TRIPLEBYTE

How to Interview Engineers

How to Interview Engineers

We do a lot of interviewing at Triplebyte. Indeed, over the last 2 years, I've interviewed just over 900 engineers. Whether this was a good use of my time can be debated! (I sometimes wake up in a cold sweat and doubt it.) But regardless, our goal is to improve how engineers are hired. To that end, we run background-blind interviews, looking at coding skills, not credentials or resumes. After an engineer passes our process, they go straight to the final interview at companies we work with (including Apple, Facebook, Dropbox and Stripe). We interview engineers without knowing their backgrounds, and then get to see how they do across multiple top tech companies. This gives us, I think, some of the best available data on interviewing.

In this blog post, I'm going to present what we've learned so far from this data. Technical interviewing is broken in a lot of ways. It's easy to say this. (And many blog posts do!) The hard part is coming up with what to do about it. My goal for this post is to take on that challenge, and lay out specific advice for hiring managers and CTOs. Interviewing is hard. But I think that many of the problems can be fixed by running a careful process [1].

The Status Quo

Most interview processes includes two main steps:

Applicant screening

In-person final interview

The goal of applicant screening is to filter out candidates early, and save engineering time in interviews. The screening process usually involves a recruiter scanning a candidate's resume (in about 10 seconds), followed by a 30-minute to 1-hour phone call. Eighteen percent of the companies we work with also use a take-home programming challenge (either in place of or in addition to the phone screen). Screening steps, interestingly, are where the significant majority of candidates are rejected. Indeed, across all the companies we work with, over 50% of candidates are rejected on the resume scan alone, and another 30% are rejected on the phone screens / take-home. Screening is also where hiring can be at its most capricious. Recruiters are overwhelmed

with volume, and need to make snap decisions. This is where credentials and pattern matching come into play.

In-person final interviews almost-universally consist of a series of 45-minute to 1-hour sessions, each with a different interviewer. The sessions are primarily technical (with one or two at each company focusing on culture fit and soft skills). The final hire/no hire decisions are made in a decision meeting after the candidate has left, with the hiring manager and everyone who interviewed the candidate. Essentially, a candidate needs at least one strong advocate and no strong detractors to be made an offer [2].

Beyond the common format, however, final interviews vary widely.

39% of the companies we work with run interviews with a marker on a whiteboard

52% allow the candidate to use their own computer (the remaining 9% are inconsistent)

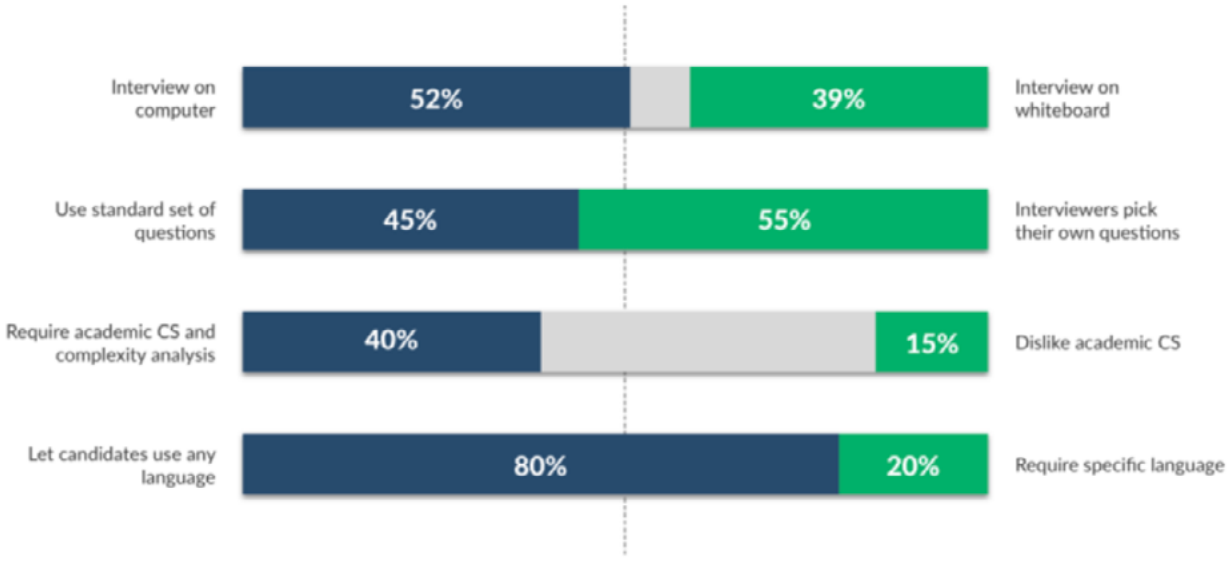
55% let interviewers pick their own questions (the remaining 45% use a standard bank of questions)

40% need to see academic CS skills in a candidate to make an offer

15% dislike academic CS (and think that talking about CS is a sign that a candidate will not be productive)

80% let candidates use any language in the interview (the remaining 20% require a specific language)

5% explicitly evaluate language minutia during the interview



Across all the companies we work with, 22% of final interviews result in a job offer. (This figure comes from asking companies about their internal candidate pipeline. Candidates applying through Triplebyte get offers after 53% of their interviews.) About 65% of offers are accepted (result in a hire). After 1 year, companies are very happy with approximately 30% of hires, and have fired about 5% [3].

False Negatives vs. False Positives

So, what's wrong with the status quo? Fire rates, after all, don't seem to be out of control. To see the problem, consider that there are two ways an interview can fail. An interview can result in a bad engineer being hired and later fired (a false positive). And an interview can disqualify someone who could have done that job well (a false negatives). Bad hires are very visible, and expensive to a company (in salary, management cost and morale for the entire team). A bad hire sucks the energy from a team. Candidates who could have done the job well but are not given the chance, in contrast, are invisible. Any one case is always debatable. Because of this asymmetry, companies heavily bias their interviews toward rejection.

This effect is strengthened by noise in the process. Judging programming skill in 1 hour is just fundamentally hard. Add to this a dose of pattern matching and a few gut calls as well as the complex soup of company preferences discussed above, and you're left with a very noisy signal.

In order to keep the false positive rate low in the face of this noise, companies have to bias decisions ever farther toward rejection. The result is a process that misses good engineers, still often prefers credentials over real skill, and often feels capricious and frustrating to the people involved. If everyone at your company had to re-interview for their current jobs, what percentage would pass? This is a scary question. The answer is almost certainly well under 100%. Candidates are harmed when they are rejected by companies they could have done great work for, and

companies are harmed when they can't find the talent they need.

To be clear, I am not saying the companies should lower the bar in interviews. Rejection is the point of interviewing! I'm not even saying that companies are wrong to fear false positives far more than false negatives. Bad hires are expensive. I am arguing that a noisy signal paired with the need to avoid bad hires results in a really high false negative rate, and this harms people. The solution is to improve the signal.

Concrete ways to reduce noise in interviews

Decide what skills you're looking for

There is not a single set of skills that define a good programmer. Rather, there is a sea of diverse skill sets. No engineer can be strong in all of these areas. In fact, at Triplebyte we often see excellent, successful software engineers with entirely disjoint sets of skills. The first step to running a good interview, then, is deciding what skills matter for the role. I recommend you ask yourself the following questions (these are questions we ask when we onboard a new company at Triplebyte).

> *Do you need fast, iterative programmers, or careful rigorous programmers?*
>
> *Do you want someone motivated by solving technical problems, or building product?*
>
> *Do you need skill with a particular technology, or can a smart programmer learn it on the job?*
>
> *Is academic CS / math / algorithm ability important or irrelevant?*
>
> *Is understanding concurrency / the C memory model / HTTP important?*

There are no right answers to these questions. We work with successful companies that come down on both sides of each one. But what is key is making an intentional choice, based on your needs. The anti-pattern to avoid is simply picking interview questions randomly (or letting each interviewer decide). When that happens, company engineering culture can skew in a direction where more and more engineers have a particular skill or approach that may not really be important for the company, and engineers without this skill (but other important skills) are rejected.

Ask questions as close as possible to real work

Professional programmers are hired to solve large, sprawling problems over weeks and months. But interviewers don't have weeks or months to evaluate candidates. Each interviewer typically has 1 hour. So instead, interviewers look at a candidates' ability to solve small problems quickly, while under duress. This is a different skill. It is correlated (interviews are not completely random). But it's not perfectly correlated. Minimizing this difference is the goal when developing interview questions.

This is achieved by making interview questions as similar as possible to the job you want the candidate to do (or to the skill you're trying to measure). For example, if what you care about is back-end programming, asking the candidate to build a simple API endpoint and then add features is almost certainly a better question than asking them to solve a BFS word chain problem. If you care about algorithm ability, asking the candidate to apply algorithms to a problem (say, build a simple search index, perhaps backed by a BST and a hashmap for improved deletion performance) is almost certainly a better problem than asking them to determine if a point is contained in a concave polygon. And a debugging challenge, where the candidate works in a real codebase, is almost certainly better than asking the candidate to solve a small problem on a whiteboard.

> ***Making interview questions as similar as possible to the job you want the candidate to do***

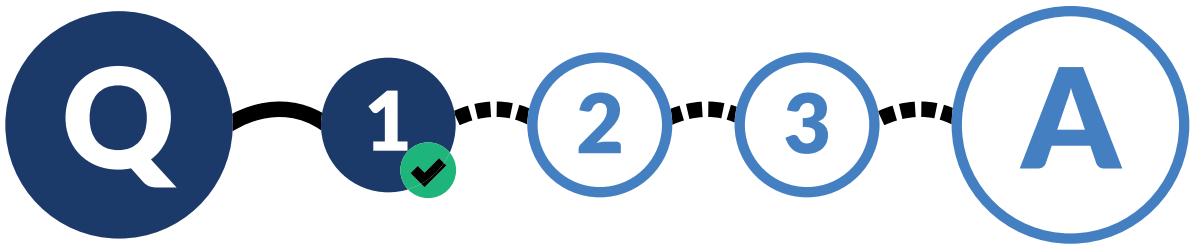
That said, there *is* an argument for doing interviews on whiteboards. As an interviewer, I don't care if an engineer has the Python itertools module memorized. I care if they can think through how to use iterators to solve a problem. By having the candidate work on a whiteboard, I free them from having to get the exact syntax right, and let them focus on the logic. Ultimately I think this argument fails, because there's just not enough justification for the different format. You can get all the benefit by allowing the candidate to work on a computer, but telling them their code does not need to run (or even better, making it an open book interview and letting them look up anything they want with Google).

There is an important caveat to the idea that interview questions should mirror work. It is important that an interview question be free from external dependencies. For example, asking a candidate to write a simple web scraper in Ruby might seem like a good real-word problem. However, if a

candidate needs to install Nokogiri (a Ruby parsing library that can be a pain to install) and they end up burning 30 minutes wrestling with the native extensions, this becomes a horrible interview. Not only has time been wasted, stress for the candidate has gone through the roof.

Ask multi-part questions that can't be given away

Another good rule of thumb for interview questions is to avoid questions that can be "given away", i.e. avoid questions where there's some magic piece of information that the candidate could have read on Glassdoor ahead of time that would allow them to answer easily. This obviously rules out brain teasers or any question requiring a leap of insight. But it goes beyond that, and means that questions need to be a series of steps that build on each other, not a single central problem. Another useful way to think about this is to ask yourself whether you can help a candidate who gets stuck, and still end the interview with a positive impression. On a one-step question, if you have to give the candidate significant help, they fail. On a multi-part problem, you can help with one step, and the candidate can then ace everything else and do well.



This is important not only because your question will leak onto Glassdoor, but also (and more importantly) because multi-part problems are less noisy. Good candidates will become stressed and get stuck. Being able to help them and see them recover is important. There is significant noise in how well a candidate solves any one nugget of programming logic, based on whether they've seen a similar problem recently, and probably just chance. Multi-part problems smooth out some of that noise. They also give candidates the opportunity to see their effort snowball. Effort applied to one step often helps them solve a subsequent step. This is an important dynamic when doing real work, and capturing it in an interview decreases noise.

To give examples, asking a candidate to implement the game Connect Four in a terminal (a series

of multiple steps) is probably a better question than asking a candidate to rotate a matrix (a single step, with some easy giveaways). And implementing k-means clustering (multiple operations that build on each other) is probably better than determining the largest rectangle that can fit under a histogram.

Avoid hard questions

If a candidate solves a really hard question well, that tells you a lot about their skill. However, because the question is hard, most candidates will fail to solve it well. The expected amount of information gained from a question, then, is heavily impacted by the difficulty of the question. We find that the optimal difficulty level is significantly easier than most interviewers guess.
This effect is amplified by the fact that there are two sources of signal when interviewing a candidate: whether they give the "correct" answer to a question, and their process / how easily they arrive at that answer. We've gathered data on this at Triplebyte (scoring questions both on whether the candidate reached the correct answer, and how much effort it took them, and then measuring which scores predict success at companies). What we found is a tradeoff. For harder questions, whether the candidate answers correctly carries most the signal. For easier questions, in contrast, most of the signal is found in the candidate's process and how much they struggle. Considering both sources of signal, the sweet spot is toward the easier end of the spectrum.

> ***We find that the optimal difficulty level is significantly easier than most interviewers guess.***

The rule of thumb we now follow is that interviewers should be able to solve a problem in 25% of the time they expect candidates to spend. So, if I'm developing a new question for a 1-hour interview, I want my co-workers (with no warning) to be able to answer the question in 15 minutes. Paired with the fact that we use multi-part real-world problems, this means that the optimal interview question is really pretty straightforward and easy.

To be clear, I am not arguing for lowering the bar in terms of pass rate. I am arguing to ask easy questions, and then including in your evaluation how easily the candidate answered the questions. I'm arguing for asking easy questions, but then judging fairly harshly. This is what we find optimizes

signal. It has the additional benefit of being lower stress for most applicants.

To give examples, asking a candidate to create a simple command line interface with commands to store and retrieve key-value pairs (and adding functionality if they do well) is probably a better problem than asking a candidate to implement a parser for arithmetic expressions. And a question involving the most common data structures (lists, hashes, maybe trees) is probably better than a question about skiplists, treaps or other more obscure data structures.

Ask every candidate the same questions

Interviews are about comparing candidates. The goal is to sort candidates into those who can contribute well to the company and those who can't (and in the case of hiring for a single position, select the best person who applies). Given this, there is no justification for asking different questions to different candidates. If you evaluate different candidates for the same job in different ways, you are introducing noise.

The reason it continues to be common to select questions in an ad-hoc fashion, I think, is because it's what interviewers prefer. The engineers at tech companies typically don't like interviewing. It's something they do sporadically, and it takes them away from their primary focus. In order to standardize the questions asked to every candidate, the interviewers would need to take more time to learn the questions and talk about scoring and delivery. And they would need to re-do this every time the question changed. Also, always asking the same question is just a little more tedious.

Unfortunately, the only answer here is for the interviewers to put in the effort. Consistency is key to running good interviews, and that means asking every candidate the same questions, and standardizing delivery. There's simply no alternative.

Consider running multiple tracks

In conflict with my previous point, consider offering several completely different versions of your interview. The first step when designing an interview is to think about what skills matter. However, some of the answers might be in conflict! It's pretty normal, for example, to want some really mathy engineers, and some very productive / iterative engineers (maybe even for the same role). In this case, consider offering multiple versions of the interview. The key point is that you need to be at enough scale that you can fully standardize each of the tracks. This is what we do at Triplebyte. What we've found is that you can simply ask each candidate which type of interview they'd prefer.

Don't let yourself be biased by credentials

Credentials are not meaningless. Engineers who have graduated from MIT or Stanford, or worked at Google and Apple really are better, as a group, than engineers who did not. The problem is that the vast majority of engineers (myself included) have done neither of these things. So if a company relies on these signals too heavily, they will miss the majority of skilled applicants. Giving credentials some weight in a screening step is not totally irrational. We don't do this at Triplebyte (we do all of our evaluation 100% background blind). But giving some weight to credentials when screening might make sense. Letting credentials sway final interview decision, however, does not make sense. And we have data showing that this happens. For a given level of performance on our background-blind process, candidates with a degree from a top school go on to pass their interviews at companies at a 30% higher rate than candidates without the name-brand resume. If interviewers know that candidate has a degree from MIT, they are more willing to forgive rough spots in the interview.

> ***There are so many different ways to be a skilled engineer, that almost no candidates can master them all.***

This is noise, and you should avoid it. The most obvious way is just to strip school and company names from resumes before giving them to your interviewers. Some candidates may mention their school or company, but we do all our interviews without knowing the candidates' backgrounds, and it's actually pretty rare for a candidate to bring it up during technical evaluation.

Avoid hazing

One of the ugliest ways interviews can fail is that they can take on an aspect of hazing. They're not

just about evaluating the skill of a candidate, they're also about a group or team admitting a member. In that second capacity, they can become a rite of passage. Yes, the interview is stressful and horrible, but we all did it so so should the candidates. This can be accentuated when a candidate is doing badly. As an interviewer, it can be frustrating to watch a candidate beat their head against a problem, when the answer seems so obvious! You can get short tempered and frustrated. This, of course, only increases the stress for the applicant in a downward spiral.

This is something you want to stay a mile away from. The solution is talking about the issue and training the interviewers. One trick that we use is, when a candidate is doing really poorly, to switch from evaluation mode, where the goal is to judge the candidate, to teaching mode, where the goal is to make the candidate understand the answer to the question. Mentally making the switch can help a lot. When you're in teaching mode, there is no reason to withhold information or be anything other than friendly.

Make decisions based on max skill, not average or min skill

So far, I've only talked about individual questions, not the final interview decision. My advice here is to try to base the decision on the maximum level of skill that the candidate shows (across the skill areas you care about), not the average level or minimum level.

This is likely what you are already doing, intentionally or not! The way hire/no hire decisions are made is that everyone who interviewed a candidate gets together in a meeting, and an offer is made if at least one person is strongly in favor of hiring, and no one is strongly against. To get one interviewer to be strongly in favor, what a candidate needs to do is ace one section of the interview. Across our data, max skill is the attribute that's most correlated with acing at least one section of a company's interview. However, to be made an offer, a candidate also needs no one to be a strong no against them. Strong noes come when a candidate looks really stupid on a question.

Here we find just a great deal of noise. There are so many different ways to be a skilled engineer,

that almost no candidates can master them all. This means if you ask the right (or wrong) question, any engineer can look stupid. Candidates get offers, then, when at least one interview lines up with an area of strength (max skill) and no areas line up with a significant weakness. The problem is that this is noisy. The same engineer who fails one interview because they looked stupid on a question about networking passes other interviews with flying colors because that topic did not come up.

The best solution, I think, is for companies to focus on max skill, and be a little more comfortable making offers to people who looked bad on parts of the interview. This is, looking for strong reasons to say yes, and not worrying so much about technical areas where the candidate was weak. I don't want to be absolute about this. There are of course technical areas that just matter to a company. And deciding that you want to have a culture where everyone on the team is at a certain level in a certain area may well make sense. But focusing more on max skill does reduce interview noise.

> ***There are so many different ways to be a skilled engineer, that almost no candidates can master them all.***

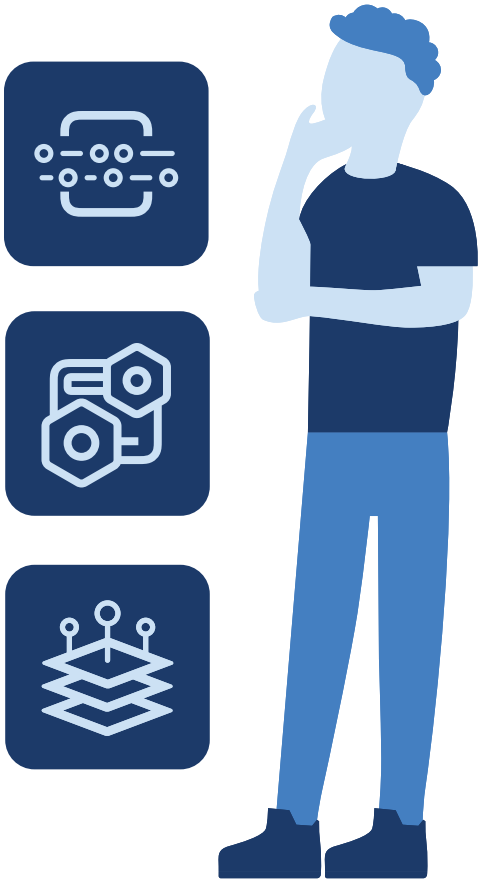
Why do interviews at all?

A final question I should answer is why do interviews at all? I'm sure some readers have been gritting their teeth, and saying "why think so much about a broken system? Just use take-home projects! Or just use trial employment!" After all, some very successful companies use trial employment (where a candidate joins the team for a week), or totally replace in-person interviews with take-home projects. Trial employment makes a lot of sense. Spending a week working beside an engineer (or seeing how they complete a substantial project) almost certainly provides a better measure of their abilities than watching them solve interview problems for 1 hour. However, there are two problems that keep trial employment from replacing standard interviews:

Trial employment is expensive for the company. No company can spend a full week with every person who applies. To decide who makes it to the trial, companies must use some other interview process.

Trial employment (and large take-home projects) are expensive for the candidate. Even when they are

> *paid, not all candidates have the time. An engineer working a full-time job, for example, may simply not be able to take the time off. And even if they can, many won't. If an engineer already has job offers in hand, they are less likely be willing to take on the uncertainty of a work trial. We see this clearly among Triplebyte candidates. Many of the best candidates (with other offers in hand) will simply not do large projects or work trials.*



The result of this is that trial employment is an excellent option to offer some candidates. I think if you have the scale to support multiple tracks, adding a trial employment track is a great idea. However, it's not viable as a total replacement for interviews.

Talking to candidates about past experience is also sometimes put forward as a replacement for technical interviews. To see if a candidate can do good work in the future, the logic goes, just see what they've done in the past. We've tested this at Triplebyte, and unfortunately we've not had great results. Communication ability (ability to sell yourself) ended up being a stronger signal than technical ability. It's just too common to find well-spoken people who exaggerate their role (take credit for a team's work), and modest people who downplay what they did. Given enough time and enough questioning, it should be possible to get to the bottom of this. However, we found that within the time limits of a regular interview, talking about past experience is not a general replacement for interviewing. It is a great way to break the ice with a candidate and get a sense of their interests (and judge communication ability and perhaps culture fit). But it's not a viable total replacement for interviews.

Good things about programming interviews!

I want to end this post on a more positive note. For everything that's wrong with interviews, there is a lot that's right about them.

Interviews are direct skill assessment. I have friends who are teachers, who tell me that teacher interviews are basically a measure of communication ability (ability to sell yourself), and a credential. This seems to be true of many many professions. Silicon Valley is not a perfect meritocracy. But we do at least try to directly measure the skills that matter, and stay open to the idea anyone with those skills, regardless of background, can be a great engineer. Credential bias often stands in the way of this. But we've been able to mostly overcome this at Triplebyte, and help a lot of people with unconventional backgrounds get great tech jobs. I don't think Triplebyte would be possible, for example, in the legal field. The reliance on credentials is just too high.

> ***We do at least try to directly measure the skills that matter, and stay open to the anyone with those skills***

Programmers also choose interviews. While this is a very controversial topic (there are certainly programmers who feel differently), when we've run experiments offering different types of evaluation, we find that most programmers still pick a regular interview. And we find that only a minority of programmers are interested in companies that use trial employment or take-home projects. For better or worse, programming interviews seem to be here to say. Other types of evaluation are great supplements, but they seem unlikely to replace interviews as the primary way engineers are evaluated. To misquote Churchill, "Interviews are the worst way to evaluate engineers, except for all the other ways that have been tried from time to time."

Conclusion

Interviewing is hard. Human beings are hopelessly complex. On some level, judging human ability in a 4-hour interview is just a fool's errand. I think it's important to stay humble about this. Any interview process is bound to fail a lot of the time. People are just too complex.

But that's not an argument for giving up. Trying to run a meritocratic process is better than not trying. At Triplebyte, our interview is our product. We brainstorm ideas, we test them, and we improve over time. This, I think, is the approach that's needed to improve how engineers are hired.

Author Info

Ammon Bartram, CEO & Co-Founder of Triplebyte

[1] I'm limiting this to technical skill assessment. I'll be writing a future post about culture fit, behavioral interviews and non-technical evaluation.

[2] There is of course variation here. At opposite ends of the spectrum we see companies that require a unanimous yes from every interviewer to make a hire, and companies where the hiring manager is solely responsible for the decision.

[3] These numbers are what companies report about their internal candidates. And the numbers vary widely between companies (they report fire rates, for example, as low as 1% and as high as 30%). The numbers are significantly better for Triplebyte candidates. So far, our candidates at companies have received offers after 53% of interviews, and 2% have been fired.

Access an elite pool of diverse talent that you won't find anywhere else.

Triplebyte is rewriting the code for technical hiring.

Triplebyte enables companies hiring technical talent to source highly-skilled candidates from diverse backgrounds, streamline hiring processes and ultimately reduce time-to-hire.

We provide a marketplace of technically pre-screened engineers evaluated through a background-blind process that we match to companies based on their specific hiring requirements. The result is more, better qualified candidates that companies can review, evaluate and progress, without the need to spend valuable time on pre-screening.

Hire the top 3% of engineers

Cut time to hire in half

Drive an onsite-to-hire rate to 40%

Request your free demo today: **hire.triplebyte.com/demo-request**

Triplebyte.com