TECH RECRUITMENT 101

How Software Teams Actually Work

PART OF THE *TECH RECRUITMENT 101* SERIES

How Software Teams Actually Work

A practical map of who does what, where decisions happen, and how work really moves

CONTENTS

How software teams work has always mattered to recruiters. Lately, the gap between the org chart and the real work has become harder to ignore. I wrote this because I kept seeing the same pattern: recruiters were asked to hire for titles, while hiring managers were really feeling pain in handoffs, ownership, review loops, release process, and team design. When that happens, the brief sounds reasonable, but the search drifts.

This topic also came up again and again in the Tech Recruitment 101 series. People wanted a clearer mental model of what sits behind titles like backend engineer, product manager, QA, platform engineer, or engineering manager. They wanted to understand how work moves from idea to release, where engineering joins too late, why collaboration breaks down, and how AI is changing team workflows without removing the need for judgment. That is the practical problem this guide tries to solve.

So I focused on the parts recruiters need most in live work. We start with the difference between reporting lines and working lines. Then we look at what each function actually owns, because titles drift but ownership tells the truth. From there, we follow the path from problem definition to discovery, scoping, build, review, release, and follow-up. We also look at the places where teams get stuck: unclear ownership, hidden dependencies, competing incentives, and briefs that ask one person to compensate for a broken system.

If you use the ideas here well, you will read hiring signals more accurately. You will ask sharper intake questions. You will screen for the real job, not the label on the req. And you will spot sooner when a team needs a hire, when it needs a different profile than the title suggests, and when it may need a redesign more than another search.

The team is not the org chart

On paper, a software team looks neat. You see titles, reporting lines, and a manager at the top. In real work, that tells you very little. The useful map is not who reports to whom. It is who depends on whom.

The same title can mean very different work. A backend engineer on one team may build mostly independently. On another, the role is full of trade-offs with product, design, QA, and infrastructure. Same title, different job.

That is why I split every team into two maps: reporting lines and working lines. Reporting lines tell you who approves compensation, headcount, and formal priorities. Working lines tell you who decides scope, who reviews code, who tests it, who unblocks releases, and who gets pulled into the Friday-afternoon dependency surprise. The second map is the one you recruit for.

Team performance depends on the system around the role, not just the person in it. Google Cloud's 2025 DORA research—DORA is a long-running software delivery research program focused on what helps teams build and ship software effectively—describes AI adoption as a systems problem and notes that 90% of organizations have adopted at least one internal platform, with stronger outcomes when that platform is high quality (Google Cloud DORA 2025 (https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report)). In plain English, even good people work differently depending on the team around them.

Start with the core functions around the role. Engineers build and change software, but they also review code, clarify requirements, fix production issues, and work around missing context. Product managers shape what gets built and why. Designers turn intent into usable flows and often surface hidden complexity. QA may sit inside the squad, in a separate function, or barely exist as a formal role, which changes what engineers must own. Engineering managers set direction, remove blockers, and decide how the team works. Then come the nearby functions that may or may not appear in the brief at first: data, DevOps or SREs (Site Reliability Engineers), the teams that usually handle deployment, infrastructure, system stability, and incident response, security, and support. If they affect the work, they belong on the map.

This matters because vague briefs often reflect vague team design. Atlassian's *State of Product 2026* found that 49% of respondents cite competing incentives or internal politics as a

barrier to collaboration, 43% cite lack of leadership support, and 35% cite unclear roles and responsibilities (Atlassian (https://dam-cdn.atl.orangelogic.com/AssetLink/iv3u81ou0u70gsy04ia61oci00d0h0je.pdf)). When collaboration breaks down, managers often blame the missing hire. Sometimes the real problem is the system.

Use one habit on every intake call: map the team around the role before you source a single person.

- **Map the role before you source**
  - Who does this person work with daily?
  - Who approves their work?
  - Who reviews code or output?
  - Where does work usually get stuck?
  - Which team joins too late?
  - What must this person decide alone?

If you ask those six questions, the brief sharpens fast. You learn whether the role needs depth, coordination, product judgment, or tolerance for chaos. After building recruiting teams a few times, I can tell you this small map saves more time than another expensive dashboard nobody opens after week two.

What each function actually owns

Titles drift. Work does not.

If you want to read a role correctly, ignore the label for a minute and ask a simpler question: what decisions does this person make, what problems do they carry, and what happens if they do the job badly? That is how you spot real ownership.

Engineers turn vague intent into working behavior. That usually includes technical design, estimates, code review, debugging, and some level of production responsibility. In one team, an engineer may fix alerts at night. In another, they only support issues during business hours. Both are engineering roles, but the ownership is different.

A brief that says “strong coder” is often missing the hard part. Many teams need engineers who can judge trade-offs, question requirements, and keep a feature healthy after release. That matters even more now that AI can speed up drafting code but not remove the need for review and verification. The 2025 Stack Overflow Developer Survey (https://survey.stackoverflow.co/2025/ai) found that 46% of developers distrust AI output accuracy, compared with 33% who trust it.

Product managers own the problem shape. They decide what deserves attention, which trade-offs are acceptable, and what success should look like. Designers usually own user flows, interaction patterns, interface clarity, and often the uncomfortable question of whether something should exist in this form at all.

Quality assurance, or QA, owns confidence, not just test cases. In some companies QA runs manual tests. In others QA builds automation, defines risk areas, improves release readiness, and coaches the team on quality practices. Sometimes QA is embedded. Sometimes it is shared. Sometimes a company says QA is “strategic” when it really means nobody funded test automation. You can smile politely, then ask better questions.

FUNCTION	TEAM A	TEAM B
Engineer	Builds and tests code	Builds, reviews, supports production
QA	Runs manual regression testing, meaning rechecking existing features to make sure new changes did not break them	Sets strategy and automation
Designer	Makes UI mockups	Owns flows and user research
PM	Writes tickets	Shapes problems and tradeoffs

Engineering managers own team performance. They hire, coach, manage delivery health, reduce friction, and create conditions where good work keeps happening. Some still code a little. Some do not. The useful question is not whether they can open an editor. It is whether they own staffing, prioritization, process health, and the growth of the team.

Shared ownership is where briefs get sloppy. Testing is the classic example. One company expects QA to catch defects at the end. Another expects engineers to own most testing and QA to focus on risk and tooling. A third has no QA and wants senior engineers who can design quality into the workflow. Those are different hiring problems wearing similar clothes.

The same pattern shows up elsewhere. A product manager may own prioritization, but engineers may still need product judgment. Designers may own research, but product may run interviews too. Engineering managers may own hiring, but senior engineers often shape the interview bar. If you recruit from titles alone, you will miss strong candidates and send the wrong profiles.

AI is adding new title drift, not removing it. “AI engineer” may mean someone building applications on top of large language models. “Machine learning engineer” may mean someone training, evaluating, and deploying models. “Platform engineer” may support the internal systems that let teams ship software and, increasingly, deploy and monitor models safely. A platform

team usually provides shared internal tools and services for other engineers—for example, a deployment system that helps product teams release software safely without rebuilding the same tooling each time. The 2025 DORA report (https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report) makes the point clearly: AI impact depends on the surrounding system.

When I read a brief, I look for ownership verbs. Who defines? Who decides? Who reviews? Who tests? Who approves release? Who gets pulled in when something breaks? Ask those questions, and the team map becomes much clearer.

How work moves from idea to release

Software rarely moves in a straight line. It starts with a problem, not a feature. A customer struggles to do something. A team sees a cost spike. Sales keeps hitting the same objection. Someone decides the problem matters enough to spend time on it.

That first stage already tells you something about the hire. If the team is strong at problem definition, they usually want engineers who ask good questions early. If they skip that stage and jump to solutions, they often need people who can handle ambiguity and recover from churn.

Product usually shapes the problem first. Designers may join early to test assumptions or sketch flows. Engineers may join early too, though not always. That matters. Atlassian's State of Product 2026 (https://dam-cdn.atl.orangelogic.com/AssetLink/iv3u81ou0u70gsy04ia61oci00d0h0je.pdf) found that 80% of product teams do not involve engineering in ideation, problem definition, or roadmap creation. When engineering joins late, roles often require more rework tolerance and more diplomacy.

Then comes discovery, if the team actually does discovery and does not just rename panic. In a healthy version, product, design, and engineering test whether the problem is real, whether the idea is useful, and what constraints will shape the solution.

Scoping comes next. The team decides what will ship now, what can wait, and what is too risky or expensive. Product pushes on value and timing. Engineering looks at capacity, technical risk, and dependencies. Design pushes on usability. Nobody gets everything they want, which is normal and cheaper than everyone getting their way.

What drives those decisions depends on the team. On a new product team, scope changes because the team is still learning what users want. On a platform team, reliability, security, or compliance may set hard boundaries. On a migration, old systems bend the plan. On an AI feature team, scope may depend on data access, model behavior, governance, and review workflows as much as code. The 2025 DORA report from Google Cloud (https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report) makes the same point: AI adoption works as a systems problem, not a tools problem.

After scope, design and implementation start to overlap. Designers refine flows and edge cases. Engineers turn those decisions into technical plans. If the team has platform engineers, they may provide the internal systems and tooling that let product teams ship safely. If the team has SREs, they may help keep systems stable, watch for problems through monitoring, and reduce risk when changes are released.

Dependencies change the shape of a role. A backend engineer on a small team may mostly build features in one codebase. At a larger company, the same title may spend half the job coordinating with security, data, infrastructure, and other service owners. The difference should change how you screen.

Testing does not happen only at the end, even if some teams behave as if reality begins at QA. Good teams test throughout. Engineers write automated checks. QA may run manual tests, write test plans, or focus on the risk that new changes could break existing features. Product and design may review whether the result solves the original problem, not just whether the button clicks.

AI is changing this stage too, but not by removing judgment. The 2025 Stack Overflow Developer Survey (https://survey.stackoverflow.co/2025) found that more developers distrust AI output accuracy than trust it, and many say debugging AI-generated code takes more time. A team using AI tools still needs people who review carefully, test thoroughly, and know when “almost right” is still wrong.

Code review now sits closer to the center of delivery than many recruiters realize. GitHub reported in March 2026 that Copilot code review now accounts for more than one in five code reviews on GitHub (GitHub Blog (https://github.blog/ai-and-ml/github-copilot/60-

million-copilot-code-reviews-and-counting/)). If a manager says the team is AI-forward, ask where AI shows up in review, testing, and release decisions.

Release is where process differences become obvious.

Common path from idea to release

1. **Problem** — Team defines what needs fixing
2. **Discovery** — Check value, risk, and constraints
3. **Scoping** — Decide what ships now
4. **Design** — Shape UX and technical approach
5. **Build** — Implement with reviews and tests
6. **Release** — Ship with approvals or rollout steps
7. **Follow-up** — Measure impact and fix issues

Some teams release many times a day. Others ship on fixed release trains with approvals, change windows, and enough coordination overhead to qualify as a weather system. Your job is not to judge the process. It is to understand what kind of person succeeds inside it.

After release, the team watches for bugs, performance issues, user confusion, support tickets, and business impact. Sometimes the result is a quick patch. Sometimes it is a rollback. Sometimes the team learns the original idea was wrong. That is not failure. That is software work.

If you want better hiring context, ask three questions. What stage consumes the most team energy? Where does work usually get blocked? What will this hire make faster, safer, or clearer? Those answers will tell you how work really moves and why the team needs this person now.

Where collaboration breaks down

Most hiring pain starts long before recruiting does. It starts in the gaps between functions.

The first breakdown is unclear ownership. A team may say product owns priorities, engineering owns delivery, and design owns user experience. Fine. But who decides when scope must shrink? Who owns release quality? Who says no when a request lands mid-sprint, meaning during a short planned work cycle, often one or two weeks? If nobody can answer in one sentence, the role is already harder than the brief suggests.

The second breakdown is mismatched expectations. Product may expect fast delivery. Engineering may expect stable priorities. Design may expect time to explore. None of those expectations are absurd on their own. The problem is that teams often discover the conflict late.

This is common, not rare. In Atlassian's *State of Product 2026* (https://dam-cdn.atl.orangelogic.com/AssetLink/iv3u81ou0u70gsy04ia61oci00d0h0je.pdf), 35% of respondents cited unclear roles and responsibilities as a barrier to collaboration. The same report found that 49% pointed to competing incentives or internal politics, and 43% cited lack of leadership support for cross-team collaboration. Sometimes the problem is not a missing hire. It is a team design problem wearing a hiring hat.

Hidden dependencies create another kind of mess. A backend engineer may be blocked by an infrastructure change. A mobile team may wait on an API (application programming interface), which is a way different software systems communicate with each other. QA may need test data that nobody prepared. Security may join late and stop the release. Work looks slow from the outside, but the real issue is that several pieces had to line up and did not.

Weak technical leadership makes this worse. A strong engineering lead reduces ambiguity, makes trade-offs visible, and protects the team from random priority changes. A weak one leaves every hard decision floating in the room until it lands on whoever is most tired.

Then there are job descriptions written around hope rather than reality. You will see requests for someone senior, hands-on, strategic, fast, collaborative, detail-oriented, and deeply technical. That person may exist, but they are not a staffing plan. When a manager asks for all of that at once, I slow the conversation down. Do they need deep technical judgment, team leadership, a fixer who can walk into chaos, or simply relief because the current team is overloaded?

MANAGER SAYS	POSSIBLE REALITY	BETTER QUESTION
Senior and hands-on	Needs credibility in the code	What hard problems will they own?

MANAGER SAYS	POSSIBLE REALITY	BETTER QUESTION
Strategic and fast	Needs trade-off judgment	Where do priorities usually collide?
Cross-functional	Needs influence across functions	Who must they align with weekly?
Can wear many hats	Team lacks structure	What work has no clear owner today?

If you understand team friction, you can read between the lines. If product changes direction often, the role may need someone comfortable with ambiguity. If engineering is frustrated by constant interrupts, the team may need stronger planning or a lead who can push back. If QA arrives at the end, the team may need engineers with better testing habits, not just more testers.

Candidate language matters too. “I worked cross-functionally” tells you almost nothing on its own. Ask what that looked like in practice. Did they help shape product decisions early? Work with design on trade-offs before building? Own release quality with QA? Or just sit in a heroic number of meetings and call it collaboration?

A good follow-up question is: “Tell me about a time two functions wanted different things. What did you do?” You learn whether the candidate negotiated scope, clarified trade-offs, escalated well, or waited for someone else to sort it out.

AI adds speed in some parts of the workflow, but it also adds coordination work. Teams use it for coding, testing, documentation, and pull request review. A pull request is a proposed code change submitted for review before being merged into the main codebase. According to Google Cloud's 2025 DORA report (https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report), AI adoption works best as a systems capability, not just a tools decision. The 2025 Stack Overflow Developer Survey (https://survey.stackoverflow.co/2025) found that more developers distrust AI output accuracy than trust it, and many say debugging AI-generated code takes extra time. So if a manager says the team is "AI-forward," do not stop at tool names. Ask who reviews AI-generated code, how quality is checked, and whether security or QA had to change their process.

You do not need to become an expert in every technical workflow. You do need to hear when a team is hiring to absorb friction, not just to fill a seat. Once you can hear that, your intake questions improve and your screening gets sharper.

Using the team structure map in real recruiting work

A team structure map matters only if you use it in live work. The point is not to produce a pretty diagram. The point is to stop recruiting from titles and start recruiting from how work actually moves.

Build the map in intake, not after the search goes sideways. Keep it light. I use six fields: role purpose, adjacent functions, decision partners, key handoffs, likely tensions, and what success looks like after the first few months.

If the hiring manager says, “We need a senior backend engineer,” pause there. Ask what the person will make better, who they work with every week, who signs off on decisions, where work gets stuck now, and what would be visibly different after a few months. A real role gets clearer under those questions. A fake role gets louder.

Here are prompts you can use in intake:

- **Intake prompts that reveal the real role**
  - What problem does this hire solve now?
  - Who do they work with every week?
  - Who shares or blocks key decisions?
  - What handoff breaks most often today?
  - What tension comes with this role?
  - What proves success by month three?

Role purpose comes first because it filters everything else. “Build features” is not a purpose. “Reduce failed payments.” “Stabilize a fragile service.” “Help a product trio ship faster.” Those are purposes. They tell you whether to look for depth, judgment, speed, patience, or some mix of them.

Adjacent functions tell you how social the role really is. A software engineer on a tight product trio, meaning product, design, and engineering working closely together, with product and design will need stronger discovery habits, trade-off thinking, and comfort with ambiguity. That matters because many product teams still keep engineering out of early shaping work. Atlassian’s *State of Product 2026* reports that 80% of product teams do not involve engineering in ideation, problem definition, or roadmap creation Atlassian – State of Product 2026 (https://dam-cdn.atl.orangelogic.com/AssetLink/iv3u81ou0u70gsy04ia61oci00d0h0je.pdf).

Decision partners show you where influence sits. Does the engineer decide implementation alone? Does the engineering manager decide scope in practice? Does product drive priority while security or SREs can veto release risk? If you miss this, you screen the wrong kind of candidate.

Key handoffs expose the real shape of the team. A role that depends on a centralized QA team needs different habits from one with embedded QA. In the first case, candidates may need

stronger documentation, cleaner acceptance criteria, and patience with queue-based testing. In the second, you may care more about daily collaboration and fast feedback loops.

Likely tensions are where many searches quietly fail. Atlassian's *State of Product 2026* says 49% of respondents cite competing incentives or internal politics as a main barrier to collaboration, 43% cite lack of leadership support for cross-team collaboration, and 35% cite unclear roles and responsibilities Atlassian – State of Product 2026 (https://dam-cdn.atl.orangelogic.com/AssetLink/iv3u81ou0u70gsy04ia61oci00d0h0je.pdf). When a manager says, "We need someone proactive," I often hear, "This team has messy boundaries and no one agrees who owns the ugly bits."

That map should change your sourcing. If the team needs someone to stabilize a service, source engineers who have reduced incidents, improved test coverage, cleaned handoffs, or made delivery more predictable. If the team needs feature velocity, source people with faster delivery in similar environments. If the team is "AI-forward," do not stop at tool names. Ask how code review, testing, and release readiness work. Google Cloud's 2025 DORA report argues that AI adoption is a systems issue, not just a tooling issue, and says 90% of organizations have adopted at least one internal platform. It also finds that a high-quality internal platform strengthens AI's positive effect on performance Google Cloud / DORA – 2025 State of AI-Assisted Software Development (https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report).

Use the map in screening too. Screen for the work, not the stack list. For a collaborative product role, ask: “Tell me about a feature where product or design changed the direction. How did you respond?” For a platform role, ask: “What kind of internal users did you support, and how did you decide what to standardize?” For a stabilization hire, ask: “What did you do when a team was shipping, but trust in the code was low?”

Candidates need the map as much as you do. Share the role in plain language. “This team ships customer features fast, but handoffs to QA are slow.” “This manager wants someone who can calm a noisy service and coach others through better review habits.” Good candidates lean in when the problem is clear. The wrong ones usually keep talking about titles.

Debriefs get better when you bring the map back into the room. Instead of “strong communicator” or “not senior enough,” ask whether the candidate matched the role purpose, worked well with the real decision partners, and showed the right instincts for the key tension.

Watch for red flags early:

- **Red flags in the brief**
  - Ownership is fuzzy
  - Title is doing too much work
  - The must-haves conflict
  - A key stakeholder is missing
  - One hire is meant to fix three team problems

One more practical test helps. Ask whether the team needs a person or a redesign. If they want one engineer to ship features, repair architecture, improve process, align product, mentor juniors, and calm a difficult manager, that is not a profile. That is untreated org debt.

From my years doing this, I learned a simple rule the hard way more than once: once you understand how the team actually works, recruiting gets calmer. Not easy, unfortunately. But calmer. You ask better questions, write better outreach, qualify people more accurately, and spend less time cleaning up confusion that should have been caught in intake.

The useful shift is simple: stop treating the title as the job. Look instead at ownership, handoffs, decision points, and where work gets stuck. If you carry that map into intake, sourcing, screening, and debriefs, you will make better calls. You will catch the difference between a genuine hiring need and untreated org debt. You will also hear more clearly when a team says it wants speed,

but actually needs stability, sharper product judgment, stronger QA habits, or better collaboration with security, platform, or design.

A practical next step is to take one open role and redraw it using the six fields introduced earlier in this section: role purpose, adjacent functions, decision partners, key handoffs, likely tensions, and what success should look like after the first few months. Then compare that map to the current job description. The gaps will tell you what to fix first. In my experience, that one exercise improves a search faster than another round of keyword tuning.

If this way of thinking helps, the other Tech Recruitment 101 books build on it. Some go deeper into technical roles and signals. Others focus on intake, screening, and partnering better with hiring managers. They all come back to the same idea: recruiting gets much easier when you understand the work behind the title.

Glossary

- **AI (Artificial Intelligence)** — Software systems that can generate, analyze, or act on information in ways that support human work. In this book, it mainly refers to tools used in software development workflows like coding, testing, and review.
- **AI engineer** — A role that may focus on building applications that use AI systems, especially large language models. The exact scope can vary a lot from company to company.
- **API (Application Programming Interface)** — A way for different software systems to communicate with each other. Recruiters often hear about APIs when teams depend on other services or teams to build features.
- **automation** — Using software or scripts to handle repeatable work with less manual effort. In engineering teams, this often shows up in testing, deployment, and quality processes.
- **backend engineer** — An engineer who works on the server-side parts of software, such as business logic, services, and data handling. Depending on the team, the role may be mostly hands-on coding or heavily cross-functional.
- **build** — The stage where engineers implement the solution in code. It often includes coding, reviews, and testing rather than just writing software alone.
- **candidate** — A person being considered for a role. In this book, the term often matters in a technical context because

recruiters are assessing fit against real engineering work, ownership, and team dynamics.

- **code review** — A process where someone checks proposed code changes before they are accepted. It helps catch issues, improve quality, and share knowledge across the team.
- **codebase** — The main collection of source code for a software product or service. Recruiters may hear this when a role focuses on one system versus coordinating across many systems.
- **coding** — Writing software code to implement features or solve technical problems. In this book, coding is only one part of engineering work, alongside review, testing, and judgment.
- **compliance** — Requirements a team must meet for legal, regulatory, or internal policy reasons. Compliance can limit what a team can ship and how quickly it can move.
- **Copilot** — GitHub's AI tool used to assist with software development tasks such as generating code and helping with code review. In this text, it is referenced as part of AI-assisted delivery.
- **cross-functional** — Work that requires collaboration across different functions like engineering, product, design, QA, or security. A cross-functional role usually depends on influence and coordination, not just technical skill.
- **data** — In this context, a nearby technical function that may support software teams with data systems, analysis, or infrastructure. Its involvement can change the shape of a role and its dependencies.

- **debugging** — The process of finding and fixing problems in software. It is a key part of engineering work, especially when code or AI-generated output is unreliable.
- **delivery** — The process of getting software from idea through development and into release. Recruiters may hear it used to describe how reliably a team ships work.
- **deployment** — The act of releasing software changes into an environment where they can run. Teams like DevOps, platform, or SRE may be involved in making deployments safe and repeatable.
- **dependencies** — Other people, teams, systems, or services a team needs in order to get work done. Hidden dependencies are a common reason roles become more complex than their titles suggest.
- **Designer** — A role focused on user flows, interaction patterns, interface clarity, and sometimes research. On some teams, designers also help shape the problem before engineers build.
- **design** — The work of shaping how a product should behave and feel for users, including flows and interfaces. It can also refer to technical design, depending on context.
- **DevOps** — A technical function or approach focused on improving how software is built, deployed, and operated. In many companies, DevOps-related teams support release process, infrastructure, and reliability.
- **discovery** — An early stage where teams test whether a problem is real, whether an idea is useful, and what

constraints matter. It helps teams avoid building the wrong thing too quickly.

- **DORA** — A software delivery research program focused on what helps teams build and ship software effectively. In the text, DORA research is used to support points about AI adoption and internal platforms.
- **edge cases** — Less common situations or user scenarios that still need to be handled correctly. Teams that care about edge cases usually need stronger design, testing, and product judgment.
- **embedded QA** — A QA setup where the QA person works closely inside a specific team rather than as a centralized shared function. This usually changes how quickly feedback happens and who owns testing work day to day.
- **engineer** — A person who builds and changes software. Depending on the team, engineers may also own design trade-offs, testing, code review, debugging, and production support.
- **engineering lead** — A technical leader who helps reduce ambiguity, make trade-offs clear, and guide engineering decisions. This person may or may not be a formal manager.
- **engineering manager** — A manager responsible for team performance, staffing, coaching, and delivery health. Some still code, but the core of the role is enabling the team to work effectively.
- **feature** — A piece of product functionality delivered to users. Teams may be asked to build features, but the text emphasizes

understanding the underlying problem instead of just the requested output.

- **follow-up** — The stage after release where the team checks results, watches for issues, and decides what to fix or change next. This is where teams learn whether the release actually worked.
- **GitHub** — A widely used platform for hosting code and managing software collaboration. In the book, it is cited in relation to AI-assisted code review.
- **ideation** — The stage where teams explore and form ideas about what could be built. It often happens before formal scoping or roadmap decisions.
- **incident response** — The process of handling serious problems or outages in live systems. Teams like SRE often help manage incident response and reduce service risk.
- **infrastructure** — The underlying technical systems and environments that software runs on. Infrastructure work often affects deployment, reliability, and cross-team dependencies.
- **intake** — The early recruiting and hiring-manager discussion where the role is defined. In this book, intake has a technical dimension because recruiters are mapping ownership, dependencies, and team workflow.
- **interface** — The part of a product users interact with, such as screens, controls, and layouts. Designers often shape interface clarity as part of the user experience.
- **internal platform** — Shared internal systems and tools that help software teams build, deploy, and operate software more

effectively. A strong internal platform can improve developer productivity and support AI adoption.

- **large language models** — AI models trained on large amounts of text that can generate or analyze language. In this book, they are mentioned as technology AI engineers may build on top of.
- **Machine learning engineer** — A role that may focus on training, evaluating, and deploying machine learning models. Like many technical titles, the exact responsibilities vary by company.
- **main codebase** — The primary version of the code that the team treats as the official source. Changes usually need review before being merged into it.
- **manager** — A person responsible for directing people or teams. In this text, managers often appear in a technical team context, especially hiring managers and engineering managers.
- **manual regression testing** — Rechecking existing features by hand to make sure new changes did not break them. It is often contrasted with automated testing.
- **manual tests** — Tests performed by a person rather than by automation. These are often used to check behavior, edge cases, or release readiness.
- **merged** — Added into the main codebase after a code change has been reviewed and accepted. Recruiters may hear this in discussions of pull requests and code review workflow.

- **migration** — Work to move from an old system, tool, or architecture to a new one. Migration projects often create extra technical constraints and dependencies.
- **mobile team** — A team focused on building software for mobile devices. Its work often depends on APIs, backend systems, and release coordination.
- **model** — In this context, an AI or machine learning system used to generate outputs or make predictions. Teams working with models may need to evaluate, deploy, and monitor them safely.
- **monitor** — To watch systems, services, or models for performance, errors, or unusual behavior. Monitoring helps teams spot problems after changes are released.
- **org debt** — Organizational problems or poor team design that create friction and get treated like hiring problems. It is similar to technical debt, but in people and structure rather than code.
- **platform engineer** — An engineer who supports the internal systems and tooling other teams use to build, deploy, and run software. Platform roles often focus on standardization, reliability, and developer productivity.
- **platform team** — A team that provides shared internal tools and services for other engineering teams. Their work helps product teams ship software without rebuilding the same tooling repeatedly.
- **PM** — Short for product manager. In the table, it refers to the person shaping problems, priorities, and trade-offs.

- **problem definition** — The work of clearly identifying what issue the team is trying to solve before jumping into solutions. Strong problem definition usually leads to better scoping and better hiring signals.
- **product manager** — A role that shapes what gets built, why it matters, and what trade-offs are acceptable. Product managers often own prioritization and help define success.
- **production** — The live environment where real users use the software. Production responsibility often means handling issues, alerts, and service reliability after release.
- **production support** — Work involved in maintaining and fixing software after it is live. This can include handling incidents, investigating bugs, and responding to alerts.
- **profile** — The combination of skills, experience, and working style needed for a role. In this book, recruiters are encouraged to define the real profile behind a title.
- **pull request** — A proposed code change submitted for review before being merged into the main codebase. It is a common step in collaborative software development.
- **QA (Quality Assurance)** — A function focused on building confidence that software works as intended. QA may do manual testing, automation, risk assessment, and release-readiness work depending on the company.
- **quality assurance** — The practice of making sure software is reliable, tested, and ready to release. It covers much more than just finding defects at the end.

- **regression testing** — Rechecking existing functionality to confirm that new changes did not break it. This can be done manually or through automation.
- **release** — The step where software changes are shipped to users or production systems. Release processes may include approvals, rollout steps, monitoring, and coordination with other teams.
- **release trains** — A scheduled release model where teams ship at set times rather than continuously. This usually adds more planning, approvals, and coordination overhead.
- **reporting lines** — The formal management structure showing who reports to whom. The text contrasts this with working lines, which better explain how work actually happens.
- **req** — Short for job requisition, the formal hiring request for a role. In this book, recruiters are warned not to confuse the req label with the real job.
- **review workflows** — The steps and processes used to review work before it moves forward, such as code review or approval checks. In AI-related work, review workflows may also cover model or quality oversight.
- **risk areas** — Parts of a system or release that are more likely to cause problems or need extra attention. QA and engineering often focus testing and review on these areas.
- **roadmap** — A plan that shows intended product direction and priorities over time. Product teams often use roadmaps to guide what gets worked on next.

- **rollback** — Reverting a release or change because it caused problems. The need for rollbacks often points to release risk and production responsibility.
- **scoping** — Deciding what will be included now, what will wait, and what is too risky or expensive. It is a key point where product, design, and engineering trade off against each other.
- **screen** — To assess whether a candidate fits the role. In this text, screening is tied to understanding actual technical work, not just matching titles or keywords.
- **security** — A technical function focused on protecting systems, software, and data from risk. Security teams may affect architecture, release timing, and approval processes.
- **senior backend engineer** — A more experienced backend engineer expected to handle not just coding but stronger judgment, ownership, and collaboration. The exact expectations depend heavily on team structure.
- **senior engineer** — An experienced engineer who is usually trusted with harder technical problems, more ownership, and broader judgment. In some teams, senior engineers also influence hiring, quality, and team practices.
- **service** — A software system or component that provides a specific function, often to other systems or users. Teams may need to stabilize or support a service when reliability is poor.
- **service owners** — The people or teams responsible for a given service. Roles that depend on many service owners usually require more coordination.

- **ship** — A common software term meaning to release or deliver software to users or production. Teams that “ship” often are usually measured on delivery speed and reliability.
- **software engineer** — A general engineering role focused on building and maintaining software. The actual work can range from feature development to platform, reliability, or cross-functional product work.
- **software teams** — Groups of people who work together to build, test, release, and maintain software. Their structure, ownership, and dependencies shape what each role really requires.
- **source** — To search for and identify potential candidates. In this book, sourcing depends on understanding the technical context of the role, not just its title.
- **source code** — The human-readable code developers write to create software. Source code is what gets reviewed, tested, and merged during development.
- **sprint** — A short planned work cycle, often one or two weeks, used by some teams to organize work. Mid-sprint changes often create tension around priorities and scope.
- **SREs (Site Reliability Engineers)** — Engineers focused on keeping systems reliable, stable, and well-monitored. They often help with incident response, release risk, and production health.
- **stack** — The collection of technologies a team uses, such as programming languages, frameworks, and infrastructure. The

book warns recruiters not to focus on stack alone at the expense of understanding the real work.

- **stabilization hire** — A hire brought in to make systems or delivery more reliable and predictable. This kind of role often involves reducing incidents, improving testing, and calming team chaos.
- **technical design** — The engineering plan for how software should be built. It includes choices about architecture, constraints, and trade-offs.
- **technical leadership** — Guidance that helps teams make sound engineering decisions and work effectively through complexity. Weak technical leadership often increases confusion, delays, and role ambiguity.
- **technical risk** — The chance that a technical decision, dependency, or implementation issue will cause problems. Teams assess technical risk during scoping, design, and release planning.
- **test automation** — Using software tools to run tests automatically instead of relying only on manual checking. It helps teams test more consistently and at greater speed.
- **test cases** — Specific scenarios used to check whether software behaves correctly. QA owns more than test cases alone, but they are a basic tool in quality work.
- **test data** — Data prepared for use in testing software. Missing or poor test data can block QA and slow releases.
- **test plans** — Structured plans for what will be tested, how, and where the biggest risks are. QA may use test plans to

guide release readiness.

- **testing** — The work of checking whether software behaves correctly and safely. It can include manual tests, automated checks, regression testing, and release-risk assessment.
- **tickets** — Written work items used to track tasks, bugs, or features. In the text, writing tickets is presented as a narrower version of product work than shaping problems.
- **tooling** — Internal tools and systems that help engineers build, test, deploy, and support software. Better tooling often improves speed, consistency, and reliability.
- **trade-offs** — Decisions where improving one thing usually means giving up something else, such as speed versus quality or scope versus risk. Strong technical and product judgment often shows up in how people handle trade-offs.
- **trio** — A close working group of product, design, and engineering. A product trio usually shapes problems and decisions together early in the workflow.
- **UI (User Interface)** — The visible parts of a product that users interact with. UI work often includes layouts, controls, and mockups.
- **user experience** — How a product feels and works for the user in practice. Design often owns this, but product and engineering also affect it through decisions and trade-offs.
- **user flows** — The paths users take through a product to complete tasks. Designers often define flows to make interactions clear and usable.

- **user research** — Structured learning from users to understand needs, behavior, and pain points. On some teams, designers own this; on others, product may share it.
- **working lines** — The practical network of who depends on whom to get work done. Unlike reporting lines, working lines show who reviews, approves, tests, unblocks, and influences delivery.
- **workflow** — The sequence of steps and collaboration patterns through which work gets done. Team workflows often shape the real demands of a role more than the title does.
- **workflows** — Multiple processes or operating patterns used by teams to move work forward. In this text, the term appears especially around team collaboration and AI-enabled work.
- **UX (User Experience)** — A shorthand for the overall experience a user has with a product. In the release-path example, UX is part of what design helps shape.

About This Book

About the Author

Natalia Romanova

Natalia is the founder of Talantrix, an applicant tracking system for tech recruiters. She began her career as a software engineer, later moved into engineering management, and eventually led several software companies, including ones she founded. Along the way she repeatedly had to build technical recruiting capability, and over time ended up learning a lot about hiring engineers—more than she ever expected to.

About Talantrix

Talantrix is an intelligent applicant tracking system built by recruiters, for recruiters. Designed specifically for technical hiring, it helps recruitment teams automate the busywork — from CV parsing and skill-based search to pipeline management — so they can focus on what matters most: connecting with top engineers and making great hires.

Learn more at https://talantrix.com (https://talantrix.com)

The Tech Recruitment 101 Series

A practical book series for recruiters who work with engineering teams. Each volume tackles a specific challenge in technical hiring — from screening candidates to understanding code — with clear explanations, real-world examples, and actionable frameworks.

Revision 1.1 — 6 April 2026

Tech Recruitment 101

A practical book series for recruiters who work with engineering teams. Each volume tackles a specific challenge in technical hiring — from screening candidates to understanding code — with clear explanations, real-world examples, and actionable frameworks.