涯。

从长远来看，承担责任

有一张1979年IBM的著名幻灯片，上面写着：“计算机永远无法被追究责任 / 因此，计算机永远不能做管理决策。”

这里的关键思想是，管理不仅仅是做出正确的决策。这是关于对你做出的决策负责，不论是好是坏。工程也同样如此——根据我的经验，随着职务的晋升，这一点尤为重要。工程师不仅仅是一个

写出优秀代码。他们是 *trusted*：具体来说，是那些非技术高管可以信任，回答技术问题并交付技术项目的人。

并非所有工程师都是这样。有些工程师把自己的角色看作是把 JIRA 工单变成可运行的代码，或向客户交付客户喜欢的功能。在我看来，一个合理地足够强的 LLM（当前能力的 2 倍或 3 倍）就能完成这些任务。但是，一个强到足以承担责任——也就是说，能够作出承诺并获得管理层信任的——LLM，必须比一名优秀的工程师强大得多得多。

为什么？因为大型语言模型（LLM）没有风险承担，这意味着正常的信任机制无法适用。高管信任工程师，因为他们知道如果工程师犯错，工程师将会面临不愉快的后果。由于工程师在承担某些风险（例如下一个奖金、晋升，或在极端情况下被解雇），高管可以相信他们的承诺的力量。大型语言模型没有什么可以承担的风险，因此信任必须完全建立在它们的过往记录上，这更加困难且需要更多时间。

从长远来看，当几乎每一位工程师都被 LLM 取代时，所有公司仍然至少会留下一名工程师，负责看护这些 LLM，并将它们的承诺和计划“洗白”为人类可读的承诺。也许如果 LLM 足够优秀，这名工程师最终也会被取代。但他们将是最后离开的。

不要以为你可以等过人工智能泡沫

在19世纪中期，美国对铁路产生了疯狂的热情。在五年的时间里，建设了超过三万英里的铁路。这一切大多是通过消费者对铁路公司的投资狂潮资助的，铁路公司被认为是一个安全且有利可图的投资。1873年，泡沫破裂。成千上万的美国人失去了他们的积蓄，大约三分之一的铁路公司破产了。但铁路线路并没有消失。它们被那些幸存下来的铁路公司以低价收购，在接下来的百年里，这些铁路运输了大量的列车。

除此之外，所有修建的铁路线都有一条配套的电报线（用于信号传递和调度）。当铁路狂潮结束时，电报狂潮才刚刚开始。这项旨在协调列车的技术最终完全改变了贸易、金融市场、战争，以及在其最新形态下，作为电话和互联网，人类的沟通方式。

换句话说，泡沫有来有去，但资本投资会留下来。最近，加密货币泡沫的破裂为人工智能热潮铺平了道路——突然间，数据中心里有大量便宜的GPU，静静地等着被用在某些地方。很少有人看出这个联系。假设现在的人工智能泡沫破裂了，那留下来的物理基础设施将会是什么，如何利用这些资源?

我们可能会有大量的GPU。不是消费级的游戏GPU，而是重型的H100和B100，设计用于将巨大的模型权重集存储在内存中，并以巨大的并行度提供LLM补全。如果我们不把它们用于AI，我们会用它们做什么？也许是模拟和建模，或者是与AI相关的领域，如蛋白质折叠或药物发现？可能有许多领域有一些被认为GPU成本过高的用例。这些用例可能会变得出乎意料的可行。

抱歉，您提供的源文本“ble”没有足够的上下文或信息，无法进行翻译。如果您能提

我们会回归加密货币吗？如果巨型数据中心将它们的所有GPU都卖掉，也许会，但我并不信服。我不认为微软和XAI之类的公司会进入比特币挖矿，也不认为有任何真正的加密货币应用场景能从微软拥有的数据中心中的成千上万的GPU中获益。为了明确一点：加密货币的主要应用场景是无需信任的协调，但超大规模公司不需要无需信任的通信。它们已经是可信的！我想我们*could*会看到某个非常不可能的事件，比如XAI将它们的闲置GPU用在对某个不幸的加密货币进行51%的工作量证明攻击上，但很难想象这种情况会有正向的预期收益。

把注意力放在 GPU 上可能抓错了重点。对于真正的系统思考者来说，GPU 只是一个实现细节。在这场大型 AI 扩张中，真正起作用的资源是能源：字面意义上的电力，而这正在以空前的规模建设。公司正在重新投资核能，并投资于具有投机性的裂变技术。如果那真的起飞，它最终可能会成为 GPU 这条铁路所对应的电报：一种伴生技术，最终对世界产生同等甚至更大的影响。

当然，人工智能泡沫破裂的真正赢家可能是……人工智能。当铁路泡沫破裂时，铁路依然存在，并且在两百年后仍然是我们世界的核心部分。这里最可能的结果是人工智能泡沫破裂，人工智能的发展在没有过多炒作的情况下安静继续，并且无论如何会占据全球GDP的相当一部分。

结论

我不知道我是否对今天应该如何进行软件工程的工作有正确的理解。但无论正确答案是什么，我确实知道，它与2010年代的工作方式不同。

回到2015年会很不错，那时候技术工作很容易找到，企业会把钱投入到有趣的技术问题上，即使这些问题没有商业价值。90年代的外包恐慌已经结束，而2020年代的AI恐慌还没有开始。

另一方面，现在从事软件工程师从未像现在这样令人兴奋。如今的项目与收入联系更紧密，也不那么虚假。你所做的工程决策有着实实在在的利害关系。以及 *nobody knows* 当前的 AI 进展将如何发展。