Vibe Coding Kills Open Source*

Miklós Koren† Gábor Békés‡ Julian Hinz§ Aaron Lohmann¶

September 23, 2026

Abstract

Generative AI is changing how software is produced and used. In vibe coding, an AI agent selects and assembles open-source software (OSS), often without users directly reading documentation, reporting bugs, or otherwise engaging with maintainers. We develop a model with endogenous entry and heterogeneous project quality. AI-mediated use raises productivity but weakens the direct engagement through which many maintainers earn returns. When OSS is monetized only through direct engagement, higher vibe-coding adoption lowers sharing, entry, availability, quality, and welfare despite higher productivity. To measure the package-choice margin, we give earlier and later frontier coding models the same technology-neutral product-requirements briefs for 500 prominent websites. We record each direct package choice in the generated `package.json`, then compare weekly npm downloads and GitHub-star gains across the resulting pair-specific package groups. In 498 GPT and 97 Claude matched-brief pairs, model versions select different dependencies for identical requirements. Packages more often named by the later model grow faster in npm downloads in both comparisons, while GitHub-star gains fall across all package groups. The comparison is descriptive, not random exposure: downloads mix human and automated installations, and falling stars suggest reduced human interaction but do not causally identify it. These patterns are consistent with the model's demand-diversion channel.

JEL Codes: O33, L86, D85

*We thank Péter Mátray, Mike Linksvayer, Gergely Orosz, Eduardo Arino de la Rubia and Kevin Xu for helpful comments. We thank participants at presentations at GitHub, Ericcson, Automattic. This project was funded by the European Research Council (ERC Advanced Grant agreement number 101097789). The views expressed in this research are those of the authors and do not necessarily reflect the official view of the European Union or the European Research Council. Project no. 144193 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the KKP_22 funding scheme. Békés thanks NRDIF for financial support under OTKA Project #147121 .

†Central European University, KRTK, CEPR and CESifo. *Corresponding author:* miklos@koren.work

‡Central European University, KRTK, and CEPR.

§Bielefeld University and Kiel Institute for the World Economy.

¶Kiel Institute for the World Economy.

Keywords: Open Source Software, Artificial Intelligence, Network Externalities, Nested Logit, Software Economics

1 Introduction

Generative AI is reshaping software development. AI coding assistants such as Claude Code, Cursor, and Lovable let users translate intent into working applications with little or no manual coding. This AI-mediated mode of building software is often called "vibe coding." Vibe coding reduces the cost of producing software, but it also changes how users interact with the software ecosystem. Traditionally, a developer selects packages, reads documentation, and interacts with maintainers and other users. Under vibe coding, an AI agent can select, compose, and modify packages end-to-end, and the human developer may not know which upstream components were used.

This shift raises a general equilibrium question about the sustainability of open source software (OSS). OSS is a nonrival input into producing more software, and it generates large social value relative to its direct production cost (Hoffmann et al., 2024). Yet many OSS projects rely on visibility and engagement from direct users—documentation visits, bug reports, public Q&A, and reputation—to sustain maintenance and capture private returns (Lerner and Tirole, 2002; Jones, 2020). If AI mediation substitutes for direct interaction, then the technology that makes software easier to use may simultaneously erode the engagement-based channel that funds and motivates its supply. We ask whether the productivity gains from vibe coding outweigh the loss of appropriable demand once developer entry and selection respond.

We compare the direct package choices of earlier and later frontier coding models on matched website briefs with fixed functional requirements. GPT-4o and GPT-5.2-Codex, as well as Claude Sonnet 3.7 and Claude Sonnet 4.5, choose different direct dependencies for the same briefs. The resulting pair-specific groups display different npm-download and GitHub-star patterns; in the Claude comparison, downloads rise more for packages more often named by the later model, while average star gains fall in every group. This is descriptive evidence: neither package exposure nor real-world agent adoption is randomized. It is limited to prominent website-building tasks and JavaScript packages; npm downloads mix human and automated installations, and stars are a narrow proxy for attention rather than a measure of maintainer returns, contribution, or long-run entry.

Not all OSS is equally exposed to this mechanism. Corporate-backed projects released for strategic or complementary reasons—such as React (Meta), Go (Google), or Kubernetes (Google/CNCF)—derive private returns primarily from the strategic value of adoption rather than from community engagement. Our model best describes the segment of OSS where individual maintainers and small teams sustain projects through engagement-dependent channels: reputation, consulting leads, documentation traffic, and discovery. This segment accounts for the bulk of OSS projects.

We build a tractable model of the OSS ecosystem in the spirit of monopolistic competition

with endogenous variety (Krugman, 1980) and heterogeneous-project selection (Melitz, 2003). Developers incur an up-front development cost before project quality is realized. They draw quality from a Pareto distribution and then decide whether to release the project as OSS by paying a fixed sharing cost. Users value variety and choose among available packages under a discrete-choice demand system. After selecting a package, users choose whether to interact with it directly or through an AI agent. This nested choice captures that vibe coding is a usage technology that affects both user utility and the degree of engagement that maintainers can monetize.

Vibe coding affects the OSS ecosystem through two channels. The first is a productivity channel: by reducing the effective cost of using a given package, AI raises user utility and lowers the cost of producing new software that builds on existing components. Even before state-of-the-art vibe coding tools, field experiments documented sizable productivity gains from AI coding assistance (Peng et al., 2023; Cui et al., 2025). The second is a demand-diversion channel: when users rely on AI agents rather than direct interaction, maintainers capture less engagement and therefore less private return per unit of usage. These channels interact because entry and sharing are endogenous. Better usage technology raises the value of the ecosystem as an intermediate input and would tend to increase entry. But a contraction in monetizable engagement shrinks the effective market for OSS maintainers and discourages sharing and entry. Since "customer demand motivates the supply of energy that drives software ecosystems" (Jones, 2020), a technology that alters how demand is expressed can reshape supply even if total usage rises.

Our main result is that under traditional OSS business models, where maintainers primarily monetize direct user engagement (higher visibility leading to paid opportunities or other forms of appreciation), higher adoption of vibe coding reduces OSS provision and lowers welfare. In the long-run equilibrium, mediated usage erodes the revenue base that sustains OSS, raises the quality threshold for sharing, and reduces the mass of shared packages. Variety shrinks and the average quality of shared OSS falls, so user utility can decline despite better AI. The decline can be rapid because the same magnification mechanism that amplifies positive shocks to software demand also amplifies negative shocks to monetizable engagement. In other words, feedback loops that once accelerated growth now accelerate contraction. Our new evidence does not identify this full long-run adjustment. It instead provides descriptive short-run patterns in package downloads and GitHub attention across the pair-specific groups.

The key intuition behind the negative channel is that when mediated adoption is responsive to AI capability, direct engagement can fall faster than development costs decline. Utility falls because entry decisions generate a business-stealing externality: when a new package enters, it attracts users away from existing packages, but the entrant does not internalize that this diversion reduces the engagement-based returns that other maintainers rely on.

The magnitude of this externality depends on a single empirical quantity: how responsive users are to improvements in AI capability. When AI tools get better, how quickly do users switch away from direct interaction? The rapid diffusion of AI coding assistants suggests high responsiveness; in the policy section, we discuss how to quantify it using adoption and usage data (Demirer et al., 2025).

We also analyze two extensions. First, we consider a benchmark in which vibe coding is used only by professional software developers and does not directly mediate final-user consumption. In this case, the demand-diversion channel is absent: AI lowers development costs without eroding engagement-based monetization. The equilibrium features higher entry and higher average quality of shared OSS. This benchmark helps interpret short-run dynamics in which AI is primarily a developer productivity tool, while vendors increasingly target non-developer users as well (Anthropic, 2026). Second, we allow for alternative monetization arrangements and ask what is the lowest per-user monetization that sustains OSS entry at its current level. This can be lower than the current level, because the increase in productivity partly offsets the loss of developer reward. But, under plausible parameters, per-user monetization must remain very close to current levels to avoid large declines in OSS provision.

To keep the analysis tied to how OSS is produced and used, we tailor the model to four salient industry features: large fixed costs and nonrivalry, user love of variety, heavy-tailed heterogeneity in project outcomes with endogenous selection into sharing, and software as an intermediate input into producing more software—a "software-begets-software" feedback. These features are quantitatively important: most repositories attract no attention and a small upper tail accounts for a large share of usage.

Our paper connects to recent work on how AI mediation affects the production of digital public goods. Gans (2024) studies contribution incentives when AI systems are trained on user-contributed data, and finds that AI contributions can lead humans to supply higher-quality and more costly contributions rather than fewer. Sun (2026) points in the other direction: in a matching model of digital public goods, private AI resolution lowers both problem submissions and public solutions, so the shared knowledge stock depreciates. Our mechanism is closer to the second, but it operates on the supply of software rather than on a stock of knowledge, and it works through the engagement channel that finances maintenance. Kleinberg and Raghavan (2021) show that convergence on a common algorithm can reduce welfare even when that algorithm is more accurate for each decision maker in isolation. This is the natural benchmark for the concentration of demand on the packages that AI agents recommend, a channel we document in Section 2 but do not model. Relatedly, Koren et al. (2026) argue that the direction of technical change matters when tasks and skills are high-dimensional. The same consideration applies to OSS, where the variety of shared packages reflects the variety of human needs and experience that

generate them.

The remainder of this paper is structured as follows. Section 2 documents industry trends and institutional details that motivate the model, including rapid AI diffusion, engagement substitution under AI mediation, and concentration in OSS outcomes. Section 3 develops the model and derives equilibrium and welfare results in the baseline economy and under vibe coding. Section 4 discusses parameter choices and illustrates the quantitative implications. Section 5 concludes and discusses implications for platform design and policy.

2 Industry Trends and Institutional Details

This section provides institutional background and descriptive patterns that motivate the model and provide reduced-form evidence consistent with its main mechanism. We emphasize three facts. First, AI-assisted coding has diffused rapidly and its capabilities have improved quickly, making the technology shock quantitatively large. Second, AI mediation appears to substitute for direct engagement with upstream knowledge producers, weakening the engagement-based channel through which many open source software (OSS) projects capture private returns. Third, outcomes in OSS are highly concentrated, consistent with selection into sharing and heavy-tailed project "quality".

2.1 AI Coding Adoption and Capability Growth

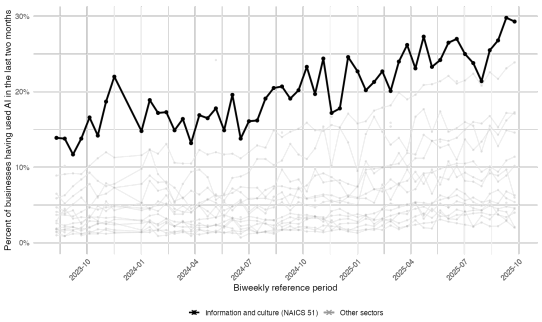
AI-assisted coding has become embedded in production at large software organizations. By October 2024, more than a quarter of all new code at Google was generated by AI and subsequently reviewed and accepted by engineers (Pichai, 2024). At Anthropic, CEO Dario Amodei reported even higher shares in September 2025, stating that "70, 80, 90 percent of the code is written by Claude" (Amodei, 2025). Consistent with these adoption rates reflecting a productivity shifter rather than full automation, qualitative evidence indicates that experienced developers use agents as assistance tools while retaining control over system design and core implementation decisions (Huang et al., 2025).

Micro-level evidence from GitHub corroborates rapid diffusion. Using commit-level data from 170,000 developers, Daniotti et al. (2025) estimate that by the end of 2024, AI generated roughly 29–30% of Python functions authored by U.S. contributors, and adoption is associated with a 3.6% increase in quarterly code contributions.

Capabilities have improved quickly along relevant dimensions. On SWE-bench,[1] which evaluates issue resolution on real GitHub repositories, the best-performing model in 2024 (Claude Sonnet 3) solved only 1.96% of issues (Jimenez et al., 2024). By early 2026, the

[1] https://www.swebench.com

(a) Business AI adoption by industry



(b) AI use in software development

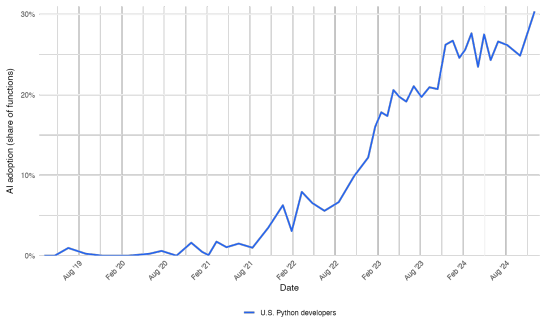


Figure 1: Adoption of AI across industries and within software development.
Notes: Panel (a) reports the share of businesses using AI by industry, based on survey evidence from the U.S. Census Bureau's Business Trends and Outlook Survey (BTOS); information-intensive industries exhibit the highest adoption rates. Panel (b) reports the estimated share of software code written with AI assistance using GitHub commit data from Daniotti et al. (2025).

same model family (Claude Sonnet 4.5) solved 74.2% of issues. Rapid improvements in model capability help explain why adoption can accelerate sharply once performance crosses a usability threshold. The same 2024–2026 window is therefore a natural period in which to look for the model's short-run implications in observed software usage and engagement.

Figure 1 summarizes adoption patterns using two complementary sources. Panel (a) reports the share of businesses using AI by industry from the U.S. Census Bureau's Business Trends and Outlook Survey (BTOS), showing that information-intensive industries lead adoption. Panel (b) reports AI intensity in software development based on GitHub commit data from Daniotti et al. (2025). In the model, these patterns motivate treating AI as a large, fast shock to the productivity of using and composing existing software.

2.2 How OSS Maintainers Capture Private Returns

OSS generates large social value relative to its direct production cost (Hoffmann et al., 2024). Individual maintainers and small teams therefore typically sustain projects by capturing only a small fraction of the surplus they create, through channels that are closely tied to user visibility and engagement (Lerner and Tirole, 2002; Jones, 2020). These channels include reputation and career opportunities (downloads, stars, citations), consulting and support leads that originate from documentation and community interaction, and paid complements such as hosted versions or enterprise add-ons. Many of these channels rely on direct interaction: a user who reads documentation, opens an issue, or asks a public question creates attention and signals demand.

The importance of engagement-based monetization varies across market segments. A large share of monetized activity around OSS occurs in enterprise services: the large enterprises segment accounted for more than 69.0 percent of the 2022 open source services market

(Grand View Research, 2022). In contrast, a large fraction of OSS usage occurs in small-team and individual production, where adoption may be high but monetization relies on discovery through community channels rather than formal procurement.

In the model, we summarize private returns by a reward per direct user, and we allow AI mediation to reduce the extent to which usage translates into monetizable engagement.

This engagement-dependent channel is most important for community-maintained packages, where individual developers or small teams rely on visibility, reputation, and consulting leads for their returns. Projects backed by large corporations for strategic or complementary reasons—such as maintaining a platform ecosystem or supporting internal tooling—derive private returns that are less sensitive to the mode of user interaction. Our analysis therefore applies most directly to the long tail of community-driven OSS, which accounts for the majority of projects even as corporate-backed packages dominate aggregate download counts.

2.3 Engagement Substitution Under AI Mediation

Before documenting how AI affects engagement, it is useful to establish how low baseline engagement already is. Table 4 and Figure 7 in the Appendix report GitHub activity for 25 npm packages with weekly downloads between 1,000 and 100,000, classified as either developer tools (directly installed by developers) or infrastructure (typically pulled in as transitive dependencies). Across all packages, the median stars-to-downloads ratio is 0.06 percent and the median ratio for other interactions (forks, issues, and pull requests) is 0.01 percent. The distribution of engagement conditional on usage is characterized by many near-zeros and a thin right tail: most users never interact with maintainers. This low baseline might suggest little room for further decline, but precisely because engagement is already thin, even modest additional diversion can be consequential for maintainer revenue. When the typical package converts a fraction of a percent of downloads into any form of engagement, a 25 percent reduction in that already-small flow can meaningfully erode the signal—and revenue—that sustains maintenance.

The tool–infrastructure distinction reveals where the demand-diversion channel bites hardest. Developer tools—packages that developers consciously select and interact with—show a strong positive relationship between downloads and engagement: the log–log slope for stars is 0.71 and for other interactions 0.58 (Figure 7). For these packages, attention metrics scale meaningfully with usage, and the model's mechanism operates visibly: AI mediation that diverts users away from documentation and discovery directly reduces the engagement that sustains maintainers.

Infrastructure packages, by contrast, show near-zero slopes (0.14 and 0.02): downloads and engagement are essentially uncorrelated, because these packages are typically second-

layer dependencies—imported by packages that developers chose, rather than chosen directly. For such packages, AI-driven demand diversion may matter less, since engagement was already negligible and monetization, if any, relies on different channels.[2] Our model is general and applies to both segments, but its empirical bite is sharpest for the tool segment where engagement-based monetization is most prevalent.

The model's long-run mechanism relies on a wedge between OSS usage and the direct engagement that sustains maintainers. If AI-mediated assistance substitutes for direct interaction, the monetizable engagement margin can shrink even as adoption expands.

For public Q&A platforms, del Rio-Chanona et al. (2024) provide causal evidence that access to ChatGPT reduced Stack Overflow activity by about 25 percent within six months relative to counterfactual platforms with limited access. Since then, overall Stack Overflow traffic has continued to decline sharply (Orosz, 2025).

Figure 2 illustrates a similar decoupling for Tailwind CSS. Weekly npm downloads rise steadily, indicating growing usage of the framework, while public questions tagged with `tailwind-css` decline. Tailwind's creator reports that "traffic to our docs is down about 40% from early 2023 despite Tailwind being more popular than ever" and that revenue is "down close to 80%" (Wathan, 2026). Together, these patterns suggest that AI mediation can divert interaction away from the surfaces where OSS projects monetize and recruit contributors.

2.4 Concentration and Selection in OSS Outcomes

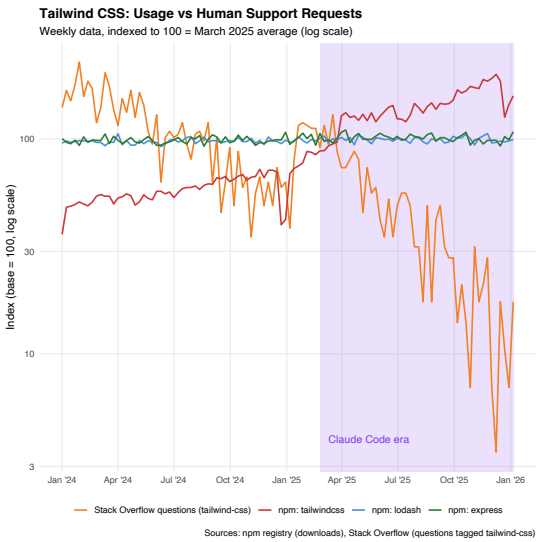
Outcomes in open source software are highly unequal. Figure 3 plots the distribution of GitHub repository success using stars and downstream dependencies as proxies for attention and adoption. The figure exhibits an approximately linear relationship on log–log scales, indicating Pareto-like behavior in the upper tail: a small fraction of projects attracts a disproportionate share of attention, usage, and downstream reliance.

Outcomes are concentrated not only in the upper tail but also on the extensive margin: most repositories attract no attention and are never reused. Of approximately 125 million GitHub repositories, only 10.32 million have ever received at least one star. Restricting attention to JavaScript projects, GHTorrent records 13.4 million repositories, of which only 1.28 million have at least one star (Gousios, 2013). Using Libraries.io (Libraries.io, 2025), we further observe that 840,419 JavaScript repositories are published as npm packages.

Downstream dependence is even more concentrated. Only 392,828 projects ever appear as a dependency of another project. Thus, only a small fraction of repositories are both

[2]For infrastructure packages, visibility metrics such as stars and issues may be poor proxies for the interactions that matter. Metrics based on dependency-graph centrality, security-audit participation, or corporate sponsorship may better capture the relevant engagement surfaces.

(a) Tailwind: usage vs public Q&A



(b) Stack Overflow questions over time

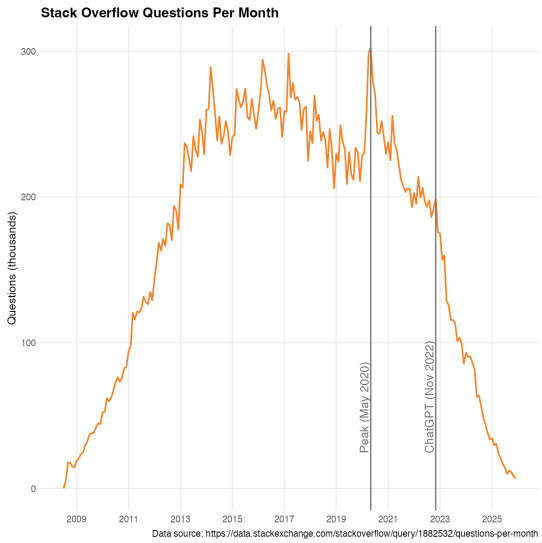


Figure 2: Usage grows while engagement falls.

Notes: Panel (a) compares Tailwind CSS usage (npm downloads) to a measure of public engagement (Stack Overflow questions tagged with `tailwind-css`). Panel (b) shows the broader decline in Stack Overflow question volume. del Rio-Chanona et al. (2024) provide causal evidence of a post-ChatGPT decline in activity; Wathan (2026) reports a concurrent decline in Tailwind documentation traffic and revenue.

shared and reused, consistent with large fixed costs of packaging and documentation and with strong selection into economically relevant OSS. In the model, these facts motivate a quality distribution with a heavy right tail and an endogenous sharing cutoff.

In addition to the graphical rank–size evidence, Table 1 quantifies the heaviness of the upper tail of OSS attention using GitHub stars. We estimate log–rank regressions of the form $\log(\text{rank}) = a + b \cdot [-\log(\text{stars})]$ on binned medians, where the slope b maps directly to the implied Pareto exponent under the model's canonical orientation. Columns progressively restrict the sample to increasingly elite repositories (top 75%, 50%, 10%, and 5% by rank).

2.5 Matched Model Choices and Package Outcomes

The Tailwind case study motivates a narrower comparison across packages. We hold a website's functional requirements fixed and ask whether earlier and later frontier models name different direct dependencies when building the same site. This comparison documents differences in model choices; it does not identify the effect of model recommendations or agent adoption on package outcomes.

2.5.1 Experimental Design

We draw 500 website tasks from BuiltWith's ranking of prominent websites. For each task, we construct a product requirements brief that describes the site's functions but names

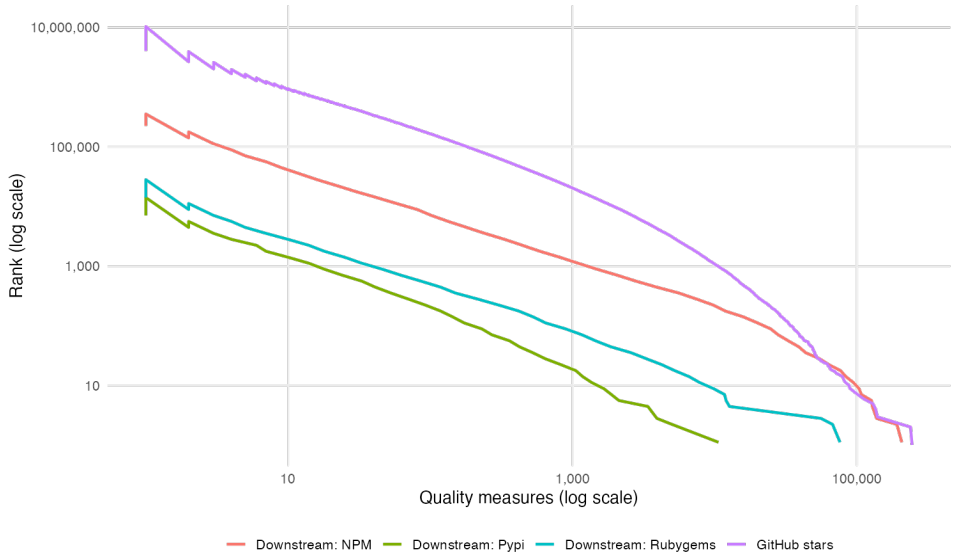


Figure 3: Concentration of attention and usage in open source software.
Notes: The figure plots log–log rank–size relationships for GitHub repositories using stars (attention) and downstream dependencies (usage) as proxies for project success. Repositories are ranked by outcome, and the figure reports median values within logarithmic rank bins. The approximately linear relationship on log–log scales indicates Pareto-like behavior in the upper tail, with extreme heterogeneity in outcomes. Data sources: GHTorrent (Gousios, 2013) and Libraries.io (Libraries.io, 2025).

neither the site nor a programming language, library, framework, or other implementation technology. We run the same brief through an earlier and a later model, then record packages named directly in the generated `package.json`. Packages installed transitively through those direct dependencies are not model choices in this comparison.

The two comparisons remain separate throughout the analysis. GPT-4o and GPT-5.2-Codex both completed 498 briefs; Claude Sonnet 3.7 and Claude Sonnet 4.5 both completed 97. For each pair, we calculate the share of common briefs that names a package directly. A package is *New AI* when the later model's share exceeds the earlier model's share by more than 1.6 pooled-binomial standard deviations, *Early AI* when it is lower by the same margin, and *Stable AI* when the difference falls inside that no-change band. Thus the Claude groups contain 95 New AI, 318 Stable AI, and 70 Early AI packages; the corresponding GPT groups are constructed with the 498 common briefs and are not pooled with the Claude groups.

Figure 4 displays the underlying package-choice variation that generates these labels. Panel (a) compares repeated runs of the same DeepSeek model to the observed cross-version distribution, showing that within-model variation is small relative to the between-version shifts. Panels (b) and (c) plot direct-choice counts for the GPT and Claude pairs on common briefs. Points off the forty-five degree line indicate that the later model names a package more often, while points below it indicate that the earlier model does. Because the briefs contain no technology names, these differences isolate model selection rather

Table 1: GitHub stars: tail regressions of log rank on log stars

Dependent Variable:	log(rank_mid)				
Model:	(1)	(2)	(3)	(4)	(5)
Variables					
Constant	16.5***	18.7***	23.3***	27.8***	25.9***
	(0.067)	(0.108)	(0.139)	(0.418)	(0.606)
Slope: $-\log(stars)$	1.08***	1.33***	1.80***	2.24***	2.07***
	(0.009)	(0.013)	(0.014)	(0.038)	(0.054)
Fit statistics					
Observations	581	436	291	59	30
R^2	0.96160	0.96240	0.98178	0.98392	0.98161

IID standard-errors in parentheses
*Signif. Codes: ***: 0.01, **: 0.05, *: 0.1*

Notes: Each column reports a binned log–rank regression for GitHub repositories, estimated on median stars within logarithmic rank bins. The dependent variable is $\log(\text{rank}_{\text{mid}})$ and the regressor is $-\log(\text{stars}_{\text{med}})$, so the reported coefficient is the (positive) slope in the canonical Zipf orientation. Column 1 uses the full sample; Columns 2–5 restrict to the top 75%, 50%, 10%, and 5% of repositories by rank, respectively. Standard errors are iid. The increasing slope in the extreme tail indicates stronger concentration among the most-starred repositories.

than stated requirements.

2.5.2 Outcomes and Descriptive Patterns

We measure weekly npm downloads and GitHub stars. npm downloads are total installations and do not distinguish human from automated use. For stars, we map packages to canonical GitHub repositories, retain only complete repository histories, exclude shared-repository conflicts within the comparison, and retain observed zero-star weeks. We also construct two comparison groups. *No AI* packages are selected from a dated pre-generative-AI npm PageRank snapshot under deterministic exclusions; *Infrastructure* packages have low direct use and high transitive use. We exclude infrastructure packages from package-concentration measures because their installations are typically transitive; that measurement restriction does not limit the model's domain. Neither group is a randomized control.

Table 2 reports arithmetic mean weekly outcomes in Q1 2025 and Q1 2026 for the Claude comparison. Mean downloads grew by 152.0% for the New AI group, 104.5% for Stable AI, 59.4% for Early AI, and 29.8% for the No AI comparison group. Star gains fell in every group: 43.9%, 32.2%, 38.5%, and 28.1%, respectively. These are contrasts among differently composed groups, not estimates of a treatment effect. The GPT comparison likewise produces different direct-dependency groups for the same briefs, but its outcomes are kept separate rather than combined with the Claude comparison.

Figure 5 shows the weekly trajectories that produce these differences. For both model

Table 2: Descriptive package outcomes by Claude direct-dependency group

Group	N	Q1 2025	Q1 2026	Growth
Panel A: Weekly npm downloads				
New AI	95	9,540,566	24,043,384	+152.0%
Stable AI	318	2,789,331	5,704,365	+104.5%
Early AI	70	9,638,217	15,360,060	+59.4%
No AI	366	3,132,715	4,065,109	+29.8%
Infrastructure	3,178	15,639,516	27,840,078	+78.0%
Panel B: Weekly GitHub stars				
New AI	95	38.89	21.83	-43.9%
Stable AI	318	18.53	12.57	-32.2%
Early AI	70	15.83	9.73	-38.5%
No AI	366	8.10	5.83	-28.1%
Infrastructure	3,178	4.47	3.15	-29.5%

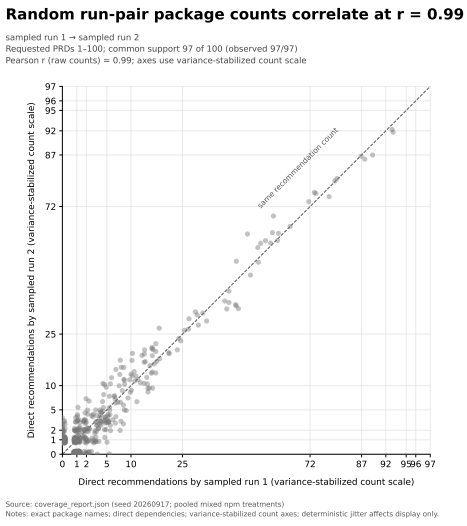
Notes: New AI, Stable AI, and Early AI classify direct `package.json` choices by Claude Sonnet 3.7 and Claude Sonnet 4.5 on the same 97 briefs. Each cell is the arithmetic mean weekly outcome for the stated quarter; displayed N is the full group count. The star means use only mapped packages with complete repository histories after shared-repository exclusions, while downloads use all packages with complete npm observations. Observed zero-star weeks remain zero.

pairs, packages named more often by the later model (New AI) gain downloads more quickly after the paired comparison window begins, while packages named more often by the earlier model (Early AI) gain more slowly. Star trajectories move in the opposite direction: weekly star gains fall for every direct-dependency group. The timing pattern in downloads is especially suggestive because it follows the model-version comparison rather than a stated-technology demand shock, but the figure remains descriptive. It does not randomize package exposure or real-world agent adoption.

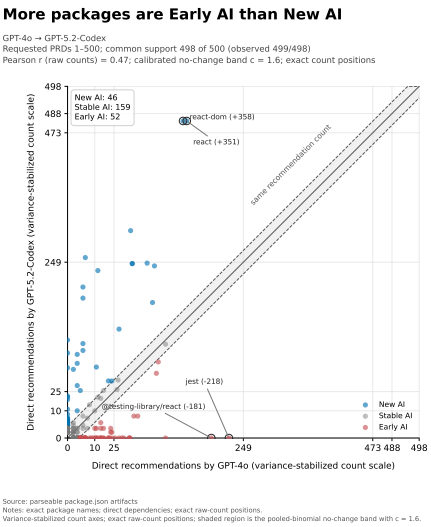
2.5.3 Interpretation

Holding requirements fixed rules out explicit technology requests as the source of the model-choice differences. It does not randomize package exposure or real-world agent adoption, so these patterns cannot establish effects on downloads, GitHub attention, maintainer returns, contribution, entry, or long-run variety. The model instead supplies causal equilibrium predictions under its stated assumptions. The descriptive comparisons show that model configurations select different direct packages for identical requirements and that the resulting package groups have different subsequent download and star trajectories; those short-run patterns are consistent with, but do not test, the model's demand-diversion mechanism.

(a) Within-model variation



(b) Across-model variation: GPT



(c) Across-model variation: Claude

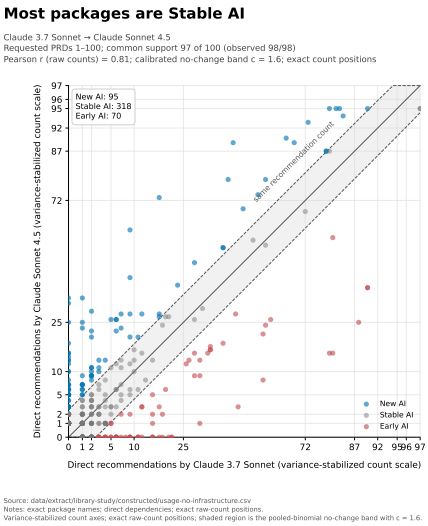
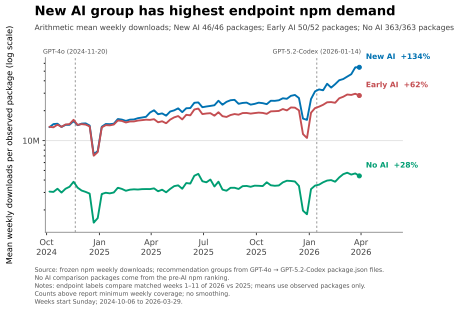
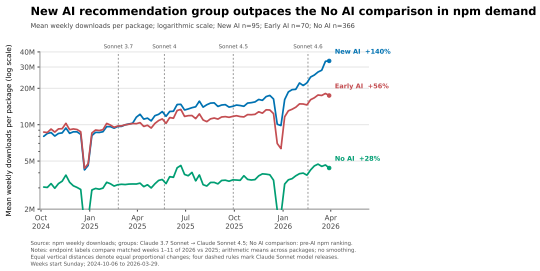


Figure 4: Within- and across-model variation in direct package choices.
Notes: Panel (a) compares repeated runs of DeepSeek to a calibrated no-change band. Panels (b) and (c) plot direct-choice counts on common briefs for GPT-4o versus GPT-5.2-Codex and Claude Sonnet 3.7 versus Claude Sonnet 4.5. Points above the forty-five degree line indicate that the later model names a package more often.

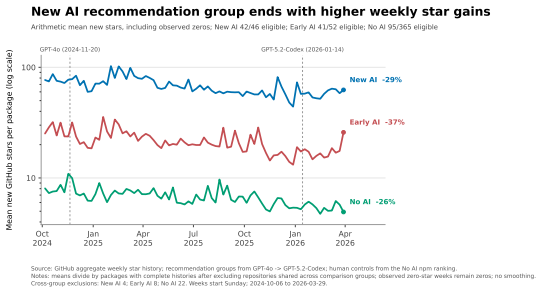
(a) Downloads: GPT-4o vs. GPT-5.2-Codex



(b) Downloads: Claude Sonnet 3.7 vs. Claude Sonnet 4.5



(c) Stars: GPT-4o vs. GPT-5.2-Codex



(d) Stars: Claude Sonnet 3.7 vs. Claude Sonnet 4.5

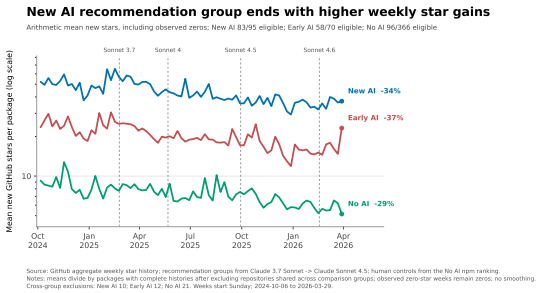


Figure 5: Weekly downloads and GitHub stars by direct-dependency group.
Notes: Panels (a) and (b) plot weekly npm downloads; panels (c) and (d) plot weekly GitHub stars. New AI packages are named more often by the later model; Early AI packages are named more often by the earlier model; No AI packages are a pre-generative-AI comparison group. The GPT and Claude comparisons use separate pair-specific groups and are not pooled.

3 A Model of the Open Source Software Ecosystem

This section develops a simple model of the open source software (OSS) ecosystem. We begin with a baseline economy without vibe coding: users choose among shared packages that differ in quality, while developers decide whether to release their projects as OSS after learning realized quality. The equilibrium jointly determines software variety and quality—captured by the mass of shared packages and the endogenous distribution of quality—together with market-clearing relationships that link user adoption to developer payoffs. We then introduce vibe coding as a technology shock that changes how users access OSS and how developers appropriate value, and we trace how this new usage technology feeds back into sharing and entry decisions.

3.1 Decision Makers and Environment

3.1.1 Users

There is a unit mass of final users with heterogeneous software needs. Each user can choose from a mass m_s of shared software packages. Because software is nonrival, the same package can be used by many users. Package k has quality q_k and delivers flow utility uq_k to the representative user. The equilibrium objects m_s, the distribution of qualities, and the utility shifter u are taken as given by individual users. Quality is heterogeneous with distribution $\mu(q)$.

The utility of user $i \in [0,1]$ from using package $k \in [0, m_s]$ is

$$U_{ik} = \nu_{ik} q_k u, \qquad \nu_{ik} \overset{\text{i.i.d.}}{\sim} \text{Fréchet}\left(\sigma, 1/\Gamma(1-1/\sigma)\right), \tag{1}$$

The idiosyncratic preference shock ν_{ik} captures heterogeneity in user needs: a given package may be a good match for some tasks but not others. The shape parameter $\sigma > 1$ governs substitutability across packages; the scale parameter normalizes $\mathbb{E}[\nu_{ik}] = 1$ (Anderson et al., 1992).

Each user chooses the package k that maximizes U_{ik}. Integrating over the preference shocks implies that a given package is chosen with probability

$$\Pr(k) = \frac{q_k^\sigma}{m_s \int q_j^\sigma \, d\mu(j)}.$$

The adoption probability is increasing in quality. If all packages have the same quality, users randomize uniformly and $\Pr(k) = 1/m_s$. We denote

$$\bar{q} = \left[\int q_j^\sigma \, d\mu(j) \right]^{1/\sigma}$$

the average quality of distributed software.

Using standard discrete-choice results (Anderson et al., 1992), the expected utility of a user can be written as

$$\mathcal{U} = \bar{q} u m_s^{1/\sigma}.$$

Expected utility is increasing in average software quality $\bar{q}$ and in the utility shifter u, which we later allow to change with vibe coding. It is also increasing in the mass of available packages m_s: holding the quality distribution fixed, a larger choice set raises welfare through a love-of-variety (or better-match) effect, with curvature governed by σ.

A package of quality q_k is chosen by $n(q_k) = \Pr(k) = q_k^\sigma / (m_s \bar{q}^\sigma)$ users. The user base is increasing in own quality, but decreasing in the number of competing packages and in their average quality.

3.1.2 Developers

Each package is developed by a single developer (we abstract from collaboration; for large projects, interpret the team as a single decision maker—internal free-riding and coordination costs are second-order for the sharing and entry margins we study). Development proceeds in two stages. First, the developer incurs an up-front cost Φ to design, build, and test the software before its quality is known. This assumption, standard in heterogeneous-firm models (Melitz, 2003), captures that software quality is hard to predict ex ante (Jones, 2020, page 136). Second, after observing realized quality, the developer decides whether to release the package as open source by paying a fixed sharing cost τ, which captures packaging, documentation, and ongoing maintenance.

Developers share because a larger user base generates private returns—reputation, career benefits, and monetization opportunities. We capture these channels by allowing developer payoff to depend on the number of users:

$$\Pi(q_k) = \pi n(q_k) = \pi q_k^\sigma / (m_s \bar{q}^\sigma),$$

where π is the reward per user. We allow π to depend on the usage technology and, in particular, on whether users interact directly with the project or through vibe coding; we specify this dependence below.

The parameter π should be interpreted as the *engagement-sensitive component* of developer returns. For corporate-backed projects where private returns are largely strategic (e.g., platform adoption, complementary cloud services), π is less sensitive to the mode of user interaction. Our comparative statics therefore apply most directly to community-maintained projects, which constitute the vast majority of OSS repositories even though corporate projects account for a disproportionate share of downstream usage. In terms of

the empirical distinction between developer tools and infrastructure packages (Section 2), the demand-diversion channel operates most visibly for developer tools, where π is engagement-sensitive and scales with usage. For infrastructure packages, π may be near zero regardless of AI mediation, so the channel is muted even under the traditional business model.

Project quality is heterogeneous. For tractability, assume each developer draws realized project quality $q \geq 1$ i.i.d. from a Pareto distribution with shape parameter $\gamma > 0$ and normalized scale,

$$\Pr(q > x) = x^{-\gamma}, \qquad x \geq 1,$$

so that the density is $\gamma q^{-\gamma-1}$. The Pareto assumption captures the empirical right tail of software project outcomes and delivers closed-form expressions for equilibrium objects.

After observing q, the developer decides whether to release the project as OSS. Sharing requires paying a fixed cost τ and yields payoff $\Pi(q)$. Because user adoption is increasing in project quality, developer payoff is increasing in q. In particular,

$$\Pi(q) = \pi n(q) = \frac{\pi q^{\sigma}}{m_s \bar{q}^{\sigma}}.$$

A developer shares if and only if the payoff from doing so covers the sharing cost,

$$\Pi(q) \geq \tau.$$

Since $\Pi(q)$ is increasing in q, the sharing decision is characterized by a cutoff q_0 defined by the indifference condition $\Pi(q_0) = \tau$: projects with $q \geq q_0$ are shared, while projects with $q < q_0$ are not.

The cutoff q_0 is endogenous. It depends on the equilibrium objects m_s and $\bar{q}$ that determine user adoption and, hence, developer payoffs; at the same time, both m_s and $\bar{q}$ depend on q_0 through the selection of which quality draws are shared.

Given a cutoff q_0, the distribution of quality among shared projects is a truncated Pareto. Under the quality aggregator defined above, average quality satisfies

$$\bar{q} = \Lambda q_0,$$

where $\Lambda = \left(\frac{\gamma}{\gamma-\sigma}\right)^{1/\sigma} > 1$. The condition $\gamma > \sigma$ ensures that the moment determining $\bar{q}$ is finite. If there is a mass m of developers who have built a software package, the mass of shared software is

$$m_s = m \Pr(q \geq q_0) = m q_0^{-\gamma}.$$

A higher cutoff reduces the fraction of projects that are shared (lower m_s), but raises the average quality of shared OSS (higher $\bar{q}$).

Imposing the sharing indifference condition $\Pi(q_0) = \tau$ and substituting $\bar{q} = \Lambda q_0$ and $m_s = m q_0^{-\gamma}$ yields the equilibrium cutoff

$$q_0 = (\tau m \Lambda^\sigma / \pi)^{1/\gamma}.$$

This implies equilibrium average quality

$$\bar{q} = \Lambda^{1+\sigma/\gamma} (\tau m / \pi)^{1/\gamma}$$

and equilibrium variety

$$m_s = \frac{\pi}{\tau \Lambda^\sigma},$$

so that the mass of shared software increases in the reward per user π and decreases in the sharing cost τ.

Notably, m_s is independent of the mass of produced projects m. Entry increases the pool of potential projects, but it also intensifies competition for users, reducing the payoff from sharing and raising the cutoff q_0; with Pareto tails, these two forces exactly offset in equilibrium. This independence is a knife-edge property of the Pareto distribution; for general heavy-tailed distributions, entry would affect variety, but the qualitative channel—that eroding π reduces both entry and sharing—is robust.

Substituting equilibrium objects back into developer payoff gives

$$\Pi(q) = \frac{\pi q^\sigma}{m_s \bar{q}^\sigma} = q^\sigma \Lambda^{-\sigma^2/\gamma} \tau \left(\frac{\pi}{\tau m} \right)^{\sigma/\gamma} = \tau \left(\frac{q}{q_0} \right)^\sigma.$$

Developer payoff is increasing in quality q and in the reward per user π, and decreasing in the mass of competing projects m. It is also increasing in the sharing cost τ because $\gamma > \sigma$: higher sharing costs discourage marginal developers from sharing, reducing competition for those who do share (formally, $\Pi(q) \propto \tau^{1-\sigma/\gamma}$).

Finally, before drawing quality, a developer's expected net payoff from entering the development stage (gross of the common up-front cost Φ) is

$$\mathbb{E}[\max\{\Pi(q) - \tau, 0\}] = \int_{q=q_0}^{\infty} [\Pi(q) - \tau] \, \gamma q^{-\gamma-1} \, dq = \frac{\sigma}{\gamma - \sigma} \cdot \frac{\pi}{m \Lambda^\sigma}.$$

Low-quality draws $q < q_0$ are not shared and therefore earn zero.

Free entry requires that the expected payoff from developing software equals the up-front cost:

$$\Phi = \frac{\sigma}{\gamma - \sigma} \frac{\pi}{m \Lambda^\sigma}.$$

Rearranging pins down the equilibrium mass of developers:

$$m = \frac{\sigma}{\gamma - \sigma} \frac{\pi}{\Phi \Lambda^{\sigma}}.$$

Entry is increasing in the reward per user π (larger market) and decreasing in the development cost Φ (higher barrier). Entry is also decreasing in the quality aggregator Λ: fatter-tailed quality distributions (lower γ, higher Λ) imply that high-quality projects capture more of the market, making entry less attractive for marginal developers.

3.2 Software Production Function

The cost of developing software is endogenous. Developers rely on their own time and effort, but also use existing software tools to build new ones. When software quality and variety increase, development becomes cheaper—a feedback we call the *software-begets-software* effect. Importantly, humans remain essential: they design software architecture, make product decisions, and oversee implementation, even when AI agents write substantial portions of the code. Survey evidence confirms that experienced developers use AI agents as productivity tools while retaining control over design and implementation (Huang et al., 2025).

We model software production with a Cobb-Douglas technology: elasticity $\beta \in (0, 1)$ to existing software and $1 - \beta$ to labor. Software can substitute for human effort, but not completely. The parameter β governs the strength of the software-begets-software feedback.

Developers value software quality and variety in the same way as final users—their productive value from the software ecosystem is

$$\mathcal{U} = \bar{q} u m_s^{1/\sigma} = u \Lambda^{\sigma/\gamma} m^{1/\gamma} \pi^{1/\sigma - 1/\gamma} \tau^{1/\gamma - 1/\sigma}.$$

This symmetry is a simplification: developers are themselves users of existing packages when building new software, though in practice they may weight API stability and documentation quality differently from final users. The same functional form also fits the final-user side of the market. In practice, the "consumer" in the model is often a programmer assembling a custom, one-off solution from existing OSS packages—a script, a data pipeline, or an internal tool. We model this assembly problem in reduced form with the same aggregator $\mathcal{U} = \bar{q}\, u\, m_s^{1/\sigma}$, because what matters for both usage and production is access to high-quality packages, a productivity shifter u, and variety m_s. Final users, however, do not contribute new OSS varieties: they take $(m_s, \bar{q})$ as given and choose which package to use and whether to use it directly or via vibe coding. We normalize away an explicit labor choice for final users by treating their time/effort constraint as a scale parameter. This affects the level of utility but not the adoption decision that pins down the

vibe-coding share v.

The software development cost is therefore

$$\Phi = \kappa^{1-\beta}\mathcal{U}^{-\beta} = \kappa^{1-\beta}u^{-\beta}\Lambda^{-\beta\sigma/\gamma}m^{-\beta/\gamma}\left(\frac{\tau}{\pi}\right)^{\beta/\sigma-\beta/\gamma}.$$

Development cost is increasing in labor cost κ and decreasing in software quality $\bar{q}$, productivity u, and variety m_s. This expression follows from cost minimization under Cobb-Douglas: minimizing $\kappa L + p_S S$ subject to $L^{1-\beta}S^{\beta} = 1$ yields unit cost proportional to $\kappa^{1-\beta}p_S^{\beta}$; interpreting the "price" of software as the inverse of its productivity $\mathcal{U}^{-1}$ gives the stated form.

The productivity shifter u will capture the effect of vibe coding on development: AI assistance makes any given package easier to use and faster to integrate. Because u enters with exponent $-\beta$, the cost reduction from higher u is proportional to the software share β. When software accounts for a larger share of production (β higher), improvements in software productivity translate into larger cost reductions.

Assumption 1 (Parameter Restrictions)**.** Quality dispersion exceeds variety substitutability:

$$\gamma > \sigma.$$

3.3 Equilibrium

We now define equilibrium in the baseline economy (without vibe coding, so u is a fixed productivity parameter and all users interact directly with OSS).

Definition 1 (Baseline Equilibrium)**.** A *baseline equilibrium* is a tuple $(m, q_0, m_s, \bar{q})$ with $m > 0$, $q_0 \geq 1$, $m_s > 0$, and $\bar{q} > 0$ such that:

1. **(User optimality)** Each user chooses package k to maximize $U_{ik} = \nu_{ik}q_k u$, implying adoption probability $\Pr(k) = q_k^{\sigma}/(m_s\bar{q}^{\sigma})$.
2. **(Sharing cutoff)** A developer shares if and only if $\Pi(q) \geq \tau$, where $\Pi(q) = \pi q^{\sigma}/(m_s\bar{q}^{\sigma})$. The marginal project satisfies $\Pi(q_0) = \tau$.
3. **(Quality aggregation)** Average quality among shared projects is $\bar{q} = \Lambda q_0$ with $\Lambda = \left(\frac{\gamma}{\gamma-\sigma}\right)^{1/\sigma}$.
4. **(Mass of shared software)** Given mass m of developed projects, $m_s = mq_0^{-\gamma}$.
5. **(Free entry)** Developers enter until expected payoff equals development cost:

$$\frac{\sigma}{\gamma-\sigma}\cdot\frac{\pi}{m\Lambda^{\sigma}} = \Phi,$$

where $\Phi = \kappa^{1-\beta}u^{-\beta}\Lambda^{-\beta\sigma/\gamma}m^{-\beta/\gamma}\left(\frac{\tau}{\pi}\right)^{\beta/\sigma-\beta/\gamma}$.

Conditions (2)–(4) jointly determine q_0, $\bar{q}$, and m_s as functions of m. Condition (5) then pins down the equilibrium mass of developers m.

The software-begets-software feedback creates a magnification. When entry increases, quality rises, which lowers development costs, which encourages more entry.

Proposition 1 (Equilibrium Existence and Uniqueness)**.** *Under Assumption 1, a unique baseline equilibrium exists. The equilibrium is characterized by:*

$$q_0 = \left(\frac{\tau m \Lambda^\sigma}{\pi}\right)^{1/\gamma}, \tag{2}$$

$$\bar{q} = \Lambda q_0 = \Lambda^{1+\sigma/\gamma}\left(\frac{\tau m}{\pi}\right)^{1/\gamma}, \tag{3}$$

$$m_s = \frac{\pi}{\tau \Lambda^\sigma}, \tag{4}$$

where m solves the reduced-form free entry condition

$$m^{1-\beta/\gamma} = \frac{\sigma}{\gamma - \sigma} \cdot \Lambda^{\beta\sigma/\gamma - \sigma} \cdot \kappa^{\beta-1} \cdot u^\beta \cdot \pi^{1+\beta/\sigma-\beta/\gamma} \cdot \tau^{\beta/\gamma-\beta/\sigma}. \tag{5}$$

Proof. See Appendix A. □

Define $\eta \equiv 1 - \beta/\gamma > 0$. Taking logs of (5) and differentiating:

Corollary 1 (Comparative Statics)**.** *Equilibrium entry m satisfies:*

$$\frac{\partial \log m}{\partial \log \pi} = \frac{1 + \beta/\sigma - \beta/\gamma}{\eta} > 1, \tag{6}$$

$$\frac{\partial \log m}{\partial \log u} = \frac{\beta}{\eta} > 0, \tag{7}$$

$$\frac{\partial \log m}{\partial \log \kappa} = -\frac{1-\beta}{\eta} < 0, \tag{8}$$

$$\frac{\partial \log m}{\partial \log \tau} = \frac{\beta/\gamma - \beta/\sigma}{\eta}. \tag{9}$$

The first elasticity exceeds one: $1 + \beta/\sigma - \beta/\gamma > 1 - \beta/\gamma$ because $\beta/\sigma > 0$. This reflects the software-begets-software feedback: higher π raises entry, which improves quality and variety, which lowers costs and further encourages entry. The second elasticity β/η is positive; since $\eta < 1$, we have $\beta/\eta > \beta$, amplifying the direct productivity effect.

Entry increases in developer reward π and software productivity u, and decreases in labor cost κ. The effect of sharing cost τ depends on parameters. The intuition for each effect:

- *Reward (π)*: Higher reward per user makes sharing more attractive, drawing in marginal developers. The elasticity exceeds one because of the software-begets-software feedback: more entry improves the ecosystem, lowering costs and inducing further entry.

- *Productivity (u)*: Higher software productivity lowers development costs, encouraging entry. The elasticity $\beta/\eta > \beta$ reflects amplification through the feedback loop.
- *Labor cost (κ)*: Higher wages raise development costs, discouraging entry. The magnitude is smaller when software accounts for a larger share of production.
- *Sharing cost (τ)*: Higher τ discourages sharing (reducing m_s) but raises the quality cutoff (improving $\bar{q}$). Both margins matter because developers use the shared stock as an input into production. Because $\gamma > \sigma$, the variety loss outweighs the quality gain in the software aggregator, so development costs rise and entry falls with τ: the exponent in (9) is negative.

Corollary 2 (Equilibrium Quality and Welfare)**.** *Average quality and user welfare in equilibrium are:*

$$\bar{q} = \Lambda^{1+\sigma/\gamma} \left(\frac{\tau}{\pi}\right)^{1/\gamma} m^{1/\gamma}, \tag{10}$$

$$\mathcal{U} = \bar{q}\, u\, m_s^{1/\sigma} = \Lambda^{\sigma/\gamma} \left(\frac{\pi}{\tau}\right)^{1/\sigma - 1/\gamma} u\, m^{1/\gamma}. \tag{11}$$

Both are increasing in m. Hence any parameter change that raises entry also raises average quality and user welfare.

Substituting the equilibrium value of m from (5), both $\bar{q}$ and $\mathcal{U}$ can be expressed purely in terms of exogenous parameters. In particular, both are increasing in software productivity u (with exponent $\beta/(\gamma-\beta)$ for $\bar{q}$ and $1+\beta/(\gamma-\beta)$ for $\mathcal{U}$) and decreasing in labor cost κ (with exponent $-(1-\beta)/(\gamma-\beta)$).

The share of user utility captured by developers is the ratio of development cost to welfare. Empirically, this ratio is approximately 0.001—development costs are about 0.1% of user utility. The small ratio reflects two forces: high productivity raises user welfare faster than development costs, and competition among developers dissipates rents. Users receive roughly 1000 times more value than developers spend creating that value—a consequence of software's nonrivalry and the scale of adoption.

3.3.1 First-Best Allocation

The comparative statics show that policy can affect entry and welfare. But is the market providing the right amount of open source software? To answer this, we characterize the first-best allocation and compare it to equilibrium.

A social planner maximizes aggregate welfare, defined as user utility minus total development costs. Developer surplus is zero under free entry, so aggregate welfare reduces to user utility minus development costs.

Definition 2 (First-Best)**.** The *first-best allocation* solves

$$\max_{m \geq 0} \mathcal{U}(m) - m \cdot \Phi(m),$$

where $\mathcal{U}(m)$ and $\Phi(m)$ are given by the equilibrium expressions (11) and the development cost evaluated at equilibrium objects.

Theorem 1 (Underprovision of Entry)**.** *Suppose $\gamma > 1 + \beta$, so that the planner's problem has an interior solution, and suppose developers capture less than a fraction $1/(\sigma\eta)$ of user welfare,*

$$\pi < \frac{\mathcal{U}^{eq}}{\sigma\eta},$$

where $\mathcal{U}^{eq}$ denotes equilibrium user welfare and, with a unit mass of users, π equals aggregate developer revenue. Then equilibrium entry is below the first-best level: $m^{eq} < m^{fb}$. Equivalently, there are too few shared software packages in equilibrium.

Proof. See Appendix A. □

The intuition is straightforward. The marginal entrant raises user welfare by $\mathcal{U}/(\gamma m)$ but appropriates only the equilibrium expected payoff $\Phi = \sigma\pi/(\gamma m)$. In addition, entry lowers the development costs of other entrants through the software-begets-software feedback, so the social marginal cost of entry is only $\eta\Phi < \Phi$. Combining the two margins, entry is underprovided whenever aggregate developer revenue π falls short of the share $1/(\sigma\eta)$ of user welfare.

The condition holds comfortably in practice: at the preferred calibration ($\sigma = 1.5$, $\beta = 1/3$, $\gamma = 3$) the threshold share is $1/(\sigma\eta) = 3/4$, while measured developer revenue is roughly 0.1 percent of user value—developers capture a share of user surplus through reputation, donations, or monetization, but far from the full amount. The theorem implies that policies raising developer appropriability (higher π) or subsidizing entry (lower Φ) move the economy toward the first-best. Real-world examples include GitHub Sponsors (raising π) and foundation grants for infrastructure projects (lowering Φ) (Boysel et al., 2024).

These baseline results set the stage for analyzing vibe coding. The key question is how AI-assisted development—which changes both u and π—affects the equilibrium provision of open source software.

3.4 Vibe Coding

We now extend the baseline model to incorporate vibe coding. The extension adds an inner nest to the user's problem: after choosing a package, the user decides how to use it. This choice affects both user utility and developer revenue, because vibe-coded usage does not generate the direct engagement that developers monetize.

Once a user has chosen a particular package k, she can choose between two usage modes:

1. *Direct usage*: the user engages directly with the open source project—reading documentation, visiting the project website, potentially discovering the developer's commercial offerings
2. *Vibe-coded usage*: the user interacts with the software through an AI assistant that has been trained on the software's documentation and codebase

We normalize the productivity of direct usage of a software package of quality q to q. Vibe-coded usage has productivity $q\zeta \geq 0$, which captures how well AI assistants can help users accomplish their goals. When $\zeta = 0$, vibe coding is not available; when $\zeta = 1$, vibe coding is equally productive as direct usage; when $\zeta > 1$, vibe coding is more productive. Both options are subject to idiosyncratic taste shocks that capture heterogeneity in user preferences over interaction modes:

$$u_k = \max\{q\varepsilon_{k1}, \zeta q\varepsilon_{k2}\}, \qquad \varepsilon_{ki} \overset{\text{i.i.d.}}{\sim} \text{Fréchet}\left(\theta, 1/\Gamma(1-1/\theta)\right), \tag{12}$$

where $\theta > 1$ is the elasticity of substitution between direct and vibe-coded usage. A higher θ means the two modes are closer substitutes, so users are more responsive to differences in productivity.

The choice probability follows from the same discrete-choice logic as the selection of a software package. Since both utilities scale with package quality q, quality cancels in the choice probability. The probability that a user chooses vibe coding is:

$$v = \frac{\zeta^\theta}{1+\zeta^\theta}. \tag{13}$$

This logistic function has an intuitive interpretation. When $\zeta < 1$, vibe coding is less productive than direct usage, and fewer than half of users adopt it. When $\zeta = 1$, users are indifferent on average, and exactly half choose vibe coding. As ζ rises above 1, vibe coding becomes more attractive, and adoption rises rapidly. Figure 6 illustrates this S-curve for $\theta = 3.5$.

The parameter θ governs how sensitive adoption is to changes in AI capability. When θ is large, small improvements in ζ around the threshold $\zeta = 1$ trigger rapid swings in adoption. The empirical evidence is consistent with large θ: AI-assisted coding went from near-zero to 29 percent of Python functions within two years (Daniotti et al., 2025), and at Anthropic "70, 80, 90 percent of the code is written by Claude" (Amodei, 2025).

Assumption 2 (Substitutability Ordering)**.** Usage modes are closer substitutes than software packages:

$$\theta > \sigma.$$

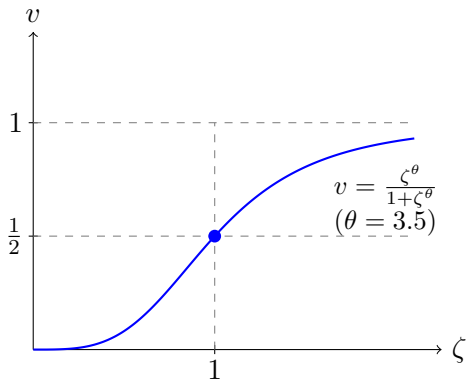


Figure 6: Vibe coding adoption as a function of AI capability ζ. When $\zeta < 1$, vibe coding is less productive than direct usage and adoption is below 50%. As ζ crosses 1, adoption rises rapidly due to the large elasticity parameter $\theta = 3.5$. The steep S-curve illustrates how small improvements in AI capability can trigger rapid shifts in usage patterns.

This assumption is empirically plausible. The parameter σ governs substitution across software packages—how easily users switch from one library to another. The parameter θ governs substitution between direct and vibe-coded usage of the *same* package. Since both modes deliver the same underlying functionality, they are closer substitutes than entirely different software packages. Empirically, the rapid adoption of vibe coding—from near-zero to majority usage within two years—is consistent with high θ: users switch easily once AI capability crosses the threshold.

The expected utility from a package of quality q, taking the max over usage modes, is:

$$\mathbb{E}[u_k] = q(1 + \zeta^\theta)^{1/\theta} = q(1 - v)^{-1/\theta}. \tag{14}$$

Define $u \equiv (1 - v)^{-1/\theta}$ as the per-unit-quality utility multiplier from having the vibe coding option. This multiplier exceeds 1 whenever $v > 0$: the option to choose between direct and vibe-coded usage always raises expected utility. The gain is larger when ζ is higher (better AI) or when θ is smaller (modes are more differentiated, so the option value is larger).

The formula $(1 - v)^{-1/\theta}$ parallels expressions in the trade literature for welfare gains from imported varieties (Arkolakis et al., 2012; Halpern et al., 2015): access to an additional option raises welfare by an amount that depends on the usage share and the elasticity of substitution.

The utility multiplier u affects both final users and developers. For users, higher u raises expected utility directly through (14). For developers, higher u lowers development cost Φ because software is an input to its own production. When AI capability ζ is high, the vibe coding share v is high, $u = (1 - v)^{-1/\theta}$ is large, and developers can build new software at lower cost. This encourages entry.

Two parameters govern the strength of this feedback. When θ is large, direct and vibe-coded usage are close substitutes, so the utility gain $(1-v)^{-1/\theta}$ is modest even for high v. When β is small, software contributes little to its own production, and the cost reduction $\Phi \propto u^{-\beta}$ is correspondingly weak.

3.4.1 Short-Run Effect: Developer Adoption Only

In the short run, developers adopt AI assistance before final users shift to vibe-coded consumption. Users still interact directly with open source projects—visiting documentation, discovering developers' commercial offerings, and generating the engagement that underlies π.

Definition 3 (Short-Run Equilibrium)**.** A *short-run equilibrium with vibe coding* is a baseline equilibrium (Definition 1) with the following modifications for some $v \in (0,1)$: developers benefit from AI-assisted development, so the productivity parameter in the development cost function is $u = (1-v)^{-1/\theta}$; users still interact directly with OSS, so the utility multiplier in their welfare remains 1; and the reward per user π remains at its baseline value $\bar{\pi}$.

Theorem 2 (Short-Run Effect of Vibe Coding)**.** *The short-run equilibrium (Definition 3) satisfies:*

$$\frac{m}{m_0} = (1-v)^{-\beta/(\theta\eta)} > 1, \tag{15}$$

$$\frac{m_s}{m_{s,0}} = 1, \tag{16}$$

$$\frac{\bar{q}}{\bar{q}_0} = (1-v)^{-\beta/(\theta\gamma\eta)} > 1, \tag{17}$$

$$\frac{\mathcal{U}}{\mathcal{U}_0} = (1-v)^{-\beta/(\theta\gamma\eta)} > 1, \tag{18}$$

where $\eta = 1 - \beta/\gamma$ and subscript 0 denotes baseline equilibrium values (computed at $v = 0$). This notation should not be confused with the quality cutoff q_0, which is an endogenous variable defined by the sharing indifference condition.

All ratios except $m_s/m_{s,0}$ exceed 1 for $v \in (0,1)$. Vibe coding raises entry, quality, and welfare in the short run, though the welfare gain is modest: users benefit only from quality selection, not from AI-assisted consumption.

Proof. From (5), $m^\eta \propto u^\beta$ (holding π fixed). Taking logs and differentiating: $\partial \log m / \partial \log u = \beta/\eta$. Substituting $u = (1-v)^{-1/\theta}$ for developer productivity gives (15). Equation (4) shows $m_s = \pi/(\tau\Lambda^\sigma)$, independent of u, yielding (16). From (3), $\bar{q} \propto m^{1/\gamma}$, giving (17). Finally, $\mathcal{U} = \bar{q} \cdot m_s^{1/\sigma}$ (with user utility multiplier equal to 1 since users interact directly) and m_s unchanged implies $\mathcal{U}/\mathcal{U}_0 = \bar{q}/\bar{q}_0$, yielding (18). □

A corollary of the short-run results is that the share of projects that get shared, m_s/m, decreases with vibe coding. Entry rises ($m \uparrow$) while variety is unchanged (m_s constant),

so a smaller fraction of projects clear the quality threshold for sharing. The model thus predicts an increase in low-quality, unshared projects—software that is created but never distributed. This prediction resonates with practitioner accounts of "AI slop": the proliferation of AI-generated code pushed to public repositories that attracts no users and serves no purpose beyond its creator's immediate needs. A related symptom is that the curl maintainer reports 20 percent of security reports in 2025 are AI-generated, consuming hours of volunteer time to review fabricated vulnerabilities (Stenberg, 2025). Meanwhile, aggregate git pushes to public repositories continue rising (GitHub, 2025), consistent with more projects being created even as quality thresholds tighten.

The short-run analysis holds π fixed, but this assumption cannot persist. Equation (12) and Figure 6 show that the vibe coding share v rises steeply with AI capability ζ. When OSS business models monetize visibility from direct users—through documentation traffic, consulting leads, or reputation—developer reward falls in proportion to $(1-v)$. As vibe coding spreads, the same library may see rising downloads but falling engagement. This erosion of the reward channel is the key mechanism behind the long-run results.

3.4.2 Long-Run Effect: Traditional Business Models

Assumption 3 (Traditional Business Model)**.** Developer reward is proportional to direct user engagement:

$$\pi = \bar{\pi}(1-v),$$

where $\bar{\pi}$ is the baseline reward per user and $(1-v)$ is the share of users who interact directly with the project.

Theorem 3 (Long-Run Effect of Vibe Coding)**.** *Under Assumptions 3 and 2, the long-run equilibrium satisfies:*

$$\frac{m}{m_0} = (1-v)^{1+\beta(1/\sigma-1/\theta)/\eta} < 1, \tag{19}$$

$$\frac{m_s}{m_{s,0}} = (1-v) < 1, \tag{20}$$

$$\frac{\bar{q}}{\bar{q}_0} = (1-v)^{\beta(1/\sigma-1/\theta)/(\gamma\eta)} < 1, \tag{21}$$

$$\frac{\mathcal{U}}{\mathcal{U}_0} = (1-v)^{1/\sigma-1/\theta+\beta(1/\sigma-1/\theta)/(\gamma\eta)}. \tag{22}$$

where $\eta = 1-\beta/\gamma$. Entry, variety, and quality all fall.

Proof. From (5), $m^\eta \propto \pi^{a_\pi} u^\beta$ where $a_\pi = 1+\beta/\sigma-\beta/\gamma$. Substituting $\pi = \bar{\pi}(1-v)$ and $u = (1-v)^{-1/\theta}$:

$$m^\eta \propto (1-v)^{a_\pi}(1-v)^{-\beta/\theta} = (1-v)^{a_\pi-\beta/\theta}.$$

The exponent $a_\pi - \beta/\theta = 1+\beta/\sigma-\beta/\gamma-\beta/\theta = \eta+\beta(1/\sigma-1/\theta)$. Taking η-th roots

gives (19). Equation (4) with $\pi = \bar{\pi}(1-v)$ gives (20). From (3), $\bar{q} \propto (m/\pi)^{1/\gamma}$. The ratio $(m/\pi)/(m_0/\bar{\pi}) = (m/m_0)/(\pi/\bar{\pi}) = (1-v)^{\beta(1/\sigma - 1/\theta)/\eta}$, yielding (21). Finally, $\mathcal{U} = \bar{q} u m_s^{1/\sigma}$ gives

$$\frac{\mathcal{U}}{\mathcal{U}_0} = \frac{\bar{q}}{\bar{q}_0} \cdot \frac{u}{u_0} \cdot \left(\frac{m_s}{m_{s,0}}\right)^{1/\sigma} = (1-v)^{\beta(1/\sigma - 1/\theta)/(\gamma\eta)} \cdot (1-v)^{-1/\theta} \cdot (1-v)^{1/\sigma},$$

and collecting exponents yields (22). □

The long-run theorem reveals a horse race between two channels. The cost channel (higher u) encourages entry with elasticity $\beta/(\theta\eta)$. The reward channel (lower π) discourages entry with elasticity $(1 + \beta/\sigma - \beta/\gamma)/\eta > 1$. The reward channel dominates: the net exponent on entry, $1 + \beta(1/\sigma - 1/\theta)/\eta$, is positive, so entry falls as vibe coding spreads.

Remark 1 (Welfare Sign Condition). The welfare exponent in (22) decomposes into three terms: a variety loss ($1/\sigma$), a productivity gain ($-1/\theta$), and a quality-selection feedback ($\beta(1/\sigma - 1/\theta)/(\gamma\eta)$). The total exponent is positive—so welfare falls—if and only if $\theta > \sigma$, which is exactly Assumption 2. Intuitively, when usage modes are closer substitutes than packages ($\theta > \sigma$), the variety loss from reduced entry outweighs the productivity gain from AI. If instead $\theta < \sigma$, welfare would rise with vibe coding adoption even under the traditional business model, because the large productivity gain from differentiated usage modes would more than offset the variety loss.

The parameters θ, σ, and β govern the magnitude. When θ is large, the productivity gain from vibe coding is small (users are nearly indifferent between modes), so the cost channel is weak. When β is small, software contributes little to its own production, again weakening the cost channel. In both cases, the reward channel dominates more strongly, and OSS provision collapses faster.

The rapid empirical adoption of vibe coding—from near-zero to majority usage within two years—tells us that either θ or ζ (or both) must be large. From $v = \zeta^\theta/(1+\zeta^\theta)$, high v requires $\zeta^\theta \gg 1$. But the welfare implications differ sharply depending on which parameter drives adoption. If θ is large, direct and vibe-coded usage are close substitutes: users switch easily, but the productivity gain $u = (1-v)^{-1/\theta}$ is modest. The cost channel is weak, and the reward channel dominates—OSS collapses quickly. If instead ζ is large (AI is highly productive), the same adoption share v delivers a larger productivity boost, strengthening the cost channel. Distinguishing these cases empirically—through revealed preference in switching behavior or direct productivity measurement—is crucial for predicting the long-run trajectory of OSS provision.

Remark 2 (Two Key Elasticities). The long-run welfare effect depends on two empirical quantities that can be read directly from the model. First, the *supply elasticity to engagement*: from Corollary 1, the elasticity of developer entry with respect to the reward per user is $\partial \log m / \partial \log \pi = (1 + \beta/\sigma - \beta/\gamma)/\eta$, which exceeds one under all calibrations

(approximately 1.25 at the preferred calibration; see the implied elasticities in Section 4). If most OSS were produced for reasons unrelated to engagement, the effective elasticity would be lower—but the npm engagement data in Appendix C and the Tailwind evidence in Section 2 suggest that the marginal revenue channel matters even when baseline engagement is sparse. Second, the *engagement elasticity to AI*: the share of users who switch to vibe coding follows $v = \zeta^\theta/(1+\zeta^\theta)$, so $\partial v/\partial \log \zeta = \theta v(1-v)$, which is largest near $v = 1/2$ and governed by θ. For the demand-diversion channel to be second-order, either the supply elasticity must be low or the engagement elasticity must be small. Under our calibration, both are large enough for the reward channel to dominate.

3.4.3 Alternative Business Models

The long-run analysis assumes developers monetize only through direct user engagement. This assumption abstracts from alternative business models that either monetize AI-mediated usage (for example through API fees charged to AI providers, telemetry-based attribution systems, or revenue-sharing agreements) or rely on revenue streams that are less tightly linked to how users access the software (enterprise licensing, foundation grants, or services sold to other developers).

The key observation is that per-user monetization can fall without destroying OSS: if vibe coding makes development cheaper, lower rewards can still support entry. However, monetization cannot fall too much. We now characterize the minimum monetization needed to sustain the pre-vibe-coding level of OSS entry.

Let π denote per-user monetization when the vibe-coding adoption share is v, and let π_0 denote the corresponding per-user monetization in a baseline economy without vibe coding ($v = 0$). We are agnostic about the specific revenue model that generates π; we only ask how large the ratio π/π_0 must be.

Theorem 4 (Minimum Monetization Bound)**.** *Under Assumptions 1 and 2, define*

$$\omega \equiv \frac{1/\theta}{1/\beta + 1/\sigma - 1/\gamma} \in (0,1).$$

Sustaining baseline entry ($m \geq m_0$) requires:

$$\frac{\pi}{\pi_0} \geq (1-v)^\omega. \tag{23}$$

Equivalently, per-user monetization can decline by at most $1-(1-v)^\omega$.

Proof. See Appendix A. □

The parameter ω measures the ratio of the productivity channel's strength to the reward channel's strength: it equals the productivity elasticity ($1/\theta$) divided by the entry-reward

elasticity ($1/\beta + 1/\sigma - 1/\gamma$). Because entry is far more elastic to monetization than to productivity ($\omega \approx 0.1$ under our calibration), the bound is tight. From Corollary 1, the elasticity of entry with respect to reward is $(1 + \beta/\sigma - \beta/\gamma)/\eta > 1$, while the elasticity with respect to the productivity shifter is $\beta/\eta < 1$. Their ratio is $1/\beta + 1/\sigma - 1/\gamma \approx 3.3$.

This asymmetry makes the quantitative implication stark. With $v = 0.7$ and $\theta = 3$, the productivity shifter is $u = (1 - v)^{-1/\theta} = 0.3^{-1/3} \approx 1.49$, which lowers development cost by a factor of $u^{-\beta} = 1.49^{-1/3} \approx 0.88$, about a 12 percent reduction. Yet Theorem 4 implies that per-user monetization can fall by at most $1 - (1 - v)^{\omega} \approx 11$ percent (since $(1 - v)^{\omega} = 0.3^{0.1} \approx 0.89$). Under the traditional business model (Assumption 3), per-user monetization falls by 70 percent ($\pi/\pi_0 = 1 - v = 0.3$), far beyond what this cost reduction can offset. The gap between a 70 percent revenue collapse and an 11 percent sustainable decline motivates alternative business models.

To connect the bound to concrete revenue models, consider a simple decomposition of per-user monetization into three components. A share $\alpha \in [0, 1]$ comes from sources independent of usage mode (for example enterprise licensing or foundation grants). The remaining share $1 - \alpha$ comes from usage-dependent sources. Direct users contribute the full usage-dependent amount, while vibe coders contribute a fraction $1 - \rho$, where $\rho \in [0, 1]$ is the *vibe discount*. Normalizing baseline per-user monetization to $\pi_0 = \bar{\pi}$, this gives

$$\frac{\pi}{\pi_0} = \alpha + (1 - \alpha)\big[(1 - v) + (1 - \rho)v\big] = 1 - (1 - \alpha)\rho v.$$

Setting $\alpha = 0$ and $\rho = 1$ recovers the traditional model, $\pi/\pi_0 = 1 - v$.

Corollary 3 (Business Model Constraint)**.** *Combining the monetization decomposition with Theorem 4, sustaining baseline entry requires:*

$$(1 - \alpha)\rho v \leq 1 - (1 - v)^{\omega}.$$

For $v = 0.7$ and $\omega = 0.1$, this requires $(1 - \alpha)\rho \leq 0.16$.

The general constraint implies two polar cases that characterize the boundaries of viable business models.

Corollary 4 (Minimum Vibe-Coder Contribution)**.** *If all revenue is usage-dependent ($\alpha = 0$), sustaining baseline entry requires:*

$$\rho \leq \frac{1 - (1 - v)^{\omega}}{v}.$$

For $v = 0.7$ and $\omega = 0.1$, this requires $\rho \leq 0.16$, meaning vibe coders must contribute at least $1 - \rho \geq 84\%$ of what direct users contribute to monetization.

In other words, if AI-mediated users pay even a modestly discounted rate, the remaining monetization can quickly fall below the level needed to sustain entry.

Corollary 5 (Minimum Usage-Independent Revenue)**.** *If vibe coders contribute nothing to monetization ($\rho = 1$), sustaining baseline entry requires:*

$$\alpha \geq 1 - \frac{1 - (1 - v)^{\omega}}{v}.$$

For $v = 0.7$ and $\omega = 0.1$, this requires $\alpha \geq 0.84$, meaning at least 84% of revenue must come from sources independent of usage mode.

This requirement is demanding: it implies that sustaining OSS without monetizing AI-mediated usage would require a large shift toward enterprise and foundation funding.

Available evidence suggests that enterprise and foundation funding is substantial but falls well short of this level in many OSS ecosystems.[3] This gap implies that sustaining OSS under high vibe coding adoption likely requires either a major shift toward enterprise and foundation funding, or new mechanisms that monetize AI-mediated usage.

The latter is technologically feasible. AI platforms already meter usage at a fine granularity (tokens, tool calls, and package downloads), which in principle supports proportional revenue-sharing schemes. In the policy discussion below, we develop a "Spotify model" in which AI platforms compensate OSS developers based on attributable usage.

4 Calibration

The key parameters are the software share β, quality heterogeneity γ, package substitutability σ, and usage-mode substitutability θ. The vibe-coding share $v = \zeta^{\theta}/(1 + \zeta^{\theta})$ captures the intensity of the technology shock. Table 3 summarizes calibration targets.

Software share ($\beta = 1/3$). Production-function estimates from Aum and Shin (2022) and Friesenbichler et al. (2024) find that software and labor are substitutes, with labor elasticities of 0.6–0.7 and software/intangible elasticities of 0.03–0.36 in ICT firms. The intangible elasticity is roughly half the labor elasticity, so we set $\beta = 1/3$. We also check robustness at $\beta = 2/3$, a plausible upper bound given that roughly two-thirds of developer effort goes to implementation and optimization where OSS usage is most relevant (Wang and Zhang, 2017).

Quality heterogeneity ($\gamma = 3$). Data from public GitHub repositories show that stars follow a heavy-tailed distribution well approximated by a Pareto. In the model, user counts follow a Pareto with shape γ/σ. Log-rank regressions on the top 50% of repositories by rank yield a slope of 1.80 (Table 1, Column 3), which maps to $\gamma/\sigma \approx 1.8$. Combined with $\sigma \approx 1.5$, this implies $\gamma \approx 2.7$; we round to $\gamma = 3$. The slope ranges from 1.08 (full sample)

[3] We documented the composition of OSS monetization in the empirical section.

Table 3: Calibration targets and preferred parameter values.

Parameter	Role in model	Calibration target	Preferred value
β	Software share in software production	Production-function evidence on software/intangibles and software–labor substitution (Aum and Shin, 2022; Friesenbichler et al., 2024)	$1/3$
γ	Tail thickness of project quality	Pareto tail of OSS outcomes (stars, downstream dependencies)	3
σ	Substitutability across OSS packages	Demand response to software variety in video games (Clements and Ohashi, 2005)	≈ 1.5
θ	Substitutability between direct and vibe-coded usage modes	Experimental evidence on AI coding assistance adoption/productivity; mapped using option-value formula (Peng et al., 2023; Cui et al., 2025)	3 or 4
ζ	Relative productivity of vibe-coded usage vs direct usage	Experimental evidence on AI coding assistance productivity (Peng et al., 2023; Cui et al., 2025)	Implied by (v, θ)
κ	Labor cost shifter in development cost	Match level of entry (package creation)	Chosen to match entry
τ	Fixed cost of packaging/sharing OSS	Match share of projects that are shared vs unshared	Chosen to match sharing

Notes: The table maps each parameter to an empirical object used for calibration. Parameters (γ, σ) are disciplined by the heavy-tailed distribution of OSS outcomes and by external estimates of love of variety in software markets. Parameters (θ, ζ) are disciplined using experimental evidence on AI coding assistance.

to 2.24 (top 10%), so γ could plausibly lie between 1.5 and 3.5; our qualitative results are robust across this range.

Package substitutability ($\sigma \approx 1.5$). No direct estimates exist for OSS. We use Clements and Ohashi (2005), who estimate how video game variety affects console demand, finding $\sigma \approx 1.5$. OSS packages are likely less substitutable than video games, so this serves as an upper bound. A lower σ would strengthen the love-of-variety effect and amplify the welfare consequences of reduced entry.

Usage-mode substitutability ($\theta = 3$–4). Field experiments document that AI coding assistance raises productivity by 26–56% (Peng et al., 2023; Cui et al., 2025). Following the option-value identification in Halpern et al. (2015), observed adoption shares and productivity gains imply $\theta \approx 2.8$. We use $\theta = 3$ and 4 in the quantitative analysis.

Implied elasticities. At the preferred calibration ($\beta = 1/3$, $\gamma = 3$, $\sigma = 1.5$, $\theta = 3$), the two key elasticities from Remark 2 take the following values. The supply elasticity to engagement is $(1 + \beta/\sigma - \beta/\gamma)/\eta = (1 + 2/9 - 1/9)/(8/9) \approx 1.25$, indicating that a one percent decline in per-user reward reduces developer entry by more than one percent after accounting for the software-begets-software feedback. The engagement elasticity to AI, evaluated at $v = 0.5$, is $\theta v(1 - v) = 3 \times 0.25 = 0.75$: a ten percent improvement in AI capability shifts 7.5 percentage points of users from direct to vibe-coded interaction. For the demand-diversion channel to be benign, one would need either the supply elasticity to be well below one (implying most OSS supply is insensitive to engagement) or the engagement elasticity to be close to zero (implying AI does not substitute for direct interaction). The empirical evidence reviewed in Section 2—the Tailwind revenue decline, the Stack Overflow traffic collapse, and rapid AI adoption rates—points against both conditions holding simultaneously.

5 Conclusion

Open source software is a critical modern industry whose economic impact has been widely underappreciated. This sector grew rapidly over the past two decades, only to face the shock of AI-mediated software development, or "vibe coding." This paper studies what happens to this ecosystem following a technological shock that is extremely large, extremely fast, and extremely important for the sector. Recent evidence from the rapid adoption of agentic AI in coding does not yet show ecosystem collapse, but it does document the leading indicator emphasized by the model: usage can rise without commensurate human engagement.

We develop a general equilibrium model of the open source software ecosystem that captures four key economic features. First, software production exhibits strong economies of scale. Second, users value variety. Third, projects differ substantially in quality, and developers observe quality ex post, leading to selection in which projects are released. Fourth, software is an intermediate input into producing more software—a richer ecosystem of existing packages lowers the cost of building new ones, creating positive feedback. In equilibrium, user adoption and developer entry jointly determine the variety and quality of available packages.

Vibe coding enters the model as an AI-mediated mode of interacting with software that operates through two channels. The cost channel raises productivity: users can adopt packages more easily, and developers can build on existing code more efficiently. The demand-diversion channel shifts users from direct interaction—reading documentation, filing bug reports, engaging with maintainers—toward AI-mediated usage. Under traditional open source business models based on visibility and engagement, this diversion erodes the

revenue base that sustains contribution.

The model's central result is a horse race between these two channels. The critical threshold is the vibe discount ρ, which measures how much monetization falls when users switch from direct to vibe-coded interaction. When ρ exceeds the ratio of substitutability parameters σ/θ, quality declines with vibe coding adoption. Under the traditional business model where $\rho = 1$, collapse is inevitable as adoption approaches universality. The same magnification logic that made open source explode—lower costs leading to more entry, more variety, and self-reinforcing growth—operates in reverse. When monetizable demand contracts, entry falls, variety shrinks, and the resulting decline in ecosystem quality further weakens incentives for sharing.

Limitations

Before turning to policy implications, we note several limitations.

The model abstracts from team collaborations; while larger teams help mitigate negative effects, the impact is marginal. The nested logit structure implies that direct and vibe-coded interactions are substitutes in user utility—defensible because the import method is unlikely to affect use, though AI may overlook aspects of large packages given context window limitations. We treat monetization as tied to user engagement, which should be interpreted broadly to include documentation traffic, developer discovery, job signaling, and consulting leads—not only bug reports and pull requests. As the npm engagement data in Appendix C document, most packages convert fewer than one in 300 downloads into an issue. The relevant economic quantity is not the absolute level of engagement but its role as a marginal signal: even sparse interactions can be decisive for monetization when the distribution is dominated by zeros, and a modest further reduction in an already-thin flow can substantially erode the revenue base that sustains maintenance. Finally, the current version excludes network effects beyond the software-begets-software channel, though such effects could accelerate negative outcomes. It also abstracts from the fact that AI models are themselves trained on OSS code and documentation; if declining OSS provision degrades AI capability, a further feedback loop—not captured here—could amplify the negative dynamics.

The model also abstracts from a distinct enterprise software or SaaS sector. In practice, much software solves the same problem for many clients and is monetized through licensing or subscriptions, whereas the "final user" in our model is closer to a custom software producer solving one client or one internal use case. This scale difference changes both incentives and how AI-mediated use affects revenues: enterprise products can price access directly, while custom work is closer to an input-assembly problem. Consistent with this distinction, Grand View Research reports that large enterprises accounted for more than 69.0 percent of the 2022 open source services market by end user (Grand View

Research, 2022).

The model treats packages as symmetric except for quality, but in practice some packages are infrastructure (high downstream dependence, low direct visibility) while others are applications (high visibility, low dependence). Heterogeneous exposure to vibe coding across package types could generate winners and losers even when aggregate welfare falls. Quantifying the key elasticities—the substitutability between direct and vibe-coded usage, and the software-begets-software parameter—remains essential for empirical work. The rapid diffusion of AI coding tools suggests high substitutability, but revealed-preference estimation would discipline this claim.

The model treats AI mediation as unambiguously reducing direct engagement, but in practice the net effect may be more nuanced. For certain categories of interaction—particularly dependency evaluation, security auditing, and architectural review—AI tools may increase rather than decrease engagement with upstream projects. Senior developers using AI assistants may spend more time selecting and vetting dependencies, understanding framework-level design choices, and reviewing compliance and licensing requirements. These activities generate forms of interaction that do not map cleanly to a decreased-engagement story. However, these countervailing channels are likely concentrated among major, often corporate-backed libraries (e.g., React, Express, TensorFlow) where security and architectural decisions carry high stakes. For the long tail of community-maintained packages, where most interactions take the form of documentation visits and basic Q&A, the substitution channel documented in Section 2 is likely to dominate. In terms of the model, these countervailing effects would manifest as a lower effective vibe discount ρ for a subset of packages, and the alternative business model analysis in Section 3.4.3 already shows that even modest reductions in ρ can substantially relax the sustainability constraint.

Policy Implications

Our results yield an important policy message: under the traditional business model, where developer revenue depends entirely on direct user engagement, the open source ecosystem cannot survive widespread AI adoption. Sustainable provision requires $\rho < 1$—vibe-coded users must generate positive revenue, even if less than direct users.

Three policy directions emerge from the analysis.

First, platform-level revenue redistribution can internalize the externality. AI coding assistants already identify which packages they import and can attribute usage to specific libraries. Relative to the cost of LLM inference and existing telemetry, collecting audited data on which packages are loaded by vibe coding agents would be cheap and minimally invasive of user privacy. A "Spotify for open source" model—where platforms share

subscription revenue with maintainers based on usage—would reduce ρ toward zero. The infrastructure for such redistribution already exists; implementation requires coordination among AI providers rather than new technology.

Second, direct transfers to OSS infrastructure—foundation grants, corporate sponsorships, and government funding—can raise the baseline reward and compensate for engagement erosion. The sustainability challenges facing OSS infrastructure have been well-documented (Eghbal, 2016). The 2024 Open Source Software Funding Survey estimates that organizations contribute $7.7 billion annually to OSS, with employee labor as the dominant channel (Boysel et al., 2024). GitHub Sponsors provides one mechanism for directing financial support to maintainers (GitHub, 2026), and the npm ecosystem includes built-in funding signals through `funding` URLs (npm, 2022).

Third, alternative monetization channels that do not depend on direct engagement—enterprise licensing, API fees, and developer-focused services—can buffer against vibe coding. The model suggests that developer-side revenue provides partial but incomplete insurance: it reduces the rate of decline but cannot prevent collapse when $\rho = 1$.

AI-Enhanced Search and Attribution

A fourth direction deserves separate treatment because it operates through a different mechanism: rather than compensating maintainers for lost engagement, it uses AI to *create new forms of engagement* that did not previously exist. Credit assignment in open source is a notoriously difficult data and network problem. Many contributions—especially non-code work such as documentation, issue triage, community management, and architectural guidance—are currently under-recognized because they leave faint traces in the metadata that platforms surface to users. If AI systems can improve the discoverability and attribution of these contributions, they could strengthen rather than weaken the reputation-based incentives that motivate OSS supply.

Several concrete design choices could move in this direction. AI coding assistants could surface provenance information alongside generated code: when an agent assembles a solution from upstream packages, it could display which libraries were used, who maintains them, and how to support them. AI-powered search could improve discoverability of smaller, high-quality packages that currently languish in the long tail—raising π for precisely the projects most vulnerable to the demand-diversion channel. Attribution systems built into AI platforms could track not only which packages are imported but which maintainers' documentation, examples, and design patterns informed the model's training data and in-context behavior. Such fine-grained attribution is technologically feasible, since AI platforms already meter usage at the level of individual tool calls, tokens, and package loads.

Importantly, these benefits are not automatic. Under current defaults, AI mediation tends to obscure rather than surface provenance: a user who receives working code from an agent has less reason to investigate upstream dependencies than one who reads documentation directly. Realizing the attribution potential requires deliberate platform design—embedding credit, sponsorship links, and contribution histories into the surfaces where AI-mediated users interact with code. The highest-value venues for reputation and networking in open source—conferences, meetups, informal communities—remain fundamentally human and operate somewhat independently of how code is consumed. But the digital layer where most discovery now happens is increasingly AI-mediated, and designing that layer to surface rather than suppress attribution is a tractable engineering problem.

In terms of the model, improved attribution and discoverability would raise the effective reward per user π for a given level of usage, partially offsetting the vibe discount. This channel is complementary to the revenue-sharing mechanisms discussed above: revenue redistribution compensates maintainers financially, while attribution strengthens the non-pecuniary reputation and career channels that Lerner and Tirole (2002) identified as central to OSS motivation. Neither alone may suffice, but together they offer a path toward sustaining OSS in a world where most interaction is AI-mediated.

Concluding Remarks

Vibe coding represents a fundamental shift in how software is produced and consumed. The productivity gains are real and large. But so is the threat to the open source ecosystem that underpins modern software infrastructure. The model shows that these gains and threats are not independent: the same technology that lowers costs also erodes the engagement that sustains voluntary contribution. Our new empirical results do not identify the full long-run adjustment in entry, sharing, or welfare, but they do show that during the recent agentic-AI coding boom, AI-mediated adoption can increase package usage without generating proportional human attention.

The solution is not to slow AI adoption—the benefits are too large and the technology too useful. The solution is to redesign the business models and institutions that channel value back to OSS maintainers. Platform-level redistribution, direct transfers, and alternative monetization can close the gap between private and social returns. The infrastructure for these interventions largely exists; what is needed is coordination and will.

We view this paper not as a doom prophecy but as a call to action. The open source ecosystem emerged from a sequence of technological and institutional innovations—the internet, distributed version control, GitHub, package managers—that aligned private incentives with collective benefit. Vibe coding disrupts this alignment. Restoring it requires deliberate effort, but the stakes justify the cost.

References

Amodei, Dario, "Interview at Axios AI+ Summit," Axios September 2025.

Anderson, Simon P., Andre de Palma, and Jacques-Francois Thisse, *Discrete Choice Theory of Product Differentiation*, The MIT Press, October 1992.

Anthropic, "Cowork: Claude Code for the rest of your work," Claude Blog January 2026. https://claude.com/blog/cowork-research-preview.

Arkolakis, Costas, Arnaud Costinot, and Andrés Rodríguez-Clare, "New Trade Models, Same Old Gains?," *American Economic Review*, 2012, *102* (1), 94–130.

Aum, Sangmin and Yongseok Shin, "Is Software Eating the World?," *Working Paper*, May 2022, (2022-10).

Boysel, Sam, Frank Nagle, Hilary Carter, Anna Hermansen, Kevin Crosby, Jeff Luszcz, Stephanie Lincoln, Daniel Yue, Manuel Hoffmann, and Alexander Staub, "2024 Open Source Software Funding Report," Open Source Software Funding Survey November 2024. https://opensourcefundingsurvey2024.com/.

Clements, Matthew T. and Hiroshi Ohashi, "Indirect Network Effects and the Product Cycle: Video Games in the U.S., 1994-2002," *Journal of Industrial Economics*, 2005, *53* (4), 515–542.

Cui, Kevin Zheyuan, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz, "The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers," *SSRN Working Paper*, August 2025, (4945566).

Daniotti, Simone, Johannes Wachs, Xiangnan Feng, and Frank Neffke, "Who is using AI to code? Global diffusion and impact of generative AI," November 2025. arXiv:2506.08945v2 [cs.CY]. https://arxiv.org/abs/2506.08945v2 (accessed January 2026).

del Rio-Chanona, R. Maria, Nadzeya Laurentsyeva, and Johannes Wachs, "Large Language Models Reduce Public Knowledge Sharing on Online Q&A Platforms," *PNAS Nexus*, 2024, *3* (9), pgae400.

Demirer, Mert, Andrey Fradkin, Nadav Tadelis, and Sida Peng, "The Emerging Market for Intelligence: Pricing, Supply, and Demand for LLMs," Working paper December 2025. https://andreyfradkin.com/assets/LLM_Demand_12_12_2025.pdf (accessed January 2026).

Eghbal, Nadia, "Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure," Technical Report, Ford Foundation 2016.

Friesenbichler, Klaus S., Agnes Kügler, and Julia Schieber-Knöbl, "Intangible Capital as a Production Factor: Firm-level Evidence from Austrian Microdata," *Applied Economics*, 2024, *56* (23), 2687–2701.

Gans, Joshua S., "Will User-Contributed AI Training Data Eat Its Own Tail?," NBER Working Paper 32686, National Bureau of Economic Research July 2024.

GitHub, "Innovation Graph: Git Pushes," GitHub Innovation Graph 2025. Global metrics on git push activity to public repositories.

_ , "GitHub Sponsors," GitHub 2026. `https://github.com/sponsors` (accessed January 2026).

Gousios, Georgios, "The GHTorrent Dataset and Tool Suite," in "Proceedings of the 10th Working Conference on Mining Software Repositories" MSR '13 IEEE Press 2013, pp. 233–236.

Grand View Research, "Open Source Services Market Size & Share Report, 2030," Grand View Research 2022. Report ID: GVR-4-68040-059-2. End-user Insights section states: "The large enterprises segment dominated the market in 2022 and accounted for a revenue share of more than 69.0%.".

Halpern, László, Miklós Koren, and Ádám Szeidl, "Imported Inputs and Productivity," *American Economic Review*, 2015, *105* (12), 3660–3703.

Hoffmann, Manuel, Frank Nagle, and Yanuo Zhou, "The Value of Open Source Software," Working Paper 24-038, Harvard Business School 2024.

Huang, Ruanqianqian, Avery Reyna, Sorin Lerner, Haijun Xia, and Brian Hempel, "Professional Software Developers Don't Vibe, They Control: AI Agent Use for Coding in 2025," December 2025. arXiv:2512.14012v1 [cs.SE]. `https://arxiv.org/abs/2512.14012v1` (accessed January 2026).

Jimenez, Carlos E., John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan, "SWE-bench: Can Language Models Resolve Real-world Github Issues?," in "International Conference on Learning Representations" January 2024. OpenReview. `https://openreview.net/forum?id=VTF8yNQM66` (accessed January 2026).

Jones, Derek M., *Evidence-based Software Engineering: based on the publicly available data*, Knowledge Software, Ltd, November 2020. Version 1.0. Accessed January 2026.

Kleinberg, Jon and Manish Raghavan, "Algorithmic Monoculture and Social Welfare," *Proceedings of the National Academy of Sciences*, 2021, *118* (22), e2018340118.

Koren, Miklós, Zsófia Bárány, and Ulrich Wohak, "Putting the Direction Back into Directed Technical Change: Digitization, Artificial Intelligence, and Alignment in High Dimensions," Working paper, Central European University 2026. Manuscript.

Krugman, Paul, "Scale Economies, Product Differentiation, and the Pattern of Trade," *American Economic Review*, 1980, *70* (5), 950–959.

Lerner, Josh and Jean Tirole, "Some Simple Economics of Open Source," *Journal of Industrial Economics*, 2002, *50* (2), 197–234.

Libraries.io, "Open Source Repository and Dependency Database," Libraries.io 2025. Accessed January 2026.

Melitz, Marc J., "The Impact of Trade on Intra-Industry Reallocations and Aggregate Industry Productivity," *Econometrica*, 2003, *71* (6), 1695–1725.

npm, "npm-fund," npm Documentation October 2022. `https://docs.npmjs.com/cli/v10/commands/npm-fund` (last edited October 5, 2022; accessed January 2026).

Orosz, Gergely, "Stack Overflow Is Almost Dead," The Pragmatic Engineer May 2025.

Peng, Sida, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer, "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot," *arXiv preprint*, February 2023.

Pichai, Sundar, "Alphabet Q3 2024 Earnings Call," Alphabet Investor Relations October 2024. `https://abc.xyz/investor/`.

Stenberg, Daniel, "Death by a thousand slops," Blog post July 2025. curl maintainer describing AI-generated low-quality security reports.

Sun, Keh-Kuan, "When AI Improves Answers but Slows Knowledge Creation: Matching and Dynamic Knowledge Creation in Digital Public Goods," Preprint arXiv:2604.00468, arXiv April 2026.

Wang, Yong and Jing Zhang, "The Effort Distribution of Software Development Phases," *Computer Science and Application*, 2017, *7* (5), 428–437.

Wathan, Adam, "Comment on pull request "feat: add llms.txt endpoint for LLM-optimized documentation #2388"," GitHub January 2026. `https://github.com/tailwindlabs/tailwindcss.com/pull/2388#issuecomment-3717222957`.

A Proofs

This appendix collects proofs for results stated in the main text.

A.1 Proof of Proposition 1

Proof. Conditions (2)–(4) of Definition 1 imply closed-form expressions for the selection cutoff q_0, average quality $\bar{q}$, and the mass of shared packages m_s.

First, the sharing indifference condition at the cutoff is

$$\Pi(q_0) = \frac{\pi q_0^{\sigma}}{m_s \bar{q}^{\sigma}} = \tau.$$

Using (3) $\bar{q} = \Lambda q_0$ and (4) $m_s = m q_0^{-\gamma}$, we obtain

$$\frac{\pi q_0^{\sigma}}{m q_0^{-\gamma} (\Lambda q_0)^{\sigma}} = \tau \quad \Longleftrightarrow \quad \frac{\pi}{m \Lambda^{\sigma}} q_0^{\gamma} = \tau,$$

so

$$q_0 = \left(\frac{\tau m \Lambda^{\sigma}}{\pi} \right)^{1/\gamma},$$

which is (2). Multiplying by Λ gives (3).

Next,

$$m_s = m q_0^{-\gamma} = m \left(\frac{\tau m \Lambda^{\sigma}}{\pi} \right)^{-1} = \frac{\pi}{\tau \Lambda^{\sigma}},$$

which is (4).

Finally, free entry requires expected net payoff to equal the up-front development cost Φ:

$$\mathbb{E}[\max\{\Pi(q) - \tau, 0\}] = \Phi.$$

Using the Pareto tail and $\Pi(q) = \tau (q/q_0)^{\sigma}$, the expected payoff evaluates to

$$\mathbb{E}[\max\{\Pi(q) - \tau, 0\}] = \frac{\sigma}{\gamma - \sigma} \cdot \frac{\pi}{m \Lambda^{\sigma}}.$$

Substituting the cost function $\Phi = \kappa^{1-\beta} u^{-\beta} \Lambda^{-\beta\sigma/\gamma} m^{-\beta/\gamma} \left(\frac{\tau}{\pi}\right)^{\beta/\sigma - \beta/\gamma}$ and rearranging yields (5).

Since $\eta \equiv 1 - \beta/\gamma > 0$, the mapping $m \mapsto m^{\eta}$ is strictly increasing on $\mathbb{R}_+$, so (5) pins down a unique $m > 0$. Given m, equations (2)–(4) uniquely determine $(q_0, \bar{q}, m_s)$. Hence a unique baseline equilibrium exists. □

A.2 Proof of Theorem 1

Proof. Consider the planner problem in Definition 2. Using the equilibrium relationships from Corollary 2 and the development cost from the proof of Proposition 1, welfare and

total development cost take the form

$$\mathcal{U}(m) = A_U\, m^{1/\gamma}, \qquad m\,\Phi(m) = A_\Phi\, m^{\eta},$$

with constants $A_U, A_\Phi > 0$ and $\eta = 1 - \beta/\gamma$. The planner's objective is $W(m) = A_U m^{1/\gamma} - A_\Phi m^{\eta}$, with derivative

$$W'(m) = m^{\eta-1}\left[\frac{A_U}{\gamma}\, m^{1/\gamma-\eta} - \eta A_\Phi\right].$$

The assumption $\gamma > 1 + \beta$ is equivalent to $1/\gamma < \eta$, so the bracketed term is strictly decreasing in m, positive as $m \to 0$ and negative as $m \to \infty$. Hence W' crosses zero exactly once, from above, and the crossing point is the unique interior maximizer m^{fb}. It follows that $m^{eq} < m^{fb}$ if and only if $W'(m^{eq}) > 0$.

In equilibrium, free entry gives $\Phi(m^{eq}) = \frac{\sigma}{\gamma-\sigma} \cdot \frac{\pi}{m^{eq}\Lambda^\sigma}$. Since $\Lambda^\sigma = \gamma/(\gamma - \sigma)$, total development cost in equilibrium is

$$m^{eq}\,\Phi(m^{eq}) = \frac{\sigma\pi}{\gamma}.$$

Evaluating the planner's marginal condition at m^{eq}:

$$W'(m^{eq}) > 0 \iff \frac{\mathcal{U}(m^{eq})}{\gamma} > \eta\, m^{eq}\Phi(m^{eq}) = \frac{\eta\sigma\pi}{\gamma} \iff \pi < \frac{\mathcal{U}^{eq}}{\sigma\eta},$$

which is exactly the condition of the theorem. Hence $m^{eq} < m^{fb}$: there are too few entrants and, by $m_s = mq_0^{-\gamma}$ evaluated at the planner's larger m, too few shared OSS packages in equilibrium. □

A.3 Proof of Theorem 4

Proof. From the reduced-form free entry condition (5), holding (κ, τ) fixed,

$$m^{\eta} \propto u^{\beta}\,\pi^{a_\pi}, \qquad \eta = 1 - \beta/\gamma, \quad a_\pi = 1 + \beta/\sigma - \beta/\gamma.$$

Taking ratios relative to the baseline economy ($v = 0$, $u_0 = 1$, monetization π_0) and substituting the productivity shifter $u = (1 - v)^{-1/\theta}$:

$$\left(\frac{m}{m_0}\right)^{\eta} = (1 - v)^{-\beta/\theta}\left(\frac{\pi}{\pi_0}\right)^{a_\pi}.$$

Since $\eta > 0$, sustaining baseline entry ($m \geq m_0$) is equivalent to

$$\left(\frac{\pi}{\pi_0}\right)^{a_\pi} \geq (1 - v)^{\beta/\theta} \iff \frac{\pi}{\pi_0} \geq (1 - v)^{(\beta/\theta)/a_\pi} = (1 - v)^{\omega},$$

where

$$\omega = \frac{\beta/\theta}{1+\beta/\sigma-\beta/\gamma} = \frac{1/\theta}{1/\beta+1/\sigma-1/\gamma}.$$

This is (23). Finally, $\omega \in (0,1)$: positivity is immediate, and $\omega < 1$ follows because $1/\theta < 1 < 1/\beta < 1/\beta + 1/\sigma - 1/\gamma$, where the first inequality uses $\theta > 1$, the second $\beta < 1$, and the last $\sigma < \gamma$ (Assumption 1). □

B Experimental Implementation

The main text analyzes the direct dependencies selected from matched website requirements. This appendix records the two prompts that produce those selections and a representative product requirements document (PRD). The frozen experiment artifacts retain the full prompts, model outputs, `PLAN.md`, `package.json`, and `package-lock.json` for every completed run.

B.1 PRD Creation

For each sampled website, the PRD-generation model inspected the public website and was instructed to describe functionality rather than implementation:

> Write a Product Requirements Document for a web development project. Describe what the website does and the features it needs, not how it should be built. Do not name a technology, framework, library, tool, website domain, brand, hosting platform, database, or programming language. Use only functional requirements, user experience, and business logic. Limit the document to 650 words and organize it under Purpose, Target Users, Core Features, UI/UX Requirements, Data Requirements, and Performance Requirements.

The prompt also required the output to begin with "Product Requirements Document" and forbade preamble or commentary. A separate validation stage rejected or regenerated briefs that violated the technology-neutrality restriction.

B.2 Website-Building Prompt

Each coding model received the same validated PRD for a task. It was instructed:

> Read the specification, write a brief plan in `PLAN.md`, run `npm init -y`, and install all npm packages needed for the project. Stop after installing packages; do not write code or create source files.

We measure direct dependencies in the resulting `package.json`; packages appearing only in `package-lock.json` are transitive installations and not model choices.

B.3 Representative Product Requirements Document

The following is the full frozen PRD 008, typeset from `prd_008.md` in the experiment stimuli directory.

Purpose. The site serves as a comprehensive digital platform for industrial manufacturing solutions, enabling businesses to discover products, access technical resources, learn about automation capabilities, and connect with consultants for manufacturing operations optimization.

Target Users.

- Industrial engineers and operations managers seeking automation solutions.
- Maintenance teams managing factory equipment and systems.
- Manufacturing executives exploring digital transformation strategies.
- OEMs (original equipment manufacturers) designing production machinery.
- Plant managers overseeing production facilities across multiple industries (oil & gas, mining, food processing, automotive, pharmaceuticals).
- System integrators implementing industrial control systems.
- Technical personnel requiring documentation, training, and support resources.

Core Features. Product discovery and catalog:

- Searchable hardware catalog with 20+ categories (controllers, sensors, drives, safety devices, motion control, etc.).
- Software solution browsing organized by use case (design, operations, maintenance, analytics).
- Product comparison and specification viewing.
- New product announcements and spotlight sections.
- Related product recommendations based on user activity.

Regional localization:

- Geographic region selector with country/territory-specific content.
- Multi-language support (20+ languages including English, Spanish, French, German, Chinese, Japanese).
- Regionalized product availability and pricing information.

- Local contact and sales office information.

Resource library:

- Technical documentation and literature downloads.
- Knowledge base with searchable answers to product questions.
- Training materials and workforce development resources.
- Case studies demonstrating industry-specific implementations.
- Blog articles on industry trends, best practices, and technical topics.
- Webinar registration (live and on-demand).
- White papers and research reports.

Industry-specific content:

- Tailored solution pages for key industries (energy, mining, food & beverage, automotive, pharmaceuticals).
- Industry-specific case studies with measurable outcomes.
- Sector-specific challenges and solution approaches.
- Regulatory compliance information by industry.

Consultation and support:

- Sales consultant contact forms and office locators.
- Live support access through knowledge base.
- Product download compatibility checker.
- Installation and troubleshooting guides.
- Software ordering and subscription management.

Content personalization:

- Recently viewed products tracking.
- Preferred products list.
- Personalized recommendations based on browsing history.
- Role-based content filtering (operator, manager, executive perspectives).

News and updates:

- Press release distribution.
- Company announcements.

- Product launch notifications.
- Industry recognition and awards.
- Sustainability reporting.

UI/UX Requirements. **Navigation and architecture:**

- Multi-level navigation supporting complex product hierarchies.
- Quick links to frequently accessed resources (downloads, knowledge base, training).
- Breadcrumb navigation for deep content structures.
- Persistent search functionality across all pages.
- Tabbed content organization for related information.

Visual design:

- Professional industrial aesthetic with emphasis on clarity and accessibility.
- High-quality product photography showing equipment from multiple angles.
- Infographics and diagrams explaining complex technical concepts.
- Video content embedded for product demonstrations and customer testimonials.
- Icon-based category navigation for quick recognition.

Responsive behavior:

- Full mobile responsiveness for field access by technicians.
- Tablet optimization for plant floor use.
- Desktop experience optimized for detailed specification review.
- Touch-friendly interfaces for industrial touchscreen devices.
- Offline-capable documentation viewing.

Interactive elements:

- Filterable product catalogs with real-time results.
- Dynamic content loading based on user selections.
- Expandable/collapsible sections for detailed specifications.
- Interactive comparison tools for products.
- Gated content requiring user registration for premium resources.
- Cookie preference management for compliance.

Data Requirements. **Product information:**

- Detailed specifications (electrical ratings, dimensions, certifications).
- Compatibility matrices between hardware and software.
- Product lifecycle status (new, active, end-of-life).
- Certification and compliance documentation.
- Installation requirements and environmental ratings.
- Pricing and availability by region.

Customer data:

- User preferences and consent management.
- Browsing history and product views.
- Download tracking and resource access logs.
- Geographic location for content personalization.
- Industry and role information for targeted content.

Content management:

- Multi-language content versions.
- Publication dates and content freshness indicators.
- Author attribution for blog posts and technical articles.
- Taxonomy for organizing 1000+ products and resources.
- Version control for technical documentation.

Support and knowledge:

- Searchable question-answer database.
- Troubleshooting decision trees.
- Software version compatibility data.
- Training course catalogs and schedules.
- Case study metrics and outcomes.

Performance Requirements. **Scale:**

- Support for global user base across 100+ countries.
- Catalog of 1000+ hardware products and multiple software suites.

- Extensive documentation library (thousands of PDFs and technical guides).
- High-resolution product imagery and video content.
- Multi-region content delivery for fast load times worldwide.

Search performance:

- Sub-second search results across product catalog.
- Faceted filtering with instant results.
- Relevance ranking based on user context.
- Suggested search terms and auto-complete.

Availability:

- 24/7 accessibility for global manufacturing operations.
- Minimal downtime for maintenance.
- Redundancy for critical support resources.
- Fast content delivery regardless of user location.

Security and compliance:

- Secure handling of user registration data.
- Cookie consent management compliant with regional regulations.
- Protection of proprietary product information.
- Secure document download tracking.
- Privacy policy enforcement across all regions.

C Baseline Engagement in npm Packages

Table 4 reports GitHub activity for 25 npm packages with weekly downloads between 1,000 and 100,000. Each package is classified as either a *developer tool*—directly installed and used by developers (e.g., scraping frameworks, CLI utilities, desktop automation)—or *infrastructure*—typically pulled in as a transitive dependency or embedded in build pipelines. Stars and other interactions (forks, issues, and pull requests) are expressed as a percentage of all-time npm downloads.

Two patterns stand out. First, engagement is sparse across the board: the median stars-to-downloads ratio is 0.06 percent and the median ratio for other interactions is 0.01 percent. Second, the gap between developer tools and infrastructure packages is striking. Developer tools attract roughly 1–2 orders of magnitude more stars per download than

infrastructure packages at comparable download levels. The median stars-to-downloads ratio for tools is 1.33 percent versus 0.01 percent for infrastructure; for other interactions the corresponding figures are 0.24 percent versus 0.01 percent.

Figure 7 visualizes this separation. Both panels plot GitHub engagement against total downloads on log–log axes, with OLS trend lines fitted separately by package type. In Panel (a), the slope of the tool trend (0.71) indicates that stars scale roughly as the square root of downloads for developer tools, while the near-zero infrastructure slope (0.14) confirms that downloads and stars are nearly uncorrelated for transitive dependencies. Panel (b) shows a similar pattern for other interactions (forks, issues, and pull requests combined), with slopes of 0.58 and 0.02 respectively.

Table 4: GitHub engagement as a share of total npm downloads, by package type.

Package	Type	Total DLs	Stars	Stars %	Other	Other %
Web scraping & screenshots						
x-ray	tool	85K	5,905	6.95	745	0.877
scrape-it	tool	182K	4,071	2.24	427	0.235
capture-website	tool	301K	2,015	0.670	267	0.089
crawlee	tool	1.9M	22,510	1.17	4,431	0.231
website-scraper	infra	462K	1,701	0.368	901	0.195
node-html-to-image	infra	1.4M	876	0.064	348	0.026
quickchart-js	infra	1.9M	74	0.004	65	0.003
CLI & terminal						
terminal-image	tool	528K	1,075	0.204	77	0.015
pastel	infra	524K	2,361	0.450	126	0.024
ascii-table3	infra	594K	35	0.006	31	0.005
chalk-animation	infra	1.5M	2,181	0.147	109	0.007
neo-blessed	infra	957K	400	0.042	91	0.010
NLP & machine learning						
wordnet	tool	17K	53	0.308	25	0.145
algebrite	tool	75K	997	1.33	238	0.317
brain.js	tool	118K	14,800	12.5	2,000	1.69
langdetect	infra	577K	24	0.004	16	0.003
ml-regression	infra	1.1M	47	0.004	45	0.004
wink-nlp	infra	2.3M	1,360	0.060	150	0.007
System & dev tools						
cost-of-modules	tool	144K	2,867	1.99	112	0.078
active-win	tool	89K	893	1.00	398	0.446
robotjs	tool	480K	12,723	2.65	1,771	0.369
tiged	tool	795K	441	0.055	190	0.024
text-readability	infra	247K	82	0.033	31	0.013
network-interfaces	infra	1.2M	2	0.000	1	0.000
tasklist	infra	1.5M	71	0.005	50	0.003

Notes: Each cell in the % columns reports (cumulative count / total all-time npm downloads) $\times$ 100. "Other" aggregates forks, issues (open and closed), and pull requests (open and closed). Packages are classified as *tool* (directly installed and used by developers) or *infra* (typically a transitive dependency). Data collected March 26, 2026. Sources: npm downloads API and GitHub REST/Search APIs.

(a) Stars vs total downloads

(b) Other interactions vs total downloads

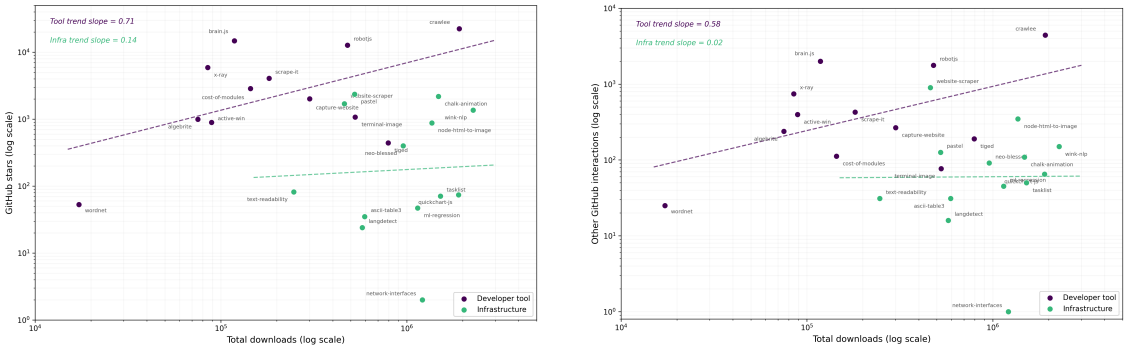


Figure 7: GitHub engagement vs npm downloads by package type.

Notes: Both panels plot GitHub engagement against total npm downloads on log–log axes for 25 packages with weekly downloads between 1,000 and 100,000. Packages are classified as "developer tool" (directly installed by developers) or "infrastructure" (typically pulled in as transitive dependencies). Dashed lines show OLS trend fits in log–log space. The vertical separation between the two trend lines indicates that developer tools accumulate roughly 1–2 orders of magnitude more engagement per download than infrastructure packages at comparable download levels. "Other interactions" aggregates forks, issues (open and closed), and pull requests (open and closed). Data collected March 26, 2026. Sources: npm downloads API and GitHub REST/Search APIs.