Screening for Engineering Talent

Gábor Nyéki[*]

Click here for the latest version.

Abstract. Software firms hire engineers whose latent ability affects firm output quality. Hiring high-ability engineers is difficult because accurate screening devices are costly. However, if high-ability engineers learn difficult programming languages more easily than low-ability engineers, then teams within a firm could recruit them by adopting a difficult language. I build a signaling model of hiring and production, and show that in a separating equilibrium, (i) difficult languages are chosen by smaller teams, and (ii) among the smallest teams, high-effort languages are chosen in more skill-intensive problem domains. I show evidence for the model in two data sets. First, using data from the question-and-answer platform *Stack Overflow,* interest in C++, a difficult mainstream language, predicts better answers about other languages and interest in transferable skills like engineering practices, mathematics, and statistics. Second, looking at a near-universe of U.S. job postings between 2010 and 2024 from *Lightcast,* C++ is more concentrated in technical problem domains such as securities trading, biomedical engineering, and artificial intelligence, than less difficult mainstream languages like Java or C#. Results for difficult niche languages are more mixed.

Date: November 5, 2024.

[*]Princeton University; e-mail: gnyeki@princeton.edu.

"...if a company chooses to write its software in a comparatively esoteric language, they'll be able to hire better programmers, because they'll attract only those who cared enough to learn it."

—Paul Graham (2004)

"However ...one thing should be obvious: C++ cannot be ideal for all of that."

—Bjarne Stroustrup (2007)

1. Introduction

Worker ability plays a crucial role in economic development as production becomes more complex (Kremer, 1993). However, frictions in the economy, including limited information about worker ability, result in misallocation of talent (Pallais, 2014; Abel et al., 2020; Carranza et al., 2022) which has large consequences for economic growth (Murphy et al., 1991; Hsieh et al., 2019). Mirroring these frictions, firms report making significant effort to select high-ability job applicants, especially for jobs where monitoring individual performance is harder (Huang and Cappelli, 2010). In such an environment, high-ability workers and firms both benefit from lower-cost screening devices.

This paper studies screening in the labor market for software engineers. Software engineers perform complex cognitive work, and they typically do so in teams, where their individual performance is difficult to measure. As a result, higher-paying U.S. software firms are known for conducting over several rounds of "behavioral" and "coding" interviews in the hiring process, and some also operate well-paid internships programs. While the costs of such search procedures are high, in this setting a firm's technology choice, namely the programming language that it chooses, can serve as an alternative or complementary screening device.

The idea that language choice can act as a screening device is motivated by anecdotal evidence in the software industry. Early internet entrepreneur and later venture capitalist Paul Graham (2004) famously described the "Python paradox": the apparent superiority of engineers who learned Python, an unpopular language at the time. While Python has become widely adopted in industry and education since then, engineers who wanted to learn it in 2004 had to engage in self-study, and Google, a pioneering start-up, was one of its early adopters. Although the existence of the Python paradox has been a matter of debate, similar views have been voiced by others over the years.

To formalize this idea, I propose a signaling model of labor-market screening in which latent engineer ability enters the production function multiplicatively, as in Kremer's (1993) O-ring theory. Firms can choose between two languages: a low-effort language that is well-known among engineers, and a high-effort language that requires self-study. I show numerically that the high-effort language can only be used to signal high ability (i) if the teams that choose the high-effort language are smaller,[1] and (ii) if they operate in problem domains that are more skill-intensive, i.e., where engineers' latent ability affects productivity more strongly.

[1]More precisely, the parameter region in which a separating equilibrium is possible shrinks with team size. Thus if teams are larger, then all teams choose the low-effort language in a pooling equilibrium. However, a pooling equilibrium is always possible, regardless of team size.

Larger teams cannot choose the high-effort language because the market wage premium for the language decreases in team size, eliminating high-ability engineers' incentive to self-select. Moreover, if the domain is sufficiently skill-intensive, then a small team may choose the high-effort language even if it is a worse fit for that domain than the low-effort language.

I show evidence in two data sets that high ability could be signaled by high-effort languages. First, I use data from the question-and-answer site *Stack Overflow,* and I look at (a) whether knowledge of high-effort languages predicts higher-quality answers in low-effort languages for the same individual, and (b) whether knowledge of high-effort languages predicts activity on other question-and-answer sites that specialize in software engineering, mathematics, and statistics and machine learning. Second, I use *Lightcast's* near-universe of U.S. job postings between 2010 and 2024, and look at whether the use of high-effort languages is more concentrated in technical problem domains than the use of low-effort languages.

I find that knowledge of C++, one of the oldest mainstream languages and a high-effort language, *instead of* another mainstream language such as Java, C#, or Python, predicts both higher-quality answers in unrelated mainstream languages and more activity on other question-and-answer sites that cover transferable technical skills. Moreover, a similar, albeit less consistent, pattern emerges for the niche high-effort languages Scala, Clojure, F#, and Haskell. However, the knowledge of an *additional* programming language almost always predicts higher-quality answers and interest in transferable skills, whether or not the additional language is high-effort.

In the job postings data, I find more differences between C++ and niche high-effort languages. Estimates of relative domain concentration show that C++ is more concentrated in securities trading, artificial intelligence, and biomedical engineering, than Java or C# is. At the same time, niche high-effort languages show a more mixed pattern. This points to the uniqueness of C++. The language, as I describe in more detail in Section 2, has accumulated more complexity than other languages due to its continuous evolution over 45 years and its strong commitment to backwards compatibility. Thus to use it well, an engineer needs to exert more effort in self-study than for other languages.

This paper contributes to the literature on limited information in the labor market. Asymmetric information about worker ability has been shown to introduce significant matching frictions between workers and firms (Aigner and Cain, 1977). Several ways to reduce these frictions have been proposed, including formal education (Spence, 1973, 2002), skill certification services (Abebe et al., 2021a; Carranza et al., 2022), and the provision of references (Pallais, 2014; Abel et al., 2020) which can all serve as signals of latent ability, workshops that teach job search skills (Wheeler et al., 2022), and optimal payment schemes to attract high-ability applicants (Delfgaauw and Dur, 2007; Abebe et al., 2021b). I contribute by showing that some engineering firms may have alternate means of selecting high-ability workers, namely via the endogenous choice of their production functions.

The paper also contributes to the literature on high-skill non-routine workers, specifically software engineers. The measurement of individual worker output has been of great interest in economics. Much of the literature, especially early on, has focused on low-skill (Hamilton et al., 2003; Bandiera et al., 2007; Mas and Moretti, 2009) or routine workers (Fenizia, 2022). Three notable exceptions are studies of teacher effectiveness (e.g., Jackson and Bruegmann, 2009; Papay et al., 2020) where student outcomes serve as the measure of teacher output, studies of medical doctors (e.g., Chen, 2021) where patient mortality is one of the key measures, and studies of academics and inventors (e.g., Jaffe et al., 1993, 2000; Azoulay et al., 2010;

Jaravel et al., 2018) where publications, patents, and citations are the measures of output. A more recent literature has studied software engineers and other workers in the technology sector (Emanuel et al., 2023; Gibbs et al., 2023). I contribute to this literature by breaking down the composition of skills that software engineers bring to firm production.

The remainder of the paper continues as follows. Section 2 describes computer scientists' views on programming languages and compares some of the major languages in use today. Section 3 introduces a signaling model of language choice. Section 4 presents results on predictors of engineer skill in the Stack Overflow data, and Section 5 on job postings by problem domain in the Lightcast data. Section 6 concludes.

2. Background

This section provides an overview of the roles that programming languages play according to computer scientists, compares some of the languages that are in relatively common use today, and explains what makes them high- or low-effort and high- or low-fitness in a particular problem domain.

2.1. Why programming languages exist. A programming language solves a constrained optimization problem. The objective is a combination of two goals. The first is to encode algorithms unambiguously so that computers can execute them efficiently. The second is to serve as a "tool of thought" (Iverson, 1980), a precise notation that enables engineers to reason about the correctness and efficiency of algorithms and to communicate their reasoning to each other. While balancing between these two goals already involves fundamental trade-offs,[2] language designers are also constrained by hardware limitations, backwards compatibility with existing technology, and their current knowledge about the usefulness and pitfalls of language features and how to implement them efficiently.

By the early 1970s, navigating this optimization problem had already given rise to dozens if not hundreds of programming languages (Landin, 1966; Sammet, 1972). Since then, many more experimental languages have been created as testbeds of novel concepts, both by academics and by engineers working in industry. Ultimately, those concepts that proved to be practical in experimental languages have slowly made their way into the mainstream.

2.2. Mainstream low-effort languages. Mainstream languages differ from experimental ones in that they prioritize familiarity.[3] Familiarity is especially important for larger organizations with long-running projects, where new and often early-career engineers must be able to understand and contribute to existing code bases (Pike, 2012).[4] The long-term maintainability that a familiar language supports is so important

[2]Every programming language makes a trade-off between the efficiency of executed code, the ease of writing, understanding, and extending code, and the ability to ensure the correctness of algorithms by finding or fully preventing certain kinds of mistakes. Horowitz (1983, p. 34) writes: "Programmers must design and implement a software system. They must document it for themselves and for others. They must produce a product which is free of errors and which performs efficiently. [...] Thus the programmer's tasks put demands on a language which are at the same time diverse and mutually antagonistic. This makes the design of a programming language an exciting challenge and a field where only a select few succeed."

[3]Hoare (1974) emphasizes the importance of complete understanding: "A necessary condition [...] is the utmost simplicity in the design of the language. [...] A programmer who fully understands his language can tackle more complex tasks, and complete them more quickly and more satisfactorily, than if he did not. In fact, a programmer's need for an understanding of his language is so great that it is almost impossible to persuade him to change to a new one."

[4]Pike (2012), one of the designers of Google's Go programming language, writes: "Programmers working at Google are early in their careers and are most familiar with procedural languages, particularly from the C family. The need to get programmers productive quickly in a new language means that the language cannot be too radical."

that focus on it has been described as a characteristic that sets "software engineering" apart from "computer programming" (Winters et al., 2020).

The most widely familiar languages are those that engineers see in their early education. Over the past couple decades, Java and Python have become by far the most widely known languages, and the majority of computer science graduates can be expected to have a working proficiency of both. A 2014 survey of computer science programs at 39 U.S. universities found that 27 used Python and 22 used Java in its introductory courses (Guo, 2014). While these are two relatively low-effort languages, almost a third of the surveyed departments also used C or C++, two high-effort programming languages.

2.3. C++, a mainstream high-effort language. C, a language that first appeared in 1973 to make the development of operating systems easier, is typically seen as "close to the machine" (Ritchie, 1996).[5] It has had a vast influence on languages that came later. Most prominently among them, C++ has become known for its high performance and memory efficiency. However, through its 45-year existence, C++ has committed to near-perfect compatibility with C and its associated infrastructure, which has limited the design space within which C++ can operate.

Programming languages are solutions to constrained optimization problems, and as constraints change over time, optimal language designs change as well. For example, computers in common use in industry, government, and scientific computing during the 1970s and 1980s had a much smaller memory capacity than the computers of today. The PDP-11, a popular computer that was used in, e.g., air traffic control, physics experiments, and the control of nuclear power plants, could not handle more than 64 KB of memory (or 4 MB through extension mechanisms). By contrast, an entry-level smartphone in 2024 has a memory capacity of 4 GB. This affected not only the design of applications but also the design of programming languages like C and, by virtue of backwards compatibility, C++, and the necessary tooling infrastructure around them.

C++ could not have become as popular as it has without strict backwards compatibility. Today, it is used in all major web browsers, the iOS and Android mobile operating systems, Microsoft Office, Adobe Photoshop, and various other desktop applications, network infrastructure, scientific computing libraries, and so on. However, maintaining compatibility has forced the its original creator, Bjarne Stroustrup, and later the ISO standardization committee, to evolve C++ in a way that has improved the language but, at the same time, is suboptimal under modern constraints.[6] As a result, C++ has accumulated more complexity than newer languages, in the form of exceptions to rules that engineers need to be aware of. Even before the 1998 publication of the first standard, Stroustrup (1996) already noted that the "cost of language complexity has been considerable but manageable." Since the first major revision to the standard in 2011, the language has evolved through revisions every three years.

Keeping up with new language features, new best practices, and the deprecation of old practices, has

[5] Linus Torvalds is the lead developer of the Linux kernel which runs, among other things, on all Chromebooks and Android devices, and most WiFi routers. In his June 2012 talk at Aalto University, he said: "When I read C, I know what the assembly language will look like."

[6] The commitment to backwards compatibility made some new features impossible. E.g., type safety was broken in order to remain compatible with C by allowing calls to undeclared functions (Stroustrup, 1996, p. 10) and implicit type conversions from `int` to `char` and so on (Stroustrup, 1996, p. 11), and memory may not be handled correctly when objects are copied because a copy constructor for a class can only be required if the class also has a destructor (Stroustrup, 2007, p. 49). Compatibility made some other features possible only in an incomplete form. E.g., the compiler must assume that global variables that are declared constant in one compilation unit remain constant within other compilation units, but in order to remain compatible with the design of 1979 tooling infrastructure, it cannot confirm nor force this (Stroustrup, 1996, p. 26).

become a problem.[7] Although Stroustrup (2020) wrote that by 2020 most engineering schools worldwide were teaching C++, he pointed out two issues in particular. First, university programs tended to give students an outdated introduction to the language, not keeping up with the newer standards. Second, they typically only gave a superficial introduction, without developing a deeper understanding of how to use the language well.

In this environment, engineers who wish to learn C++ need to exert significant effort outside of formal classroom instruction. Those who engage in self-study can participate in annual regional conferences in North America, Europe, and Asia where speakers give talks to industry practitioners about the latest features, changing best practices, and so-called "footguns" to avoid. Outside of conferences, dozens of books have also been published on the language, often with new editions after each three-year revision to the standard. Yet the impression remains that C++ is difficult to understand. For example, the 4th edition of the book *Professional C++* described the language as this: "C++ is notoriously complex, and whether you use it for gaming or business, maximizing its functionality means keeping up to date with the latest changes."

Complexity has made the language not only hard to use but also easy to misuse. In 2019, Microsoft and Google have both estimated that around 70 percent of all security vulnerabilities in their code bases are due to memory management bugs which, among modern languages in widespread use, only C and C++ are prone to. Google has decided to switch from C++ to Rust, a newer memory-safe language, when writing new code in its Android operating system, and they have subsequently reported a decline in memory-related vulnerabilities. Microsoft has announced the adoption of Rust in the Windows code base in 2023. Motivated by the same security concerns, the U.S. federal government has also called for the adoption of memory-safe languages (The White House, 2024). Many among the designers of C++ have also taken note. Most prominently, in 2024, the chair of the ISO C++ standards committee summarized the numerous obstacles facing an engineer who wants to enforce safety guarantees in program code, in a keynote talk (Sutter, 2024, slide 37).

2.4. Other high-effort languages. C++ is not only a high-effort language, it is also one of the oldest mainstream languages currently in use. However, many niche high-effort languages have appeared over the past decades. Despite the familiarity of mainstream languages, some organizations adopt niche languages because of their perceived benefits. For example, Jane Street, a securities trading firm, has been a prominent user of OCaml. A book co-authored by its head of technology describes OCaml's strengths compared with mainstream languages, and writes that when mainstream languages adopt its features, they usually do so in a limited form:

> *"Despite their importance, these ideas have made only limited inroads into mainstream languages, and when they do arrive there, like first-class functions in C# or parametric polymorphism in Java, it's typically in a limited and awkward form. The only languages that completely embody these ideas are statically typed, functional programming languages like OCaml, F#, Haskell, Scala, Rust, and Standard ML." (Madhavapeddy and Minsky, 2022)*

Table 14 shows some basic characteristics of the programming languages that this paper studies. Several

[7]Another manifestation of language complexity is that the grammar of C++ is highly irregular. This makes C++ code more difficult to parse and analyze than code written in other languages, which has stymied the development of tools and the engagement of academic researchers (Stroustrup, 2007, 2020).

niche languages maintain interoperability with Java or C#, two low-effort mainstream languages. Java and C# are notable in that they represent not only two mainstream languages but also their associated "runtimes." These runtimes serve two purposes: (i) they enable portability of compiled programs across desktop computers, servers, and smart devices, and (ii) they hide certain technical details, e.g., memory management, from software engineers. However, since both runtimes are relatively language-agnostic, they can also accommodate alternative programming languages. This has opened up the possibility for software modules that are written in different languages, but for the same runtime, to interoperate with each other. Today, Java's virtual machine hosts Scala, Clojure, and Kotlin, and C#'s virtual machine hosts F#, among other languages.

Scala was originally developed as an academic research and teaching language, aiming to address the shortcomings of Java and C# (Odersky et al., 2006). Within the decade after its first public release in 2004, it saw adoption among technology and finance firms (Odersky and Rompf, 2014).[8] Clojure, taking its inspiration from a well-known niche language called Common Lisp, was released in 2007 as an alternative to C++, Java, and C# in enterprise software. The language was subsequently adopted mainly for web services, data analysis, and financial technology applications (Hickey, 2020). Kotlin has been promoted by Google for building Android applications, and its use is largely limited there.

F# began as an attempt at Microsoft Research to adapt OCaml, a niche language mentioned earlier, to C#'s runtime. Its first major version was released in 2005. F# saw early interest among finance firms, and due to the inherent characteristics of the language and Microsoft's change in business strategy, cloud computing became an area of focus around 2012 (Syme, 2020).

In a 2019 Stack Overflow survey cited by Hickey (2020), engineers who used Clojure or F# were among the most experienced and highest earning respondents.

2.5. Screening devices. As proposed by Paul Graham (2004), high-effort programming languages may serve as valuable screening devices in the labor market. The trade-offs between high- and low-effort languages have been the subject of much discussion in the software industry. Proponents of hiring for high-effort languages say that high-ability engineers may be disproportionately more likely to apply. This view implicitly assumes a large gap between the marginal products of labor of high- and low-ability engineers. Opponents say that high-effort languages reduce the firm's applicant pool, making it harder to recruit engineers. This view implicitly assumes either that high-effort languages are ineffective as screening devices or that the MPL gap is small.

3. Theoretical Framework

This section proposes a signaling model of programming language choice. The analysis focuses on the production decision of teams rather than firms. Software firms in the industry usually consist of multiple teams that are responsible for different projects, each of which may use a different language. The model's key predictions will be tested in Sections 4 and 5.

[8] As an example, most of Twitter's code base has been written in Scala. Although the firm initially used Ruby, a popular language for web development, Ruby's slow execution, high memory use, and lack of static typing made it difficult for Twitter's web services to scale as its user base grew and its code base became more complex (Venners, 2009).

3.1. Engineer ability. Engineer i has latent ability $\theta_i \in \Theta := (0,1)$. She takes a job in a team that works with programming language $\ell \in \mathcal{L}$ and pays a wage $w(\ell)$. This yields her a utility $u(w(\ell),\theta_i) := w(\ell) - \gamma_\ell/\theta_i$. She chooses among her possible employers so as to maximize this utility.

The parameter $\gamma_\ell > 0$ reflects the objective difficulty of learning and using language ℓ. The ratio γ_ℓ/θ_i is the subjective difficulty from the perspective of engineer i. An engineer with higher latent ability finds the same language less difficult, i.e., $u(w(\ell),\theta_i) > u(w(\ell),\theta_j)$ for all $\theta_i > \theta_j$.

While I take a reduced-form approach to modeling latent ability, it is worth thinking about the real-world considerations that θ_i captures. From the team's perspective, the engineer's latent ability is salient because it affects team output via several mechanisms:

1. Defect rate: engineers with higher latent ability can write program code that is easier for other engineers to understand and that is more likely to be correct.
2. Long-term productivity: engineers with higher latent ability can structure program code in a way that is easier for other engineers to interface with and extend.
3. Task prioritization: engineers with higher latent ability can set priorities that better align with the team's strategic goal of delivering output and staying competitive.
4. Problem solving: engineers with higher latent ability can solve problems more quickly and they can find better solutions.

3.2. Team profit. Each team in the economy operates in a specific problem domain. The representative team in domain $\delta \in \Delta$ chooses a language ℓ. Its realized profit is

$$\Pi(\ell,\delta) := p(\delta)y(\ell,\delta) - N(\delta)w(\ell)$$

where $y(\ell,\delta)$ is the team's output, $p(\delta)$ is the market price of the team's output, $N(\delta)$ is the number of engineers hired by teams in domain δ, and $w(\ell)$ is the wage that it pays its engineers if it chooses language ℓ.

The team's output is defined in terms of the latent abilities of the engineers that it hires as

$$y(\ell,\delta) := \varphi_\ell(\delta)\left[N(\delta)\left((1-\alpha(\delta)) + \alpha(\delta)\prod_i \theta_i\right)\right]^{\beta(\delta)}.$$

In this formula, $\varphi_\ell(\delta) > 0$ is language ℓ's "fitness" in domain δ.[9] The parameter $\alpha(\delta) \in [0,1]$ determines the skill intensity of domain d, i.e., how reliant the team is on high-ability engineers, and $\beta(\delta) \in (0,1)$ is the labor share. The product term $\prod_i \theta_i$ reflects complementarities in engineer skill, as in Kremer's (1993) O-ring theory of the effect of worker skill on output quality and economic development.

[9]"Fitness" reflects the effect that a language has on team productivity via (i) its library ecosystem and (ii) its inherent characteristics. Examples of (i), such as R's CRAN, Python's PyPI, Stata's SSC, and TeX's CTAN, are well-known to economists. In the software industry, the recognition that code reuse improves productivity drove the 1990s popularization of both object-oriented programming and package ecosystems such as the Perl language's CPAN. Two examples of (ii) are automatic memory management which was introduced into mainstream software development by Java in 1996 (originally invented for the Lisp language in 1959), and strong static typing which has made it easier to manage large code bases, e.g., in web development when TypeScript was released in 2012.

The team observes the output price $p(\delta)$, the language fitness $\varphi_\ell(\delta)$, and the market wage $w(\ell)$, but it does not observe the ability of its engineers. Thus it maximizes *expected* profit:

$$\mathbb{E}(\Pi(\ell,\delta) \mid \ell) = p(\delta)\varphi_\ell(\delta)\mathbb{E}\left\{ \left[N(\delta)\left((1-\alpha(\delta)) + \alpha(\delta)\prod_i \theta_i \right) \right]^{\beta(\delta)} \,\middle|\, \ell \right\} - N(\delta)w(\ell).$$

The role of skill intensity, $\alpha(\delta)$, can be understood by considering the team's optimal choice at the extreme values that this parameter can take. At its minimum, $\alpha(\delta) = 0$, which implies that engineer skill is irrelevant to the team. In this case, the expectation operator can be dropped, and the team chooses the language that maximizes the difference $\varphi_\ell(\delta) - w(\ell)N(\delta)^{1-\beta(\delta)}/p(\delta)$ between language fitness and a normalized form of market wage. At its maximum, $\alpha(\delta) = 1$, the team is completely reliant on the ability of its engineers. Consequently, it might choose a language ℓ that has a lower value for $\varphi_\ell(\delta) - w(\ell)N(\delta)^{1-\beta(\delta)}/p(\delta)$ than other languages. Although such a choice may seem suboptimal to an outside observer who assumes that $\alpha(\delta) \ll 1$, it would in fact be optimal for the team because the language serves as a screening device to recruit high-ability engineers.

The team's optimal language choice fundamentally depends on the domain within which it operates. Languages can be ranked by their fitness parameter $\varphi_\ell(\delta)$. Depending on the team's domain, the fitness parameters, and thus the ranking of languages, can change. For example, due to its ecosystem and it being designed for statistical analysis, R is a much better fit for data analysis than C++. At the same time, C++ is a much better fit than R for implementing the estimation methods that are used in data analysis, as the methods require speed and memory efficiency.

3.3. Equilibrium. In equilibrium, each team chooses the language ℓ to use in production, and each engineer chooses the team for which to work based on teams' language choices and the market wage for each language. Each team takes all other aspects of production to be exogenous: the domain in which it operates, δ, the number of engineers it hires, $N(\delta)$, the fitness of each language, $\varphi_\ell(\delta)$, the skill-intensity parameter, $\alpha(\delta)$, the labor share, $\beta(\delta)$, the market wage, $w(\ell)$, and the price of its output good, $p(\delta)$. The labor market is perfectly competitive, and thus the market wage for each language is equal to the expected marginal revenue product of labor of the engineers that choose it in equilibrium.

To illustrate team and worker behavior in equilibrium, I make two additional simplifying assumptions. First, there are two ability levels, high and low, so that the ability space is $\Theta := \{\theta_H, \theta_L\}$ where $0 < \theta_L < \theta_H < 1$. Second, there are two languages that teams and engineers can choose from, a high- and a low-effort language, with the corresponding language space $\mathcal{L} := \{\ell_H, \ell_L\}$. There are two types of possible equilibria: separating equilibria in which engineers sort into languages by latent ability, and pooling equilibria in which all teams choose the same language and thus each team hires a mix of high- and low-ability engineers.

3.3.1. The engineer's choice. Recall that engineer i's utility is given by the function $u(w(\ell), \theta_i) := w(\ell) - \gamma_\ell/\theta_i$ where γ_ℓ/θ_i is the subjective difficulty of language ℓ. She decides whether to take a job that uses the high-effort language by evaluating the inequality $u(w(\ell_H), \theta_i) > u(w(\ell_L), \theta_i)$. In a separating equilibrium, where the high-effort language signals high ability, high-ability engineers prefer the high-effort language and low-ability engineers prefer the low-effort one:

$$\begin{aligned} u(w(\ell_L),\theta_H) &= w(\ell_L) - \frac{\gamma_{\ell_L}}{\theta_H} < w(\ell_H) - \frac{\gamma_{\ell_H}}{\theta_H} = u(w(\ell_H),\theta_H), \\ u(w(\ell_L),\theta_L) &= w(\ell_L) - \frac{\gamma_{\ell_L}}{\theta_L} \geq w(\ell_H) - \frac{\gamma_{\ell_H}}{\theta_L} = u(w(\ell_H),\theta_L).. \end{aligned}$$

These inequalities imply

$$\begin{aligned} \frac{\gamma_{\ell_H} - \gamma_{\ell_L}}{\theta_H} &< w(\ell_H) - w(\ell_L) \text{ and} \\ \frac{\gamma_{\ell_H} - \gamma_{\ell_L}}{\theta_L} &\geq w(\ell_H) - w(\ell_L), \end{aligned}$$

and if the high-effort language pays a higher wage than the low-effort language, then

$$0 < \theta_L \leq \frac{\gamma_{\ell_H} - \gamma_{\ell_L}}{w(\ell_H) - w(\ell_L)} < \theta_H.$$

3.3.2. The team's choice. The team chooses the high-effort language if and only if its expected profit is greater with the high-effort than with the low-effort language, $\mathbb{E}(\Pi(\ell_H,\delta) \mid \ell_H) > \mathbb{E}(\Pi(\ell_L,\delta) \mid \ell_L)$. In a separating equilibrium, all high-ability engineers choose the high-effort language and all low-ability engineers choose the low-effort language, so the expectation operator can be dropped.

3.4. Numerical illustration. The model may have multiple equilibria. Figure 1 depicts equilibria in an economy with two problem domains, domain a and domain b. The panels show parameter regions in which only a pooling equilibrium is possible (blue) and those in which both a pooling and a separating equilibrium exists (green). All parameters are kept constant throughout Figure 1, except for the fitness of the high-effort language (x axis), the skill intensity of domain a's production technology (y axis), and team size in domain a which increases from Panel (a) through Panel (d).

In the blue region, the high-effort language cannot be profitable either because its fitness is not high enough or because the skill intensity of production is not high enough. In the green region, by contrast, teams in domain a may choose the high-effort language, resulting in a separating equilibrium. In the shaded subsets of the green region in Panels (a) and (b), domain-a teams may choose the high-effort language purely for its signaling value: the low-effort language has a higher fitness (equal to 1) than the high-effort language (below 1). This situation only arises for small teams with highly skill-intensive production technologies.

Panel (c) shows that as team size increases, the high-effort language can only be chosen in equilibrium if its fitness is greater than that of the low-effort language. If domain-a teams have the same size as domain-b teams (Panel (d)), then the high-effort language is only chosen if it has an extremely high fitness and if skill intensity is very low.

Figure 2 shows team profits and wages for those points that are marked by a cross ("$\times$") in Figure 1. In a separating equilibrium, profits increase in team size for a team that chooses the high-effort language and thus hires high-ability engineers. However, the wage premium paid to engineers for using the high-effort language declines, driven by the decline of the marginal product of labor of high-ability engineers. This is because teams have an O-ring production technology. Team members' ability (θ_i) enters the production function multiplicatively, which results in a higher probability that any one team member makes a mistake. Mistakes render the output defective and thus reduce the team's productivity. As a consequence, for the parameter values used in Figure 2, no separating equilibrium exists above a team size of 5 (shown in gray).

3.5. Testable predictions. The model points towards two predictions that can be tested in data. First, *high-effort languages are chosen by smaller teams.* With smaller teams, the market wage premium for the high-effort language is large enough to compensate high-ability engineers for the disutility of using the language. In terms of the model parameters, a separating equilibrium is only possible for values of domain-a team size $N(a)$ that are small relative to $N(b)$.

Second, among the smallest teams, *high-effort languages are chosen in more skill-intensive problem domains.* For these teams, high-ability engineers' marginal product of labor is high enough to compensate them for having to use the high-effort language. In terms of the model parameters, with small teams in domain a, a separating equilibrium is only possible if the skill intensity $\alpha(a)$ of the production technology is high relative to the fitness $\varphi_H(a)$ of the high-effort language.

4. Languages as Signals of Engineer Skills

Within the theoretical framework of Section 3, engineer i is said to have ability θ_i which determines her productivity, and which firms thus care about. However, rather than directly observing θ_i, firms only observe i's experience with programming languages. In essence, they are interested in the difference

$$\Pr(\theta_i = \theta_H \mid x_{i\ell} = 1) - \Pr(\theta_i = \theta_H \mid x_{i\ell} = 0) \tag{1}$$

where $x_{i\ell} = 1$ if and only if i has worked with language ℓ.

For the purpose of screening workers, firms do not need to identify the causal effect of $x_{i\ell}$ on θ_i, merely its usefulness as a predictor. Thus the goal of this section is to assess how experience with high-effort languages predicts skills in other areas of software engineering.

4.1. Data. I construct my measures of programming-language skills using data from the question-and-answer platform *Stack Overflow.* This is the largest English-language platform on which contributors discuss specific technical questions and their answers are scored by the audience. While Stack Overflow is dedicated to programming languages and their associated ecosystems, there are dozens of spin-off sites that use the same scoring mechanism and operate under the umbrella *Stack Exchange.*

Both Stack Overflow and Stack Exchange publish "data dumps" that contain information on their users, the questions asked, and the answers proposed. For Stack Overflow, I use the December 2022 dump which covers August 2008 through early December 2022. The key variables in this data set are a question's score, date, and tags, an answer's score, date, and the unique identifier of its author.

For Stack Exchange, I use the April 2024 dump which covers late September 2009 through March 2024. The Stack Exchange sites that I draw data from are *Software Engineering, Code Review, Mathematics,* and *Cross Validated* (dedicated to statistical theory and machine learning). The first two sites cover skills that are useful in the software industry. The latter two sites cover skills that are not directly related to programming but that are nevertheless transferable across jobs and may prove useful in certain problem domains. The key variables in this data set are users' reputation, first answer date, and unique identifier that links them to the Stack Overflow dump.

A user's reputation is mainly a function of their question and answer scores.[10] New users start out with a reputation of 1. Users who have attained a reputation of at least 200 on either Stack Overflow or any other Stack Exchange site get a reputation boost of 100, so their starting reputation is 101 when they register on a new site.

4.2. Knowledge of low-effort mainstream languages. First, I show that the knowledge of high-effort languages predicts better knowledge of some low-effort languages than the knowledge of other low-effort languages does. This fact can be leveraged by a firm. For example, suppose that engineers who have experience with C++, a high-effort language, also tend to be more highly skilled in Java, a low-effort mainstream language that is widely taught at universities. Then a firm that uses Java for most of its code base could consider choosing C++ for some of its projects because the engineers hired into C++ teams can bring their expertise to the Java code base, too.

To set up the specification, let i index individuals who answered questions about a low-effort language, and let j index such questions. On Stack Overflow, both questions and answers are voted on by users and assigned (possibly negative) scores based on those votes. As empirical counterparts to (1), one can think of

$$\begin{aligned}&\mathbb{E}\left(\text{JavaAnswerScore}_{ij} \mid \text{KnowsCpp}_i = 1, \text{KnowsPython}_i = 0\right) - \\ &\quad - \mathbb{E}\left(\text{JavaAnswerScore}_{ij} \mid \text{KnowsCpp}_i = 0, \text{KnowsPython}_i = 1\right)\end{aligned} \tag{2}$$

which captures how knowledge of C++, *as opposed to* knowledge of Python, predicts i's ability at Java, and

$$\begin{aligned}&\mathbb{E}\left(\text{JavaAnswerScore}_{ij} \mid \text{KnowsCpp}_i = 1, \text{KnowsPython}_i = 1\right) - \\ &\quad - \mathbb{E}\left(\text{JavaAnswerScore}_{ij} \mid \text{KnowsCpp}_i = 0, \text{KnowsPython}_i = 1\right)\end{aligned} \tag{3}$$

which captures how knowledge of C++, *in addition to* knowledge of Python, predicts i's ability at Java. I find that (2) is positive in the data, and so are many similar comparisons between high- and low-effort languages. By contrast, while (3) is positive, analogous comparisons are positive for virtually any programming language, whether high- or low-effort.

To operationalize comparisons like (2) and (3), I denote the score assigned to question j by QuestionScore_j and the score assigned to individual i's answer by AnswerScore_{ij}. The main specification that I fit to the data is

$$\begin{aligned}\text{AnswerScore}_{ij} = \beta_0 &+ \beta_H \text{KnowsHighEffort}_i + \beta_L \text{KnowsLowEffort}_i + \\ &+ \beta_{HL} \text{KnowsHighEffort}_i \text{KnowsLowEffort}_i + \mu_{y(j)} + \sigma_{\text{QuestionScore}_j} + \varepsilon_{ij}\end{aligned} \tag{4}$$

where AnswerScore_{ij} is the score of an answer about a low-effort language (Java, C#, JavaScript, or Python), KnowsHighEffort_i is a binary indicator for whether individual i has also posted at least one answer about a

[10]A user's reputation also changes if they receive votes on comments that they have made (displayed underneath other users' questions or answers), if edits that they suggest to others' questions or answers are approved, and if they vote someone else down. However, the amounts by which these events change reputation are smaller.

high-effort language (e.g., C++, Scala, F#), KnowsLowEffort_i is a binary indicator for whether i has also posted at least one answer about a low-effort language that is different from the language that question j is about, $\mu_{y(j)}$ is a fixed effect for the year in which question j was asked, $\sigma_{\text{QuestionScore}_j}$ is a fixed effect for the question's score, and ε_{ij} is an exogenous residual term. The fixed effects account for two factors that affect AnswerScore_{ij}, independently of the quality of the answer itself: the age and the popularity of the question that is being answered. Questions that are older or more popular receive more views. Answers to questions with more views can naturally accumulate more votes.

I test two null hypotheses in equation (4). As an example, let's take C++ as the high-effort language, Python as the low-effort language, and Java as the language that the answer is about. Then the two null hypotheses and their interpretation in the context of the theoretical framework are:

H_A : $\beta_H = \beta_L$. Knowing C++ but not Python predicts the same Java ability as knowing Python but not C++. This means that C++ and Python have the same signaling value to firms.

H_B : $\beta_H + \beta_{HL} = 0$. Knowing *both* C++ and Python predicts the same Java ability as knowing only Python. This means that C++ has no signaling value beyond that which Python has.

Table 1 shows estimates of equation (4). C++ is a better signal of both Java and C# ability than Python is. We see this for Java in column (1) by comparing the base coefficients for C++ (1.117) and Python (0.737). The difference of the two base coefficients, $1.117 - 0.737 = 0.380$, is positive, indicating that Stack Overflow users who have answered questions about C++ (but not Python) achieve higher answer scores on Java than users who have answered questions about Python (but not C++). Similarly, for C#, column (2) shows a base-coefficient difference of $1.208 - 0.686 = 0.522$. The t-statistics for the null hypothesis H_A that show that these differences are statistically significantly non-zero at the 5-percent level are in Table 2.

Scala and F# are two other high-effort languages. However, they are unlike C++ which compiles to native machine code. Instead, Scala and F# both primarily compile to a virtual machine, with Scala being bound to the Java platform and F# to the C# platform. Table 1 mirrors the dichotomy between the two languages. Scala is a much better signal of Java ability than Python is, with a difference of $1.606 - 0.886 = 0.720$ ($t \approx 5.27$ in Table 2). However, its signal value is no different for C#, with a difference of only $1.382 - 1.369 = 0.013$ ($t \approx 0.05$). The opposite is true for F# which is a much better than signal of C# ability ($t \approx 3.59$) but not statistically different from Python as a signal of Java ability ($t \approx 1.20$).

Overall, Stack Overflow answer scores suggest that C++ is the most versatile among high-effort languages as a signal of latent ability. C++ has more comparisons in which it is a better signal of ability than other high-effort languages. It is statistically better than three other languages for Java, four other languages for C#, two other languages for JavaScript, and four other languages for Python (Table 2). By contrast, Scala is a good signal for Java ability but not so much for ability in the other three languages, and F# is a good signal for C# ability but a weaker signal for other languages.

Unlike hypothesis H_A, H_B holds for nearly any language pair. E.g., the knowledge of C++ and Python signals higher C# ability than the knowledge of Python alone. This is shown by the sum of the base coefficient and the interaction term, $1.208 + 0.528$, in column (2) of Table 1, and by the corresponding t-statistic, 9.72, in Table 3. The only languages in this study that are occasional exceptions are Go and R. Go, a language designed for extreme simplicity at Google primarily for writing network services, has little extra signaling value for Python ability (Panel D of Table 3). R, a language mainly used for statistical analysis, has little extra signaling value for Java or C# (Panels A and B).

4.3. Interest in transferable technical skills. In addition to skills in specific programming languages, firms may also value an engineer's broader interest in software engineering and other technical topics. I show that knowledge of high-effort languages predicts such interests using the Stack Exchange data.

The specification that I fit is

$$\begin{aligned}\mathbf{1}_{\{\text{Reputation}_i \geq 201\}} = \beta_0 + \beta_H \text{KnowsHighEffort}_i + \beta_L \text{KnowsLowEffort}_i + \\ + \beta_{HL} \text{KnowsHighEffort}_i \text{KnowsLowEffort}_i + \mu_{y_0(i)} + \varepsilon_i\end{aligned} \tag{5}$$

where $\mu_{y_0(i)}$ is a fixed effect for the first year in which individual i posted an answer on Stack Overflow and ε_i is an exogenous residual term. The outcome variable is binary and defined in terms of Reputation$_i$ which is i's reputation on a specific Stack Exchange site. I test the same null hypotheses, H_A and H_B, for equation (5) that I tested for (4) in the previous subsection.

Table 4 shows estimates of equation (5). C++ is a better signal of interest in software engineering, code review, and computer science, than Python is. This is seen by comparing the base coefficients, e.g., $0.010 - 0.004 = 0.006$ for software engineering in column (1). The t-statistics for the null hypothesis H_A in Table 5 confirm that these base coefficients are different at the 5-percent level. Likewise, F# is a better signal than Python (columns (7)–(9)). However, results are mixed for Scala. Column (4) shows that it is a better signal for software engineering with a base-coefficient difference of $0.014 - 0.008 = 0.006$ ($t \approx 5.96$). But columns (5) and (6) shows little difference for code review and computer science which is confirmed by the t-statistics in Table 5.

Table 5 shows that high-effort languages are generally better signals of interest in transferable technical skills than low-effort languages. Unsurprisingly, the one clear exception is statistics and machine learning (Panel E), for which Python and R and much better signals than any high-effort language.

For hypothesis H_B, similarly to the results in the previous subsection, the results show that additional knowledge of nearly any programming language signals interest in transferable skills. This is shown in Table 6.

Overall, the results in the Stack Exchange data suggest that high-effort languages are strong signals of transferable skills that go beyond an engineer's expertise in specific programming languages. C++ and Haskell are particularly versatile in this capacity, but Scala, Clojure, and F# also do well. Among low-effort mainstream languages, Python is perhaps the best signal of an engineer's interest in mathematics and statistical theory. For firms that operate in highly technical problem domains, e.g., securities trading, artificial intelligence, or biomedical engineering, these signals may be valuable in the screening process.

5. The Language Choice of Firms

The theoretical framework of Section 3 predicts that the teams that adopt high-effort languages are smaller, and that high-effort languages see more adoption in skill-intensive problem domains than they do in non-skill-intensive ones. The goal of this section is to assess whether these predictions hold in U.S. job postings data.

5.1. Data. I use *Lightcast's* near-universe of U.S. job postings between January 2010 and June 2024. The key variables for my analysis are a unique firm identifier, metropolitan area, posting date, occupation code, software skills listed, and problem domains mentioned in the text of the posting. The search terms used to construct the problem-domain indicators are listed in Appendix A. I filter the job postings for software engineering, hardware engineering, and data science occupations. The occupation codes used are shown in Table 13.

5.2. Job posting gaps between languages within the firm. The model in Section 3 predicts that high-effort languages are adopted by smaller teams.[11] I don't observe team size directly in the data. However, I can compare the number of job postings for a high-effort language with the number of job postings for a low-effort language within the *same* firm. I find that for nearly every such comparison, the high-effort language has fewer job postings than the low-effort language. These comparisons are a valid test of the prediction as long as, on average, teams that use the high-effort language (a) have the same or higher turnover, and (b) post just as many or more job ads for the same number of vacancies, as teams that use the low-effort language.

Let $\text{Postings}_{cm\ell y}$ be the number of job postings made by firm c in metropolitan area m mentioning language ℓ in year y. My estimand is the average gap between, e.g., $\text{Postings}_{cm,\text{Cpp},y}$ and $\text{Postings}_{cm,\text{Java},y}$, for those firms that operate in a given problem domain and that hire for both C++ and Java in at least one metropolitan area and one year in the data. Formally, this can be written as

$$\mathbb{E}\left(\text{Postings}_{cm\ell y} - \text{Postings}_{cm\ell' y} \;\middle|\; \text{Domain}(c) = \delta, \sum_{m',y'} \text{Postings}_{cm'\ell y'} > 0, \sum_{m',y'} \text{Postings}_{cm'\ell' y'} > 0\right) \tag{6}$$

for a fixed language pair (ℓ, ℓ') and a fixed domain δ. I estimate (6) by fitting

$$\text{Postings}_{cm\ell y} - \text{Postings}_{cm\ell' y} = \beta_0 + \sum_{\delta \in \Delta} \beta_\delta \mathbf{1}_{\{\text{Domain}(c)=\delta\}} + \varepsilon_{cm\ell\ell' y} \tag{7}$$

to the data for each (ℓ, ℓ'). The problem domains included in Δ are securities trading, biomedical engineering, medicine, manufacturing, embedded systems, financial technology, and web applications.[12] I cluster the standard errors by company c and metropolitan area m.

Table 7 shows estimates of equation (7). These estimates can be used in a straightforward way to test whether firms tend to post fewer jobs for high- than for low-effort languages. For example, if securities trading firms that use both C++ and Python tend to post fewer C++ than Python jobs, then $\beta_0 + \beta_{\text{Trading}} < 0$. In column (1), this inequality indeed holds, with a point estimate of $-0.507 - 0.287 < 0$. The inequality also holds for Scala (column (2)) and F# (column (3)) compared with Python, as well as for nearly every other problem domain shown in Table 7. The domains for which it does not hold are embedded systems ($t \approx 3.68$)

[11] For high-effort niche languages, this prediction is supported by anecdotal evidence. Firms that hire for such a language often not only post a job ad but also reach out to the language community directly. Such outreach shows that at larger firms, high-effort languages are often adopted by individual teams that work on specific projects. E.g., Facebook has had a Haskell team that uses the language for code search and abuse detection, and Target had a Haskell team that worked on data science and optimization, but no use of the language was reported elsewhere in either organization.

[12] Web applications are commonly referred to as "software as a service" (SaaS), i.e., the delivery of software as a web service rather than as a program that the user needs to install.

and manufacturing ($t \approx -0.40$), where C++ has a much greater fitness than Python for many or most tasks.

Table 8 shows t-statistics for the null hypothesis that $\beta_0 + \beta_\delta = 0$, across various domains δ and for various language pairs. These results further support the finding that the same firm posts fewer jobs for high-effort languages than for low-effort ones. The cases where the null cannot be rejected at the 5-percent level, or where the null is rejected but the t-statistic is positive, are those in which the low-effort language has a domain-specific disadvantage. An example is Go in biomedical engineering (Panel C) where developing network services, the language's strongest application area, is not a key focus.

5.3. Relative domain concentration. The second prediction of the model of Section 3 is that small teams are more likely to adopt high-effort languages in skill-intensive problem domains. To test this prediction, I look at whether the use of high-effort languages is more concentrated in more technical problem domains than the use of low-effort languages. For example, C++ has fewer job postings in both securities trading and web applications than Java. However, the ratio of C++ postings in securities trading to C++ postings in web applications is higher than the corresponding ratio for Java. This indicates that the use of C++ is more concentrated in securities trading, a more technical domain, than the use of Java.

Let $\text{Postings}_{cm\ell y}$ be the number of job postings made by firm c in metropolitan area m mentioning language ℓ in year y. I use ordinary least squares to fit the specification

$$\ln \text{Postings}_{cm\ell y} = \alpha_\ell + \sum_{\delta \in \Delta} \beta_{\ell\delta} \mathbf{1}_{\{\text{Domain}(c)=\delta\}} + \lambda_c + \mu_m + \theta_y + \varepsilon_{cm\ell y} \tag{8}$$

where Δ contains the same domains as for (7). The regression sample for equation (8) necessarily excludes observations in years in which the company made zero postings for a given language. Thus as a robustness check, I also use negative binomial regressions to fit

$$\mathbb{E}\left(\text{Postings}_{cm\ell y}\right) = \exp\left(\alpha_\ell + \sum_{\delta \in \Delta} \beta_{\ell\delta} \mathbf{1}_{\{\text{Domain}(c)=\delta\}} + \lambda_c + \mu_m + \theta_y\right). \tag{9}$$

The null hypothesis that I test is constructed to compare the two ratios in the motivating example above. First, I look at the difference between a language ℓ's adoption in domain δ and its adoption in web applications: $\beta_{\ell\delta} - \beta_{\ell,\text{WebApp}}$. Then I compare this difference to the analogous difference for another language ℓ', by subtracting one from the other: $(\beta_{\ell\delta} - \beta_{\ell,\text{WebApp}}) - (\beta_{\ell'\delta} - \beta_{\ell',\text{WebApp}})$. Testing whether this double difference equals zero is equivalent to testing whether the ratios are equal, due to the logarithmic specification of equations (8) and (9). I call the double difference the *relative concentration* of language ℓ, compared with another language ℓ', in domain δ.

The definition of relative domain concentration uses web applications as the reference domain because there is reason to believe that this is the least technical among the problem domains that I consider. Many key skills required for web services have been taught in "coding bootcamps" that take an average of 14 to 17 weeks to complete. Coding bootcamps became popular during the 2010s, and there were over a hundred different bootcamps in 2019 across the U.S. found by a market report (Eggleston, 2021). Almost all bootcamp participants study web development. Most participants have no formal computer science or engineering education, and no work experience as a software developer.

Table 9 shows estimates of equation (8). The use of C++ is more concentrated in highly technical domains than the use of Java. For example, for securities trading, relative domain concentration is $0.024 + 0.147 + 0.033 - 0.139 = 0.065$ in column (1) which omits fixed effects, and 0.112 in column (3) which includes firm, year, and MSA fixed effects ($t \approx 4.40$ in Panel A of Table 10). Similar results hold for C++ in artificial intelligence, biomedical engineering, medicine, and manufacturing.

Table 10 shows domain-by-domain t-statistics for the null hypothesis that the relative domain concentration of a given high-effort language compared with a given low-effort language is zero. C++ has a statistically significantly positive relative concentration in almost every comparison. Results for Scala are more mixed, with positive relative concentration in securities trading and biomedical engineering (Panels A and C) but no statistical evidence of non-zero relative concentration in many comparisons in the other problem domains. Haskell, a language that was originally created as a testing bed for programming language researchers, only sees positive relative concentration in biomedical engineering. The two remaining high-effort languages that I study, Clojure and F#, tend to see no positive relative concentration. This overall pattern is robust to changing the estimator from OLS to a negative binomial regression and estimating equation (9) instead (Table 11).

6. Conclusion

Software firms choose key aspects of their production function endogenously. I focus on one major aspect, the programming language that they write their software in. A signaling model illustrates that endogenous language choice could, at least partly, be driven by the firm's effort to select high-ability applicants in the labor market.

The premise of the model, i.e., that engineers who know high-effort programming languages have higher latent ability, is supported in detailed data on skills from the largest online question-and-answer platform, *Stack Overflow.* I show that the most versatile predictor of programming skills is C++, one of the oldest and most complex mainstream languages. Newer high-effort languages are also good predictors in some cases but less reliably so. The history of C++ may explain this difference, in particular the language's backwards compatibility guarantee that forces engineers to remain aware of lower-level technical details and of a detailed list of rules and exceptions that are laid out in popular guidelines.

Two testable predictions derived from the model are supported in data on U.S. job postings from *Lightcast.* First, among firms that hire for both a high- and a low-effort language, the high-effort language appears in fewer job postings than the low-effort language within the same firm. This suggests that the high-effort language is used by fewer engineers within the firm. Second, C++, and to a lesser extent Scala, are more concentrated in technical problem domains than low-effort languages. This suggests that high-effort languages are adopted by firms that have more skill-intensive production functions, that is, where latent ability is more important.

Since at larger firms, not every team uses the same language, high-effort languages usually coexist with low-effort ones. Thus high-ability engineers hired for one team can advise lower-ability engineers on another team. Detailed data on individual work output, such as engineers' program code quality and their participation in "code review," could shed further light on the extent to which engineers who use high-effort languages contribute to improving productivity at the firm.

References

Abebe, G., A. S. Caria, M. Fafchamps, P. Falco, S. Franklin, and S. Quinn (2021a): "Anonymity or Distance? Job Search and Labour Market Exclusion in a Growing African City," *The Review of Economic Studies*, 88, 1279–1310.

Abebe, G., A. S. Caria, and E. Ortiz-Ospina (2021b): "The Selection of Talent: Experimental and Structural Evidence from Ethiopia," *American Economic Review*, 111, 1757–1806.

Abel, M., R. Burger, and P. Piraino (2020): "The Value of Reference Letters: Experimental Evidence from South Africa," *American Economic Journal: Applied Economics*, 12, 40–71.

Aigner, D. J. and G. G. Cain (1977): "Statistical Theories of Discrimination in Labor Markets," *ILR Review*, 30, 175–187.

Azoulay, P., J. S. Graff Zivin, and J. Wang (2010): "Superstar Extinction," *The Quarterly Journal of Economics*, 125, 549–589.

Bandiera, O., I. Barankay, and I. Rasul (2007): "Incentives for Managers and Inequality among Workers: Evidence from a Firm-Level Experiment," *The Quarterly Journal of Economics*, 122, 729–773.

Carranza, E., R. Garlick, K. Orkin, and N. Rankin (2022): "Job Search and Hiring with Limited Information about Workseekers' Skills," *American Economic Review*, 112, 3547–83.

Chen, Y. (2021): "Team-Specific Human Capital and Team Performance: Evidence from Doctors," *American Economic Review*, 111, 3923–62.

Delfgaauw, J. and R. Dur (2007): "Signaling and screening of workers' motivation," *Journal of Economic Behavior & Organization*, 62, 605–624.

Eggleston, L. (2021): "2019 Coding Bootcamp Market Size Study," report, https://web.archive.org/web/20240715025508/https://www.coursereport.com/reports/coding-bootcamp-market-size-research-2019.

Emanuel, N., E. Harrington, and A. Pallais (2023): "The Power of Proximity: Training for Tomorrow or Productivity Today?" working paper.

Fenizia, A. (2022): "Managers and Productivity in the Public Sector," *Econometrica*, 90, 1063–1084.

Gibbs, M., F. Mengel, and C. Siemroth (2023): "Work from Home and Productivity: Evidence from Personnel and Analytics Data on Information Technology Professionals," *Journal of Political Economy Microeconomics*, 1, 7–41.

Graham, P. (2004): "The Python Paradox," https://web.archive.org/web/20240515221423/https://paulgraham.com/pypar.html.

Guo, P. (2014): "Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities," https://web.archive.org/web/20240616180637/https://cacm.acm.org/blogcacm/python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/.

Hamilton, B. H., J. A. Nickerson, and H. Owan (2003): “Team Incentives and Worker Heterogeneity: An Empirical Analysis of the Impact of Teams on Productivity and Particip ation,” *Journal of Political Economy*, 111, 465–497.

Hickey, R. (2020): “A history of Clojure,” *Proc. ACM Program. Lang.*, 4.

Hoare, C. A. R. (1974): “Hints on Programming Language Design,” in *Computer Systems Reliability*, ed. by C. Bunyan, vol. 20, 505–534.

Horowitz, E. (1983): *Fundamentals of Programming Languages*, Rockville, Md.: Computer Science Press.

Hsieh, C.-T., E. Hurst, C. I. Jones, and P. J. Klenow (2019): “The Allocation of Talent and U.S. Economic Growth,” *Econometrica*, 87, 1439–1474.

Huang, F. and P. Cappelli (2010): “Applicant Screening and Performance-Related Outcomes,” *American Economic Review: Papers & Proceedings*, 100, 214–18.

Iverson, K. E. (1980): “Notation as a Tool of Thought,” *Commun. ACM*, 23, 444–465.

Jackson, C. K. and E. Bruegmann (2009): “Teaching Students and Teaching Each Other: The Importance of Peer Learning for Teachers,” *American Economic Journal: Applied Economics*, 1, 85–108.

Jaffe, A. B., M. Trajtenberg, and M. S. Fogarty (2000): “Knowledge Spillovers and Patent Citations: Evidence from a Survey of Inventors,” *American Economic Review*, 90, 215–218.

Jaffe, A. B., M. Trajtenberg, and R. Henderson (1993): “Geographic Localization of Knowledge Spillovers as Evidenced by Patent Citations,” *The Quarterly Journal of Economics*, 108, 577–598.

Jaravel, X., N. Petkova, and A. Bell (2018): “Team-Specific Capital and Innovation,” *American Economic Review*, 108, 1034–73.

Kremer, M. (1993): “The O-Ring Theory of Economic Development,” *The Quarterly Journal of Economics*, 108, 551–575.

Landin, P. J. (1966): “The Next 700 Programming Languages,” *Commun. ACM*, 9, 157–166.

Madhavapeddy, A. and Y. Minsky (2022): *Real World OCaml: Functional Programming for the Masses*, Cambridge University Press, 2 ed.

Mas, A. and E. Moretti (2009): “Peers at Work,” *American Economic Review*, 99, 112–45.

Murphy, K. M., A. Shleifer, and R. W. Vishny (1991): “The Allocation of Talent: Implications for Growth,” *The Quarterly Journal of Economics*, 106, 503–530.

Odersky, M., P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger (2006): “An Overview of the Scala Programming Language,” technical report.

Odersky, M. and T. Rompf (2014): “Unifying functional and object-oriented programming with Scala,” *Commun. ACM*, 57, 76–86.

Pallais, A. (2014): "Inefficient Hiring in Entry-Level Labor Markets," *American Economic Review*, 104, 3565–99.

Papay, J. P., E. S. Taylor, J. H. Tyler, and M. E. Laski (2020): "Learning Job Skills from Colleagues at Work: Evidence from a Field Experiment Using Teacher Performance Data," *American Economic Journal: Economic Policy*, 12, 359–88.

Pike, R. (2012): "Go at Google: Language Design in the Service of Software Engineering," https://talks.golang.org/2012/splash.article.

Ritchie, D. M. (1996): *The development of the C programming language*, New York, NY, USA: Association for Computing Machinery, 671–698.

Sammet, J. E. (1972): "Programming Languages: History and Future," *Commun. ACM*, 15, 601–610.

Spence, M. (1973): "Job Market Signaling," *The Quarterly Journal of Economics*, 87, 355–374.

——— (2002): "Signaling in Retrospect and the Informational Structure of Markets," *American Economic Review*, 92, 434–459.

Stroustrup, B. (1996): *A history of C++: 1979–1991*, New York, NY, USA: Association for Computing Machinery, 699–769.

——— (2007): "Evolving a language in and for the real world: C++ 1991–2006," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, New York, NY, USA: Association for Computing Machinery, HOPL III, 4–1–4–59.

——— (2020): "Thriving in a crowded and changing world: C++ 2006–2020," *Proc. ACM Program. Lang.*, 4.

Sutter, H. (2024): "Safety, Security, Safety (sic), and C/C++ (sic)," ACCU Conference, keynote.

Syme, D. (2020): "The early history of F#," *Proc. ACM Program. Lang.*, 4.

The White House (2024): "Back to the Building Blocks: A Path Toward Secure and Measurable Software," report.

Venners, B. (2009): "Twitter on Scala: A Conversation with Steve Jenson, Alex Payne, and Robey Pointer," https://web.archive.org/web/20240325081304/https://www.artima.com/articles/twitter-on-scala.

Wheeler, L., R. Garlick, E. Johnson, P. Shaw, and M. Gargano (2022): "LinkedIn(to) Job Opportunities: Experimental Evidence from Job Readiness Training," *American Economic Journal: Applied Economics*, 14, 101–25.

Winters, T., T. Manshreck, and H. Wright (2020): *Software Engineering at Google: Lessons Learned from Programming over Time*, O'Reilly Media.

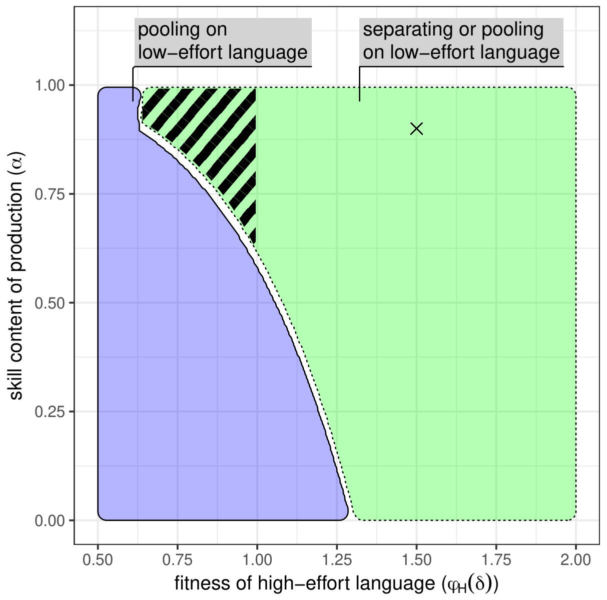
A. Search terms used in job descriptions for problem domains

The problem-domain indicators are constructed using the following search terms:

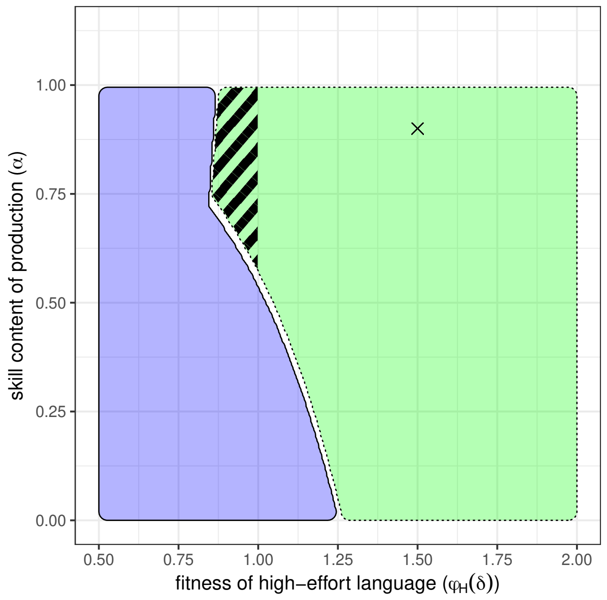
- Securities Trading: *"financial market," "commodity market," "financial option," "financial derivative," "trading."*
- Artificial Intelligence: *"artificial intelligence," "AI."*
- Biomedical Engineering: *"bio med…," "bio engin…"*
- Medicine: *"hospitals," "pharma…"*
- Manufacturing: *"manufacturing."*
- Embedded Systems: *"embedded."*
- Financial Technology: *"financial tech…," "fintech," "payments."*
- Web Applications: *"saas," "software as a service."*

If a search term consists of multiple words, I also include its hyphenated variant to account for alternative spellings.

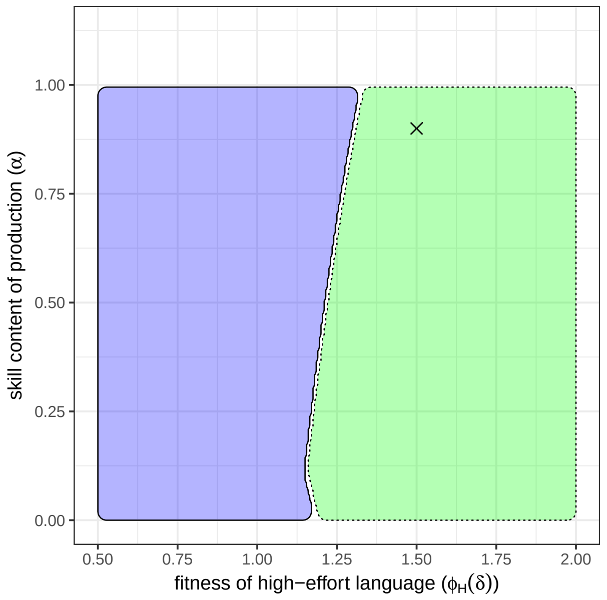
Figure 1. Parameter regions with pooling and separating equilibria



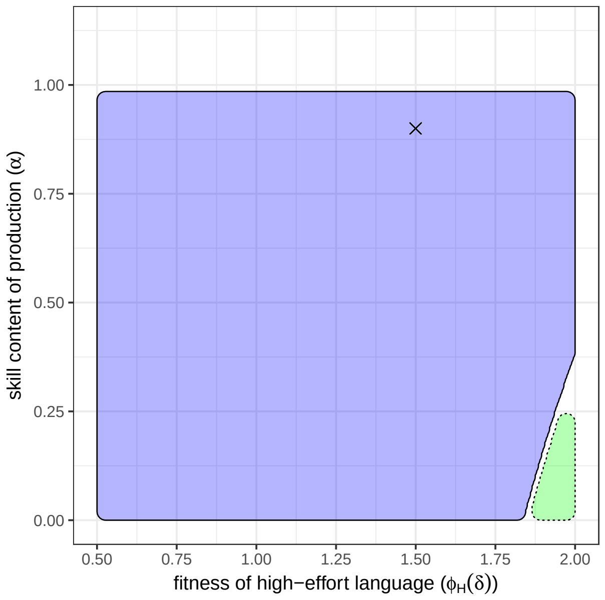
(a) $N = 2$



(b) $N = 3$



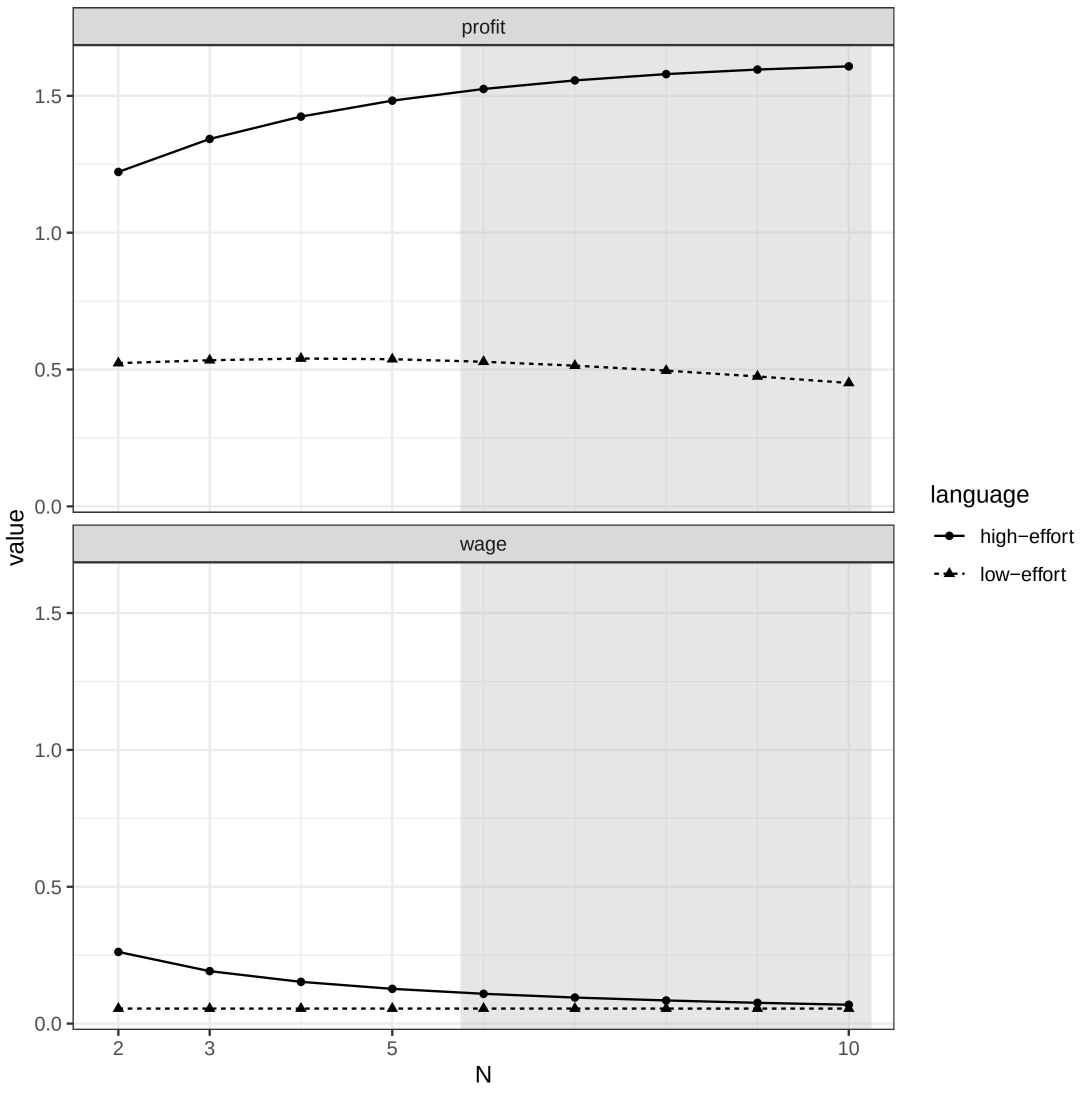
(c) $N = 5$



(d) $N = 10$

Notes: This figure shows that a separating equilibrium in which the high-effort language is used only exists if teams are smaller. Panel (a) shows equilibria for a team size of $N = 2$, and panels (b)–(d) show equilibria for larger team sizes. In the blue region, only a pooling equilibrium exists, with all teams choosing the low-effort language. In the green region, a separating equilibrium is also possible. In the shaded region, a separating equilibrium is possible even though the high-effort language has lower fitness than the low-effort language. Profits and wages that correspond to the points marked by "×" are shown in Figure 2. The complete list of model parameters is shown in Table 12.

Figure 2. Profits and wages in candidate separating equilibria



Notes: This figure shows that in a candidate separating equilibrium, profit increases but the wage premium for high-ability engineers decreases with team size. In the region shown in gray, no separating equilibrium exists because the wage premium is too low and high-ability engineers prefer the low-effort language. The profits and wages for $N = 2$, 3, 5, and 10 correspond to the points marked by "×" in Figure 1, i.e., they are obtained under $\varphi_H(a) = 1.5$ and $\alpha = 0.9$. The complete list of model parameters is shown in Table 12.

TABLE 1. Knowledge of high-effort languages and Stack Overflow answer scores on low-effort languages

	Java (1)	C# (2)	Java (3)	C# (4)	Java (5)	C# (6)
C++	1.117***	1.208***				
	(0.1124)	(0.0840)				
C++ × Python	0.378*	0.528*				
	(0.1949)	(0.2714)				
Scala			1.606***	1.382***		
			(0.1279)	(0.2614)		
Scala × Python			0.043	0.365		
			(0.2359)	(0.8829)		
F#					1.908***	1.748***
					(0.4457)	(0.2001)
F# × Python					0.139	0.615
					(0.9871)	(0.5132)
Python	0.737***	0.686***	0.886***	1.369***	1.366***	1.017***
	(0.0603)	(0.0822)	(0.0687)	(0.1114)	(0.0865)	(0.0650)
Observations	2,927,074	2,463,895	2,927,074	2,463,895	2,927,074	2,463,895
Question Score FE	YES	YES	YES	YES	YES	YES
Question Year FE	YES	YES	YES	YES	YES	YES

Notes: This table shows estimates for equation (4). The outcome variable is an answer score on a question about the language that is indicated in the column. Tests of the equality of the base coefficients, e.g., $\beta_{\text{Cpp}} - \beta_{\text{Python}} = 0$, are shown in Table 2. Tests of whether the sum of a base coefficient and its corresponding interaction term is zero, e.g., $\beta_{\text{Cpp}} + \beta_{\text{Cpp}\times\text{Python}} = 0$, are shown in Table 3. Standard errors are clustered by individual. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 2. Tests for whether high-effort languages predict higher answer scores than low-effort languages

	Other language						
High-effort language	C	Java	C#	Go	JavaScript	Python	R
Panel A: Outcome variable is answer score for Java							
Knows...							
C++	−1.71*		−3.81***	7.52***	1.69*	3.10***	6.97***
Scala	1.13		0.79	3.51***	3.41***	5.27***	5.89***
Clojure	1.78*		0.86	2.23**	1.57	3.70***	5.99***
F#	1.23		1.51	1.92*	1.61	1.20	2.96***
Haskell	1.03		0.97	1.71*	0.18	2.04**	4.38***
Panel B: Outcome variable is answer score for C#							
Knows...							
C++	1.96*	−0.29		5.17***	4.35***	4.71***	10.34***
Scala	−1.29	0.28		−0.68	0.13	0.05	0.73
Clojure	1.33	−1.38		1.00	−0.78	0.34	2.18**
F#	2.62***	1.73*		4.02***	3.24***	3.59***	5.55***
Haskell	1.11	0.46		2.13**	2.02**	0.10	4.01***
Panel C: Outcome variable is answer score for JavaScript							
Knows...							
C++	0.43	−2.24**	−1.51	5.60***		1.56	7.92***
Scala	−0.82	0.24	−0.95	1.56		0.58	2.84***
Clojure	0.96	1.03	0.98	3.02***		0.57	5.14***
F#	0.90	1.20	1.35	2.52**		0.91	3.34***
Haskell	0.26	0.75	0.35	3.93***		1.09	5.53***
Panel D: Outcome variable is answer score for Python							
Knows...							
C++	−1.91*	2.07**	0.76	6.38***	3.42***		5.00***
Scala	−3.66***	−5.20***	−4.14***	2.40**	−2.70***		0.37
Clojure	−0.71	−0.26	0.29	4.28***	−0.16		3.05***
F#	−0.74	0.74	0.94	4.15***	1.13		3.69***
Haskell	1.22	2.15**	2.52**	6.07***	2.58**		6.25***

Notes: Each number is a t-statistic for the null hypothesis that $\beta_H - \beta_L = 0$ in equation (4), with KnowsHighEffort_i and KnowsLowEffort_i defined using the languages indicated in the row and the column, respectively. Standard errors are clustered by individual. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 3. Tests for whether an additional programming language predicts higher answer scores

	Other language						
Additional language	C	Java	C#	Go	JavaScript	Python	R
	Panel A: Outcome variable is answer score for Java						
Knows...							
C++	5.43***		9.59***	5.20***	10.90***	9.72***	4.56***
C			8.37***	5.09***	10.30***	8.81***	4.11***
C#	8.88***			4.85***	12.50***	12.05***	5.49***
Go	2.38**		2.72***		3.71***	3.06***	1.94*
JavaScript	4.30***		5.04***	4.92***		10.06***	4.81***
Python	4.25***		6.98***	4.00***	10.04***		3.94***
R	1.00		1.63	0.87	2.76***	1.89*	
	Panel B: Outcome variable is answer score for C#						
Knows...							
C++	5.50***	7.46***		3.56***	8.16***	6.56***	3.43***
C		6.20***		3.77***	7.01***	5.79***	2.80***
Java	6.87***			3.58***	8.95***	8.41***	3.71***
Go	2.58***	2.57**			2.88***	2.44**	1.41
JavaScript	5.50***	6.02***		3.03***		5.68***	1.87*
Python	4.48***	5.57***		2.97***	6.35***		1.25
R	1.59	1.87*		0.93	2.13**	1.50	
	Panel C: Outcome variable is answer score for JavaScript						
Knows...							
C++	5.50***	11.41***	10.39***	6.63***		10.98***	5.97***
C		10.54***	9.69***	6.20***		9.92***	4.53***
Java	9.09***		13.95***	7.29***		12.89***	7.17***
C#	8.48***	15.43***		7.03***		13.52***	4.68***
Go	1.90*	5.25***	4.85***			4.48***	2.05**
Python	5.91***	10.74***	10.47***	4.79***			4.53***
R	−0.08	2.69***	2.13**	0.39		2.02**	
	Panel D: Outcome variable is answer score for Python						
Knows...							
C++	8.31***	10.41***	7.48***	5.77***	10.54***		5.58***
C		10.63***	8.01***	6.96***	10.73***		6.11***
Java	6.45***		6.54***	4.67***	9.53***		5.37***
C#	6.96***	9.32***		4.51***	9.66***		5.01***
Go	0.28	1.49	0.06		2.70***		0.86
JavaScript	5.65***	7.84***	4.75***	5.50***			4.49***
R	2.36**	4.26***	1.94*	1.91*	4.78***		

Notes: Each number is a t-statistic for the null hypothesis that $\beta_H + \beta_{HL} = 0$ in equation (4), with β_H being the coefficient on the language indicated in the row ("additional language") and β_{HL} being the coefficient on the interaction term with the language indicated in the column ("other language"). Standard errors are clustered by individual. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 4. Knowledge of high-effort languages and activity on Stack Exchange sites

	Soft.Eng. (1)	Code Rev. (2)	Comp.Sci. (3)	Soft.Eng. (4)	Code Rev. (5)	Comp.Sci. (6)	Soft.Eng. (7)	Code Rev. (8)	Comp.Sci. (9)
C++	0.010***	0.006***	0.002***						
	(0.0004)	(0.0003)	(0.0002)						
C++ × Python	0.044***	0.027***	0.010***						
	(0.0011)	(0.0009)	(0.0005)						
Scala				0.014***	0.006***	0.003***			
				(0.0011)	(0.0007)	(0.0005)			
Scala × Python				0.059***	0.038***	0.014***			
				(0.0030)	(0.0024)	(0.0015)			
F#							0.061***	0.033***	0.007***
							(0.0052)	(0.0037)	(0.0018)
F# × Python							0.179***	0.082***	0.041***
							(0.0115)	(0.0087)	(0.0055)
Python	0.004***	0.004***	0.001***	0.008***	0.007***	0.002***	0.009***	0.007***	0.003***
	(0.0001)	(0.0001)	(0.0001)	(0.0002)	(0.0001)	(0.0001)	(0.0002)	(0.0002)	(0.0001)
Observations	1,955,191	1,955,191	1,955,191	1,955,191	1,955,191	1,955,191	1,955,191	1,955,191	1,955,191
First Answer Year FE	YES	YES	YES	YES	YES	YES	YES	YES	YES

Notes: This table shows estimates for equation (5). The outcome variable is a binary indicator for a minimum reputation of 201 on the Stack Exchange site indicated in the column. Tests of the equality of the base coefficients, e.g., $\beta_{\text{Cpp}} - \beta_{\text{Python}} = 0$, are shown in Table 5. Tests of whether the sum of a base coefficient and its corresponding interaction term is zero, e.g., $\beta_{\text{Cpp}} + \beta_{\text{Cpp}\times\text{Python}} = 0$, are shown in Table 6. Standard errors are clustered by individual. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 5. Tests for whether one programming language predicts more Stack Exchange activity than another

	Other language						
High-effort language	C	Java	C#	Go	JavaScript	Python	R
Panel A: Outcome variable is Software Engineering activity							
Knows...							
C++	1.03	2.52**	4.20***	15.19***	12.32***	15.48***	39.35***
Scala	−7.91***	−3.67***	1.43	7.25***	2.89***	5.96***	20.23***
Clojure	2.16**	2.16**	5.56***	13.27***	6.25***	5.40***	17.44***
F#	6.78***	5.88***	2.19**	17.75***	8.40***	10.06***	21.31***
Haskell	1.99**	5.44***	7.95***	16.13***	9.17***	7.75***	21.95***
Panel B: Outcome variable is Code Review activity							
Knows...							
C++	0.06	14.34***	14.73***	8.64***	16.63***	6.59***	23.71***
Scala	−9.19***	−0.46	3.15***	0.89	0.88	−1.80*	10.44***
Clojure	1.26	3.73***	6.35***	7.54***	4.00***	3.43***	10.83***
F#	5.63***	5.91***	3.59***	10.52***	6.21***	6.92***	13.41***
Haskell	7.40***	9.59***	12.48***	14.61***	11.11***	7.03***	18.51***
Panel C: Outcome variable is Computer Science activity							
Knows...							
C++	−2.12**	10.26***	19.94***	9.51***	17.75***	6.51***	15.03***
Scala	−6.71***	2.77***	7.00***	2.14**	6.29***	1.18	6.23***
Clojure	1.14	2.56**	5.67***	5.55***	4.80***	2.35**	7.03***
F#	1.85*	3.44***	5.16***	5.95***	4.73***	2.59***	7.02***
Haskell	9.20***	10.69***	13.42***	15.12***	11.48***	9.19***	16.19***
Panel D: Outcome variable is Mathematics activity							
Knows...							
C++	1.89*	31.63***	43.59***	23.80***	38.55***	5.88***	4.89***
Scala	−17.20***	3.53***	8.93***	3.79***	7.58***	−9.42***	−7.82***
Clojure	−3.49***	3.82***	7.53***	7.23***	4.83***	−0.96	2.04**
F#	−0.90	4.18***	6.25***	8.28***	6.61***	1.06	3.85***
Haskell	10.92***	15.60***	20.12***	22.27***	17.57***	11.01***	17.70***
Panel E: Outcome variable is Statistics & Machine Learning activity							
Knows...							
C++	2.79***	16.12***	26.46***	10.34***	22.24***	−31.23***	−53.85***
Scala	1.92*	7.36***	12.96***	7.70***	11.74***	−13.62***	−50.43***
Clojure	1.91*	3.37***	5.77***	4.42***	5.41***	−5.25***	−32.24***
F#	0.71	1.88*	3.17***	4.46***	4.45***	−3.19***	−29.84***
Haskell	4.00***	5.96***	9.19***	8.56***	8.18***	−3.43***	−31.85***

Notes: Each number is a t-statistic for the null hypothesis that $\beta_H - \beta_L = 0$ in equation (5), with KnowsHighEffort_i and KnowsLowEffort_i defined using the languages indicated in the row and the column, respectively. Standard errors are clustered by individual. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 6. Tests for whether an additional programming language predicts Stack Exchange activity

	Other language						
Additional language	C	Java	C#	Go	JavaScript	Python	R
Panel A: Outcome variable is Software Engineering activity							
Knows...							
C++	42.36***	56.25***	53.37***	22.11***	54.07***	50.72***	21.80***
C		52.29***	50.43***	21.06***	50.11***	48.46***	21.71***
Java	48.73***		59.61***	22.15***	59.61***	54.32***	22.68***
C#	47.94***	60.59***		21.80***	57.55***	52.41***	21.19***
Go	17.69***	21.26***	21.12***		20.70***	20.16***	12.35***
JavaScript	43.14***	56.31***	52.36***	19.07***		49.37***	20.93***
Python	39.36***	50.44***	48.83***	19.29***	48.63***		20.25***
R	16.41***	19.67***	18.77***	10.81***	19.29***	15.71***	
Panel B: Outcome variable is Code Review activity							
Knows...							
C++	31.19***	40.85***	38.57***	18.10***	39.54***	39.26***	19.67***
C		39.00***	36.33***	17.67***	37.59***	37.20***	19.05***
Java	32.09***		40.76***	16.80***	41.28***	38.55***	19.53***
C#	30.76***	41.07***		16.47***	38.98***	35.91***	18.52***
Go	15.63***	18.19***	17.40***		18.49***	17.70***	11.05***
JavaScript	30.22***	41.26***	38.02***	16.41***		37.62***	19.06***
Python	32.34***	42.12***	37.68***	17.61***	42.29***		22.56***
R	15.84***	18.79***	17.88***	10.02***	18.79***	18.01***	
Panel C: Outcome variable is Computer Science activity							
Knows...							
C++	18.92***	26.01***	22.17***	11.78***	23.67***	24.47***	12.97***
C		25.75***	21.63***	11.03***	23.54***	23.96***	12.56***
Java	21.47***		22.32***	11.40***	23.37***	23.95***	12.64***
C#	14.26***	18.55***		9.37***	15.80***	17.79***	10.65***
Go	9.33***	11.41***	10.48***		11.06***	10.77***	7.37***
JavaScript	16.28***	18.09***	15.98***	8.78***		17.93***	11.56***
Python	19.97***	25.89***	21.76***	11.15***	24.73***		13.62***
R	10.67***	12.40***	11.15***	7.01***	12.25***	11.73***	
Panel D: Outcome variable is Mathematics activity							
Knows...							
C++	29.84***	44.34***	38.89***	15.71***	42.46***	42.46***	21.82***
C		41.37***	36.57***	15.15***	39.71***	38.14***	19.43***
Java	26.51***		36.36***	13.90***	37.75***	30.79***	16.54***
C#	20.18***	29.97***		11.15***	24.12***	23.77***	13.92***
Go	10.05***	14.03***	13.00***		14.14***	10.79***	8.11***
JavaScript	23.25***	31.10***	26.25***	11.38***		21.95***	14.69***
Python	33.45***	45.40***	38.06***	14.96***	45.42***		24.09***
R	18.37***	22.13***	19.13***	9.54***	22.04***	25.41***	
Panel E: Outcome variable is Statistics & Machine Learning activity							
Knows...							
C++	15.12***	23.67***	19.62***	8.64***	20.71***	20.31***	12.24***
C		20.98***	18.36***	8.41***	19.06***	16.00***	8.92***
Java	13.89***		19.35***	7.52***	16.97***	12.80***	7.15***
C#	8.71***	13.37***		5.23***	7.21***	6.28***	3.31***
Go	6.30***	8.08***	7.22***		7.90***	3.60***	−0.25
JavaScript	9.46***	11.68***	10.63***	5.10***		2.37**	3.05***
Python	25.30***	34.58***	25.97***	10.20***	33.27***		23.77***
R	21.58***	28.46***	21.38***	8.35***	27.07***	44.52***	

Notes: Each number is a t-statistic for the null hypothesis that $\beta_H + \beta_{HL} = 0$ in equation (5), with β_H being the coefficient on the language indicated in the row ("additional language") and β_{HL} being the coefficient on the interaction term with the language indicated in the column ("other language"). Standard errors are clustered by individual. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 7. Job posting gaps between languages within the same firm

	Postings Difference (1)	Postings Difference (2)	Postings Difference (3)
Constant	−0.507***	−1.549***	−2.972***
	(0.1254)	(0.1598)	(0.5646)
Securities Trading	−0.287	−1.523**	−3.447
	(0.2154)	(0.7461)	(2.7061)
Artificial Intelligence	−3.484***	−5.741***	−11.559***
	(0.4886)	(0.6657)	(1.5618)
Biomedical Engineering	−1.350***	−2.398***	−5.845**
	(0.3028)	(0.6418)	(2.3768)
Medicine	−0.751***	−0.238	−0.441
	(0.2087)	(0.3513)	(0.9824)
Manufacturing	0.457***	−1.000**	−1.385
	(0.1347)	(0.4202)	(1.1768)
Embedded Systems	1.306***	−2.766***	−3.283***
	(0.2055)	(0.3999)	(1.0213)
Web Applications	−1.115***	−1.361***	−2.375**
	(0.1626)	(0.3158)	(0.9734)
Observations	781,771	558,000	119,570
High-effort Language	C++	Scala	F#
Low-effort Language	Python	Python	Python

Notes: This table shows estimates for equation (7). The regression samples only include firms that have hired for both the high-effort and the low-effort language. In a problem domain δ, the posting difference between the two languages is non-zero if the sum of the constant and the domain coefficient ($\beta_0 + \beta_\delta$) is non-zero. For this null hypothesis, t-statistics are shown in Table 8. Standard errors are clustered by firm and MSA. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

TABLE 8. Tests for job posting gaps between languages

High-effort language	Other language: C	Java	C#	Go	JavaScript	Python	R
Panel A: Problem domain is Securities Trading							
Knows...							
C++	3.58***	−4.89***	−0.93	2.78***	−4.91***	−4.15***	2.98***
Scala	−2.70***	−4.36***	−5.53***	1.22	−5.50***	−4.05***	−0.46
Clojure	−2.42**	−3.18***	−3.78***	−2.88***	−3.49***	−2.75***	−2.91***
F#	−2.07**	−2.55**	−3.00***	−2.22**	−2.77***	−2.30**	−2.11**
Haskell	−1.85*	−2.84***	−3.21***	−2.02**	−3.19***	−2.54**	−3.06***
Panel B: Problem domain is Artificial Intelligence							
Knows...							
C++	5.90***	−5.65***	1.44	4.76***	−2.77***	−7.31***	2.47**
Scala	−2.87***	−9.35***	−5.99***	−0.64	−8.33***	−9.36***	−5.13***
Clojure	−5.94***	−6.98***	−6.45***	−3.84***	−6.31***	−7.23***	−6.20***
F#	−5.24***	−7.15***	−6.59***	−5.09***	−6.45***	−7.89***	−5.27***
Haskell	−4.54***	−6.14***	−5.58***	−3.43***	−6.01***	−6.40***	−5.64***
Panel C: Problem domain is Biomedical Engineering							
Knows...							
C++	4.02***	−5.01***	−0.22	3.88***	−5.89***	−5.79***	−1.83*
Scala	−2.79***	−5.34***	−4.75***	1.99**	−6.39***	−5.93***	−6.38***
Clojure	−1.15	−2.58***	−2.18**	0.12	−2.59***	−2.70***	−3.74***
F#	−3.59***	−4.21***	−5.41***	−1.54	−4.21***	−3.60***	−2.70***
Haskell	−0.54	−2.37**	−1.06	1.20	−1.91*	−1.56	−2.04**
Panel D: Problem domain is Medicine							
Knows...							
C++	3.17***	−7.49***	−6.67***	3.01***	−9.01***	−6.30***	−2.94***
Scala	0.40	−7.08***	−8.91***	2.41**	−9.27***	−5.66***	−4.72***
Clojure	−2.69***	−4.21***	−4.90***	−0.57	−5.22***	−4.05***	−3.78***
F#	−3.53***	−4.22***	−4.56***	−3.15***	−5.71***	−3.78***	−3.70***
Haskell	−2.24**	−3.76***	−4.74***	−2.03**	−5.63***	−3.22***	−3.35***
Panel E: Problem domain is Web Applications							
Knows...							
C++	4.83***	−12.14***	−3.02***	3.11***	−10.76***	−9.38***	3.44***
Scala	−3.32***	−11.83***	−7.24***	−3.52***	−15.18***	−8.37***	−1.71*
Clojure	−3.71***	−8.35***	−6.42***	−5.45***	−12.01***	−5.68***	−5.94***
F#	−3.32***	−7.41***	−5.59***	−4.93***	−10.40***	−5.09***	−3.64***
Haskell	−3.24***	−7.85***	−5.25***	−5.58***	−11.16***	−5.60***	−5.84***

Notes: Each number is a t-statistic for no within-firm difference in job postings between a pair of languages. Formally, they test the null hypothesis that $\beta_0 + \beta_\delta = 0$ in equation (7). Each panel shows results for a different domain δ. Rows correspond to language ℓ and columns correspond to language ℓ'. Standard errors are clustered by firm and MSA. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 9. Job postings by language and problem domain

	Postings (1)	Postings (2)	Postings (3)
C++ × Securities Trading	0.024	0.172***	0.148**
	(0.0576)	(0.0602)	(0.0609)
C++ × Artificial Intelligence	−0.015	0.032	0.028
	(0.0413)	(0.0337)	(0.0337)
C++ × Biomedical Engineering	−0.067	0.173*	0.207**
	(0.0695)	(0.0938)	(0.0948)
C++ × Medicine	−0.132***	−0.072**	−0.078**
	(0.0368)	(0.0332)	(0.0339)
C++ × Embedded Systems	0.006	0.103***	0.117***
	(0.0389)	(0.0273)	(0.0242)
C++ × Manufacturing	−0.090***	−0.012	−0.018
	(0.0318)	(0.0284)	(0.0289)
C++ × Web Applications	−0.147***	−0.164***	−0.176***
	(0.0343)	(0.0290)	(0.0269)
Java × Securities Trading	−0.033	0.039	0.026
	(0.0402)	(0.0480)	(0.0494)
Java × Artificial Intelligence	−0.122***	−0.114***	−0.114***
	(0.0398)	(0.0328)	(0.0312)
Java × Biomedical Engineering	0.008	0.147*	0.167*
	(0.0674)	(0.0848)	(0.0862)
Java × Medicine	−0.100***	−0.080***	−0.082***
	(0.0327)	(0.0282)	(0.0286)
Java × Embedded Systems	−0.152***	−0.224***	−0.221***
	(0.0396)	(0.0299)	(0.0272)
Java × Manufacturing	−0.154***	−0.163***	−0.175***
	(0.0337)	(0.0313)	(0.0321)
Java × Web Applications	−0.139***	−0.171***	−0.186***
	(0.0377)	(0.0335)	(0.0325)
Observations	3,122,138	3,122,138	3,122,138
Firm FE		YES	YES
Year FE			YES
MSA FE			YES

Notes: This table shows estimates for language-by-domain interaction terms in equation (8). Tests of the null hypothesis of no domain concentration difference between C++ and Java, i.e., that $(\beta_{\text{Cpp},\delta} - \beta_{\text{Cpp,WebApp}}) - (\beta_{\text{Java},\delta} - \beta_{\text{Java,WebApp}}) = 0$ for a given domain δ, are shown in Table 10. Standard errors are clustered by firm and MSA. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 10. Tests for relative concentration of high-effort languages across problem domains

	Other language						
High-effort language	C	Java	C#	Go	JavaScript	Python	R
Panel A: Problem domain is Securities Trading							
Knows...							
C++	4.37***	4.40***	4.31***	4.71***	3.88***	4.61***	4.90***
Scala	2.90***	3.07***	2.68***	2.78***	2.10**	3.25***	2.84***
Clojure	1.37	1.57	1.09	1.15	0.60	1.69*	1.09
F#	0.30	0.47	0.04	0.13	−0.33	0.55	0.05
Haskell	0.06	0.24	−0.25	−0.12	−0.75	0.35	−0.26
Panel B: Problem domain is Artificial Intelligence							
Knows...							
C++	4.44***	3.18***	2.92***	3.78***	3.19***	5.01***	5.68***
Scala	3.39***	1.73*	1.37	2.46**	1.61	3.73***	4.34***
Clojure	0.00	−0.93	−1.42	−0.50	−1.18	0.38	1.09
F#	−2.51**	−3.31***	−3.70***	−2.79***	−3.57***	−2.23**	−1.59
Haskell	0.97	0.22	−0.11	0.58	0.07	1.29	1.81*
Panel C: Problem domain is Biomedical Engineering							
Knows...							
C++	3.89***	4.26***	3.61***	3.44***	3.91***	3.78***	4.28***
Scala	2.93***	3.13***	2.38**	1.54	2.57**	2.78***	3.24***
Clojure	0.62	0.76	0.23	−0.18	0.34	0.62	0.99
F#	1.93*	2.07**	1.58	1.17	1.71*	1.90*	2.21**
Haskell	3.76***	4.37***	3.54***	3.05***	3.78***	3.74***	4.05***
Panel D: Problem domain is Medicine							
Knows...							
C++	2.83***	2.70***	2.43**	1.31	2.42**	2.78***	3.61***
Scala	2.03**	1.71*	1.56	0.49	1.45	2.02**	3.06***
Clojure	1.82*	1.72*	1.61	1.02	1.56	1.91*	2.62***
F#	−0.91	−0.94	−1.08	−1.56	−1.10	−0.66	0.10
Haskell	0.49	0.39	0.20	−0.50	0.16	0.70	1.60
Panel E: Problem domain is Manufacturing							
Knows...							
C++	4.10***	2.24**	2.33**	1.35	1.73*	3.04***	3.72***
Scala	2.38**	0.39	0.42	−0.49	−0.17	1.15	1.88*
Clojure	0.74	−0.57	−0.57	−1.09	−0.93	−0.10	0.44
F#	−0.50	−1.94*	−1.97**	−2.50**	−2.35**	−1.43	−0.78
Haskell	1.30	0.07	0.08	−0.46	−0.27	0.52	1.02

Notes: Each number is a t-statistic for no difference in domain concentration between a pair of languages. Formally, they test the null hypothesis that $(\beta_{\ell\delta} - \beta_{\ell,\text{WebApp}}) - (\beta_{\ell'\delta} - \beta_{\ell',\text{WebApp}}) = 0$ in equation (8), i.e., that for language ℓ (shown in rows) the difference between the number of postings in domain δ and the number of postings in the domain of web applications is the same as the corresponding difference for language ℓ' (shown in columns). Each panel shows results for a different domain δ. Specifications include firm, year, and MSA fixed effects. Standard errors are clustered by firm and MSA. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 11. Tests for relative domain concentration using negative binomial regressions

High-effort language	Other language						
	C	Java	C#	Go	JavaScript	Python	R
	Panel A: Problem domain is Securities Trading						
Knows...							
C++	4.68***	5.20***	4.50***	4.71***	4.42***	5.10***	5.02***
Scala	3.17***	3.64***	2.85***	3.11***	2.68***	3.62***	3.16***
Clojure	1.69*	2.12**	1.33	1.63	1.16	2.12**	1.51
F#	0.59	0.95	0.23	0.56	0.08	0.98	0.41
Haskell	0.31	0.68	−0.09	0.38	−0.27	0.75	0.15
	Panel B: Problem domain is Artificial Intelligence						
Knows...							
C++	5.23***	4.31***	3.92***	4.83***	4.44***	5.84***	6.41***
Scala	4.28***	2.89***	2.57**	3.55***	2.91***	4.52***	4.97***
Clojure	1.41	0.46	0.08	0.93	0.32	1.74*	2.29**
F#	−1.63	−2.29**	−2.58***	−1.87*	−2.49**	−1.33	−0.83
Haskell	1.86*	1.05	0.77	1.46	0.97	2.13**	2.55**
	Panel C: Problem domain is Biomedical Engineering						
Knows...							
C++	4.28***	4.24***	3.92***	4.03***	4.15***	4.13***	4.52***
Scala	3.56***	3.39***	3.03***	2.92***	3.23***	3.37***	3.74***
Clojure	2.12**	1.99**	1.68*	1.47	1.83*	2.04**	2.37**
F#	3.25***	3.14***	2.79***	2.66***	3.00***	3.14***	3.45***
Haskell	3.12***	3.07***	2.77***	2.79***	2.92***	3.03***	3.33***
	Panel D: Problem domain is Medicine						
Knows...							
C++	2.73***	2.85***	2.27**	1.66*	2.47**	3.24***	4.18***
Scala	2.39**	2.33**	1.85*	1.23	1.98**	2.79***	3.82***
Clojure	2.01**	2.02**	1.65*	1.23	1.74*	2.35**	3.15***
F#	0.07	0.06	−0.28	−0.64	−0.20	0.43	1.26
Haskell	0.78	0.76	0.36	−0.03	0.47	1.13	1.95*
	Panel E: Problem domain is Manufacturing						
Knows...							
C++	3.44***	2.51**	2.38**	1.77*	2.29**	3.05***	3.41***
Scala	2.07**	1.00	0.96	0.34	0.78	1.59	1.97**
Clojure	1.38	0.49	0.49	0.01	0.33	0.96	1.28
F#	0.34	−0.66	−0.62	−1.17	−0.85	−0.12	0.28
Haskell	1.54	0.65	0.64	0.14	0.47	1.12	1.46

Notes: Each number is a t-statistic for no difference in domain concentration between a pair of languages. Formally, they test the null hypothesis that $(\beta_{\ell\delta} - \beta_{\ell,\text{WebApp}}) - (\beta_{\ell'\delta} - \beta_{\ell',\text{WebApp}}) = 0$ in equation (9), i.e., that for language ℓ (shown in rows) the difference between the number of postings in domain δ and the number of postings in the domain of web applications is the same as the corresponding difference for language ℓ' (shown in columns). Each panel shows results for a different domain δ. Specifications include firm, year, and MSA fixed effects. Standard errors are clustered by firm and MSA. Significance codes: * $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$.

Table 12. Model parameters

Parameter	Value		Description
	$\delta = a$	$\delta = b$	
$p(\delta)$	1	1	Price of output good
$\beta(\delta)$	3/10	3/10	Labor share
$\varphi_H(\delta)$	*varies*	1	Fitness of the high-effort language
$\varphi_L(\delta)$	1	1	Fitness of the low-effort language
$\alpha(\delta)$	*varies*	1/4	Skill content of production
$N(\delta)$	*varies*	10	Team size
γ_H	1.05		Difficulty of the high-effort language
γ_L	1		Difficulty of the low-effort language
λ_H	1/2		Population share of high-ability engineers
θ_H	9/10		Latent ability of high-ability engineers
θ_L	1/10		Latent ability of low-ability engineers

Notes: This table shows the parameter values that were used to generate Figures 1 and 2. The economy consists of two domains, a and b. The figures show how equilibria and equilibrium outcomes change with $\varphi_H(a)$, $\alpha(a)$, and $N(a)$, holding all other parameters fixed.

Table 13. Occupation codes used for job postings

Code	Description
15-1251.00	Computer Programmers
15-1252.00	Software Developers
15-1253.00	Software Quality Assurance Analysts and Testers
15-1254.00	Web Developers
15-1299.07	Blockchain Engineers
17-2061.00	Computer Hardware Engineers
17-2071.00	Electrical Engineers
15-2051.00	Data Scientists
15-2051.01	Business Intelligence Analysts

Notes: The occupation codes follow the O*NET-SOC 2019 taxonomy.

Table 14. High- and low-effort programming languages

Language	First appeared	Effort	Use	Platform
C	1972	High	Mainstream[a]	–
C++	1979[b]	High	Mainstream	–
Scala	2004	High	Niche	Java
Clojure	2007	High	Niche	Java
F#	2005	High	Niche	C#
Haskell	1990	High	Niche	–
Java	1995	Low	Mainstream	Java
C#	2000	Low	Mainstream	C#
Go	2009	Low	Mainstream	–
JavaScript	1995	Low	Mainstream	–
Python	1991	Low	Mainstream	–
R	1993	Low	Niche	–

[a] C became mainstream in the 1980s but few new projects have used it since the 2000s.
[b] C++'s precursor, "C with Classes," first appeared in 1979.