2026年第3周 · 工程

工程技术：在智能体优先的世界中构建 Codex

作者：Ryan Lopopolo，技术人员

作者 Ryan Lopopolo · 1月30日 · 分享

OpenAI 团队在过去五个月内尝试了一项实验：从一个空代码库开始，使用 Codex 来构建、发布和内部使用一个全新的软件产品。该项目从一开始就完全由 Codex 模型驱动——我们刻意选择不编写任何手动代码作为基础。这种选择有助于明确当前前沿人工智能模型在实际工程工作中能发挥哪些作用。

这次实验有别于我们之前对代码代理、长一序列、代码生成模型能够实现哪些功能的探索。我们将这项工作建构于当时最新前沿模型之上，并对其实施了一系列日益复杂的任务。其中一项早期任务是编写一个能接收并回答请求、并通过 OpenAI 平台 Responses API 实现流式传输响应的聊天应用程序。这项任务非常适合模型发挥优势的领域，因为其结果具有明确的预期特性，且易于进行测试和验证。我们要求模型实现产品功能，并通过指导模型分解目标、将高层设计原则转化为可执行代码来实现这一目标。

在这个项目中，我们学到的最重要经验之一是：在工程资源投入方面，当前最有价值的投入通常是指明模型实现目标所需具备的能力。对于这类工程工作，通常需要将一个宏大目标拆解为更小的构成要素——设计、编码、评审、测试、发布运维，并赋予智能体实现这些目标所需的工具、抽象概念和反馈机制。这种方法不仅能加速我们的工作进度，其效果也清晰地表明：智能体能够承担工程工作中通常由人类完成的大部分工作。

本文将探讨在这种新工程模式下，工程团队的职责如何发生变化，以及我们目前在为这个世界构建过程中学到的经验。我们选择从一个观点出发：人类工程师始终是这个循环的一部分，但其工作内容发生了变化。例如，我们要求工程师将反馈内容固化为文档、教程或可执行代码，以实现对智能体而言清晰可见、可复现且可执行的工作。

工程工作的重新定义

自项目开始以来，我们共生成了约五个月的代码，这一时间大致相当于工程师采用早期 AI 辅助工具进行开发所需的时间。整个项目期间，平均每天有约三名工程师在对代码库进行提交。在模型吞吐量方面，我们观察到每个工程师平均每天提交约 3.5 个合并请求。随着团队规模扩大，这一数字仍在持续提升，这表明随着我们对工作流程的理解不断加深，工程效率并未因代码库规模扩大而出现下降。

这个项目中最令人意外的部分，或许是模型能够执行哪些工作才能推进剩余工作内容的变化。在项目初期，模型能力表现出明显的边界，其表现高度依赖于明确定义的环境和清晰指定的意图。随着任务推进，我们要求模型执行更为开放性的任务，例如：模块重构、修复遗留实现方案中的缺陷、为新功能制定实现方案、对代码库特定区域进行安全性审查、或通过实现新特性来扩展系统。模型通常能够顺利完成这类任务，其成果具有合理的设计思路，并能很好地适应现有代码库的模式与约束条件。

人类掌控，智能体执行

我们最初设定的工程目标，是在不编写任何手动代码的前提下，完成一个内部测试版产品的全部工作流程：从代码库开始为空的状态开始，通过智能体生成代码，并由人类工程师通过提交拉取请求（Pull Request）来实现代码合并。我们将产品构建视为学习如何利用平台能力来赋能智能体发挥价值的过程。这种能力目前尚不明确：例如，一个工程师能否在一个代码库中安全地发布多个合并请求？模型在面对它从未接触过的代码时，其表现如何？当代码变更发生冲突时，应采取哪些恢复策略？在代码库内部，哪些工程实践能够发挥最大效益？这些问题在项目伊始时并不明朗，但我们预期，对于任何要实现规模化应用人工智能技术的工程团队，这些问题都具有重要意义。

我们决定采用一种结构化方法来推进模型的工作：将复杂的大型目标拆解为更小的构成模块（设计、编码、评审、测试、发布运维），并要求模型能够以端到端的方式实现这些模块。这种结构化方法有助于我们明确工作内容的优先级，例如：哪类模块对模型而言具有挑战性，因此需要优先投入资源以实现其稳定可用；哪些工作流程目前对模型而言尚属陌生，因此需要通过示例和明确的验收标准来进行定义。在模型能力与我们期望之间存在差距时，我们会暂停新功能的开发，转而聚焦于提升模型在该领域的表现能力。

代理优先的世界中的工程技能

在智能体成为主要开发模式的项目中，工程技能的重心在于确保系统具备智能体能够独立推进工作所需的能力，这与以人类开发者为主导的项目有所不同。例如，当模型能够独立实现一项功能并发起拉取请求（Pull Request）时，我们会将其视为一种信号，表明对于该模型而言，当前的任务定义、反馈机制和验证流程已足够清晰且具有可验证性。我们的目标是持续提升这些能力，以扩大模型能够可靠执行工作的范围。这种方法要求工程师投入时间观察模型尝试开展哪类工作、在哪些环节上遇到困难，并随后调整仓库环境，以使后续任务对模型而言更具可实现性。这种调整通常包括：明确模块的所有权边界、简化测试搭建流程、或将一个新的生产环境问题固化为一项可回归测试的任务。

仓库是智能体的可读知识库

在传统工程实践中，架构文档通常会随着时间推移而逐渐失效，因为人类工程师能够通过代码评审和口头沟通来保持团队对系统设计的共同理解。对于一个以智能体为主要开发者的项目，这类文档的容忍度要低得多：如果仓库内关于系统结构、模块所有权或设计原则的信息不再成立，模型的表现很可能会开始出现偏差，其影响会随着时间推移而不断累积。因此，我们采用了一种方法，将仓库内的结构化知识视为系统的一部分，并要求对其进行版本控制、保持更新，并通过与代码本身相同的评审流程来进行维护。这使得仓库能够持续保持可读性，并确保模型能够基于当前实现方式进行推理，而不是依赖过时的历史惯例。

关于如何组织这类知识，我们尝试过多种方式，以确定哪种结构能够实现模型的有效推理。我们发现，将一个大型指令文件（例如 AGENTS.md）用于描述所有流程，通常会因以下原因而失效：上下文是一项稀缺资源，过长的指令会挤占模型需要关注的任务、代码或文档的空间；当所有内容都被标记为“重要”时，模型将开始在局部模式匹配上进行优化，而非采用有意的导航方式；单一文件会随时间推移而腐败，因为代理无法区分哪些内容仍具有时效性；且该文件难以进行机械验证，从而降低了工程师为保持其准确性而投入时间的意愿。

最终，我们采用了一种结构化方案，将仓库内的知识组织在多个顶层目录下，以保证内容具有明确的范围和生命周期。AGENTS.md 文件现在仅用于简要说明如何定位相关信息，而模型则会被指引去查阅特定目录下的版本控制文件，以了解关于当前任务的更详细上下文。我们要求工程师对代码评审流程进行调整，以确保知识库的更新与代码变更保持同步；在合并请求中，我们会评估文档更新是否与行为变更一样，具有必要的合理性。这种方式使我们能够将大部分工程工作重新聚焦于：为模型提供清晰、有版本控制、且易于验证的上下文信息，而不是编写单一指令文件来覆盖所有特殊情况。

应用可读性

一旦复杂任务成为模型的主要工作内容，代码库的可读性就成为保证其后续开发质量的关键因素。在传统工程实践中，代码评审通常以人类工程师作为主要阅读对象，其重点在于确保变更内容能够被团队成员理解，并维持系统的整体一致性。在这种模式下，随着代码库日益成熟，单个功能的实现成本会逐渐降低，因为模型能够根据既有模式进行推理并生成符合规范的代码。我们发现，在智能体优先的工作流程中，这种趋势会发生转变，因为代码库的可读性——即模型能够顺利地对其进行推理、变更并扩展的程度——将成为我们优先关注的核心目标，以确保随着代码库规模的扩大，模型的生产效率不会因其对自身工作成果理解难度的增加而受到限制。

应用可读性的技术栈

我们观察到模型在 Codex 运行任务期间，会利用一系列工具和能力来执行任务：这包括启动本地开发环境、采用 Chrome DevTools 协议实现对 DOM 的直接检查与操作、查询日志与指标数据、以及通过浏览器自动化工具模拟用户交互。这种能力使模型能够复现问题、验证修复方案、并对其实现效果进行推理——这一过程与人类工程师在处理生产环境问题时采用的方法高度相似。

代理视角下的可观测性

为了使模型能够以端到端方式推进工作，我们把可观测性功能（日志、指标、链路追踪）直接集成进了开发环境。对于人类工程师而言，这意味着调试工具可以作为本地开发环境的一部分来提供；对于模型而言，这意味着遥测数据对其可用，且指令内容明确指出了如何查询这些信息以验证特定任务的运行时特性。例如，一个模型可以运行一个测试用例，通过 Chrome DevTools 协议采集性能指标，随后利用日志来判断关键路径上哪些操作耗时较长，并通过重新部署并重复该流程来验证后续变更的影响。我们发现，这种紧密的反馈回路极大地提高了模型生成内容的质量，因为它赋予模型对其所做变更的实际运行时效果具有可验证的认知。

分层架构

随着代码库规模的扩大，单纯依靠人类工程师来把握整个系统的设计原则，已不足以保证模型保持一致性。我们发现，采用一种具有明确依赖关系、且机械可验证的分层架构，是推进大型基于 Codex 项目时最有效的方法之一。这种分层结构使我们能够定义模块可以依赖哪些其他模块的规则，并通过自动化检查来强制执行这些边界。例如，每个业务模块可能被要求采用一组固定的层级：类型（Types）、配置（Config）、数据仓库（Repo）、服务（Service）、运行时（Runtime）、对外接口（Providers）。在此基础上，我们允许一条单一的、经过精心设计的接口层级通过代理子模块（Agent）、飞书（Feishu）、监控（Metrics）、特性开关（Flags）或实验功能（Exp）等横切关注点进行注入。这种模式通常适用于大型工程组织中成熟的微服务架构，其优势在于，一旦模块边界被清晰定义，大多数工程决策便可在局部层面做出，因为模块内部的变更只要不违反其对外接口的约定，通常不会对系统其他部分产生不利影响。

这种结构通常要等到团队规模达到足够规模时才会推广，对于编码智能体来说，这是一个有趣的机会：除了代码，还需要为工程师、编码代理赋能。

在实践中，我们通过自定义结构测试来强制执行这种分层结构，这类测试会对整个仓库进行静态检查。例如，我们通过自定义 linter 脚本来强制执行依赖关系合法性检查，并要求所有自定义 linter 脚本输出具有教学意义的报错信息，以确保当模型发生违规时，能将修复指令注入到模型的上下文环境中。这种方法将原本通常需要手动执行的架构评审工作，转变为一条自动化规则，从而能够在不牺牲速度的前提下，保证代码库保持结构一致性。

吞吐量改变了工程的理念

随着 Codex 项目的推进，对于单个任务所投入的时间成本通常较低，因为我们的工程团队更倾向于以多个任务并行的方式来推进项目，而模型执行单个任务的耗时则可以忽略不计。这种吞吐量的提升带来了一系列新问题，例如：在代码库变更速度极快的情况下，如何保持一致的工程质量？答案通常取决于建立合理的护栏机制，以确保新增代码能够顺利地与现有代码库实现集成，而不会因局部优化而导致整体质量下降。典型的工程实践包括：对测试覆盖率设定明确要求、实施分层架构检查、以及建立一套稳定的重构流程，用于修复技术债务，而这些技术债务通常会随着代理生成大量代码而开始积累。

合并等待成本较低

在吞吐量极高的工程模式中，我们发现一项原本在其他情况下成本较高的决策：即对大多数问题采用后续修复（follow-up）而非阻塞合并（block merge）的方式。在一个模型生成代码的流程中，大多数错误通常会在后续任务开始执行时被发现，并可以通过一个小型修复任务来解决，其成本远低于阻塞一条高吞吐量流水线所带来的成本。这种方法成立的前提是：模型的失败通常是局部性的，且可以通过后续变更进行纠正，而测试失败的成本则会随着后续任务继续推进而不断累积，因此通常需要优先进行排查。在我们的工作流程中，模型生成的代码通过合并后，仍会接受异步的代码评审，这些评审会对代码库进行更广泛的一致性检查，并将发现的问题作为后续任务进行处理，有时甚至会为此创建新的任务来实现自动化修复。这种方法与大型工程组织中处理大规模遗留代码重构工作的方式一致，在这类场景中，技术债通常会被分散到多个工作日中逐步偿还，而不会阻塞日常开发工作的推进。

不断提高的自主水平

随着智能体能力的不断提升，我们预期工程工作的大部分内容将以这种模式进行组织：人类工程师设定目标并确保优先级，而大量具体执行工作则由模型以端到端的方式完成。这种模式对工程实践的要求，与在代理出现之前，为了实现高效协作而优化的工程实践，在很多方面是一致的：清晰的目标、明确的接口、稳定的测试、以及可机械执行的规则，这些规则共同构成了模型能够高效运作的基础。我们目前尚不清楚的是，随着模型能力不断提升，这种结构在哪些方面仍会保持有效，我们会通过持续实验来学习这一点。对于构建与模型同步演进的工程团队，这项工作本身可能会成为一项高价值的投资。

编写验证规范

规范是实现智能体对齐的关键杠桿。Codex 要求我们对高层目标进行拆解，并将这些目标转化为模型可执行的工作单元。在早期阶段，模型常常会因指令过于宽泛而偏离预期目标，例如：它可能会尝试实现一项内容不完整或未经充分测试的功能，或者在面对多个合理方案时做出与我们期望不一致的设计选择。为提高模型输出结果的可预测性，我们开始编写更结构化的规范文档，用于明确界定新功能应具有的行为表现、边界条件以及验收标准。模型随后会以这些规范为输入，生成实现方案并直接在代码中实现。即使是对模型而言，这项工作仍具有价值，因为它能减少模型在意图理解方面所需做出的假设数量，从而提高首次输出结果符合团队预期的概率。

我们加强仓库内知识体系的结构化治理。随着智能体能力的提升，我们发现仓库内大量指导模型工作的知识都可以被编码进来。这包括编码规范、架构模式、产品原则、以及关于特定功能应如何实现的具体示例。对于人类工程师而言，这些信息通常会分散在聊天记录、评审意见或个人经验中；而对于一个以智能体为主要开发者的项目，这类信息必须具备版本控制、易于发现、且结构清晰，才能成为有效的上下文信息。我们采用了一种方法，将仓库内的知识视为结构化目录的一部分，并要求新增功能或重构工作提交更新内容，以保证模型能够获取到关于系统当前运行状态的最新指导。这种方法使得文档本身成为工程流程的一部分，并通过与代码相同的机制来保持其时效性。

代理可观测性对应用启动时间高度敏感。随着单个任务的执行时间因模型能力提升而不断缩短，我们发现应用启动时间成为了一个新的瓶颈。例如，在模型能够并行运行多个服务的情况下，每个服务都需要自己的启动时间，这会显著拖慢反馈回路的速度。我们通过优化服务启动流程，以实现本地开发环境的高效运行，这项工作本身也成为了一项高价值的投资，因为它使模型能够在不牺牲生产环境稳定性的前提下，采用更为激进的测试策略。

代理协调模式仍处于早期阶段。目前，我们尚未对多个模型协同工作以推进单个功能的最佳实践形成定论。但初步证据表明，将专门的任务分配给具有明确角色定位的模型（例如：一个模型实现功能，另一个模型对其进行安全性审查，第三个模型编写测试用例），通常有助于提升整体系统的可靠性，因为这种分工模式能够在各个环节引入不同的关注点和检查机制。我们目前正在开展进一步实验，以确定在当前模型能力水平下，这种模式是否具有普遍适用性。

我们的学习内容

回顾项目以来，这一情况与 Scaling AI 的预测结果相吻合：随着时间推移，瓶颈将从模型能力转移到工程基础设施、反馈机制和信息结构方面。在为智能体构建高效工程流程的过程中，我们发现了多个高杠桿投资方向，这些投资共同提升了模型输出结果的可靠性与实用价值，其中包括：为代码库建立清晰的结构与验证规范；为模型提供结构化的调试工具；以及随着代码库规模的扩大，持续优化仓库的组织方式与可读性。这些投资的回报方式与在一个以人类工程师为主要受益对象的代码库中进行投资时大体相同，但随着模型能力的持续提升，这些回报的复利效应预计将会更为显著，因为模型能够承担越来越多原本需要人类工程师手动执行的工作。

结语

Codex 项目表明，在一个以智能体为主要开发者的工程团队中，软件工程的核心工作内容仍然是：设计环境、明确意图、并构建反馈回路，以使模型能够可靠地为我们的利益而工作。随着模型能力的持续提升，这种工程基础设施的复利效应预计将会随着代码库内容的不断积累而加速发展。目前尚不清楚的是，模型能力的哪些方面最有可能率先发生变化，以使人类工程师的工作重心发生新的调整，但这一点在当前这种模式下仍具有很高的可预测性。我们将继续探索这种模式如何随模型能力的提升而演进，并期待学习随着模型能力边界的不断拓展，哪些工程实践仍保持有效的结论。

致谢

本项目由 Jeff Tarakajian、Alex Kotchnev、Bryan Gin-ge、Anna Tarabanova、Yiming Qian、Shasank Prasad、Sandy Mahendra、Joseph Trasatti、Lucian Wischik、Alex Hancock、Gregory Whitaker、Anuj Gupta、Tyler Mcdonald、Will Aitchison、Lars Howe、Shaun Maguire、Gabe Moothart、Justin Young、Eric Mitchell、Lisa Jaffe、PJ Hagerty、Sherwin Wu、Shikhar Huddar 和 Alex Huettner 的贡献。

脚注

1. 在 Codex 的早期实验中，模型采用结构化文件格式来记录其计划与进度，例如一个用于跟踪已完成工作内容与剩余工作内容的 PLANS.md 文件。随着模型能力的提升，这种工作流程已变得不再必要，模型现在可以直接从进行中的工作内容出发，并将其推进至完成状态。

相关阅读

- 工具调用（Function calling）
- 流式传输（Streaming）
- 长时运行任务（Long-running tasks）
- 代理工作流（Agent workflows）
- 代理工具（Agent tools）
- 评估（Evals）
- 可观测性（Observability）
- 模型上下文协议（Model Context Protocol）
- 代理安全性（Agent safety）
- 模型上下文协议（Model Context Protocol）
- 代理安全性（Agent safety）
- 评估（Evals）
- 可观测性（Observability）
- 工具调用（Function calling）
- 流式传输（Streaming）
- 长时运行任务（Long-running tasks）
- 代理工作流（Agent workflows）
- 代理工具（Agent tools）