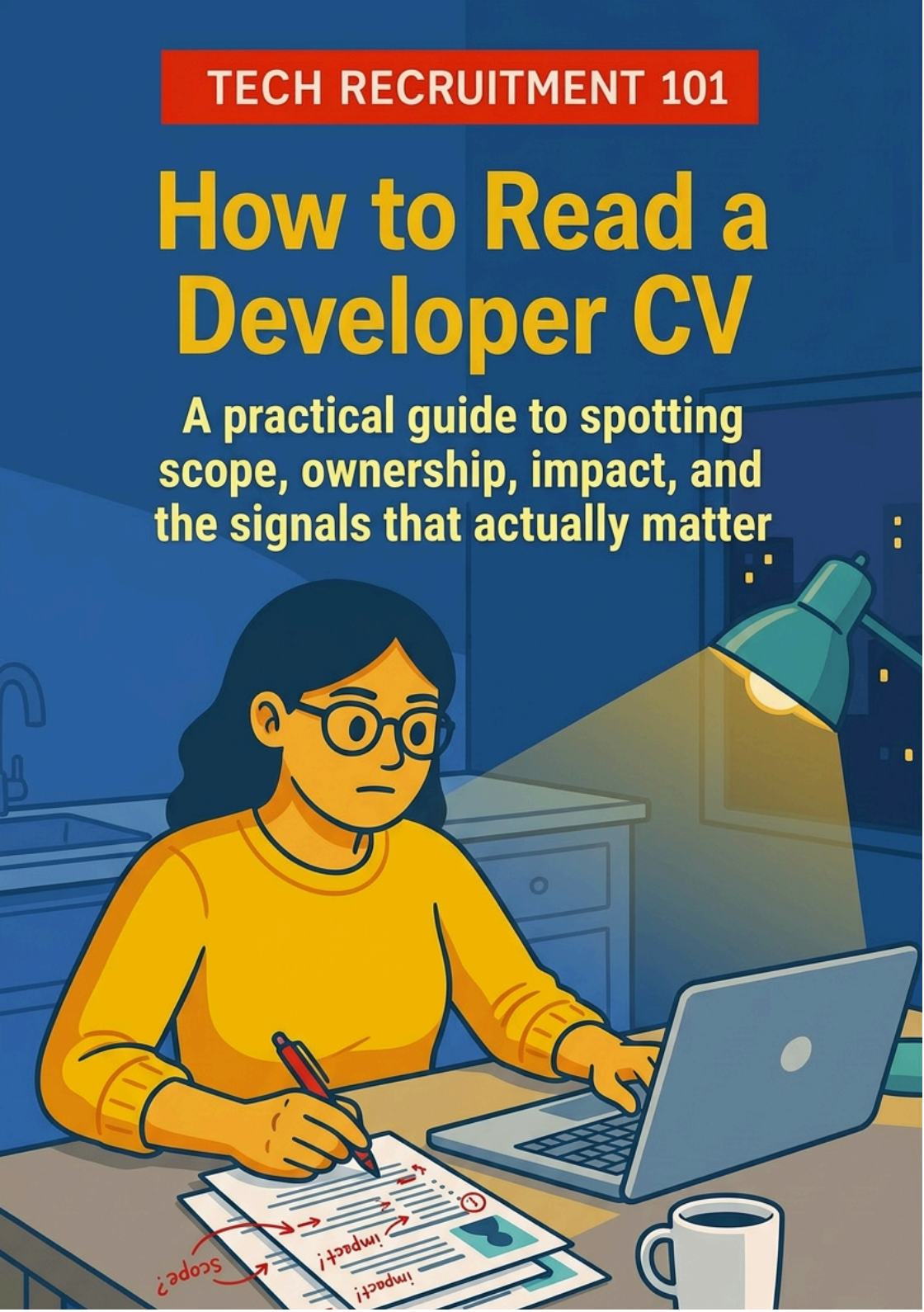
TECH RECRUITMENT 101

How to Read a Developer CV

A practical guide to spotting scope, ownership, impact, and the signals that actually matter



PART OF THE *TECH RECRUITMENT 101* SERIES

How to Read a Developer CV

A practical guide to spotting scope, ownership, impact, and the signals that actually matter

CONTENTS

How to Read a Developer CV

I wrote this book because I kept seeing the same problem from both sides of hiring. Recruiters were asked to screen technical candidates quickly, but the signals on the page were often weak, inflated, or simply hard to interpret. Good people were missed. Glossy CVs got too much credit. Everyone lost time.

After spending years close to engineering work and hiring engineers in different settings, I stopped treating developer CVs as biographies and started treating them as evidence. That shift made screening much sharper. It also made interviews better, because the goal stopped being keyword matching and became something more useful: building a grounded hypothesis about what the person has actually done.

That is what this book is for. I want to help you read a developer CV with enough technical judgment to separate real signal from noise, ask better follow-up questions, and avoid the common screening mistakes that create expensive optimism.

This is one book in a broader series on practical technical recruiting. The theme across all of them is simple: less ritual, more evidence. Developer hiring already has enough ceremony.

What a developer CV is actually telling you

The first reset is simple: a developer CV is not a life story. It is compressed evidence.

Your job is not to admire how complete it looks. Your job is to infer, as accurately as you can, what kind of work this person has actually done.

That matters because a polished CV can describe shallow work very well, and an awkward CV can hide a strong engineer. I have seen both often enough that I no longer confuse writing quality with technical depth. That is even more important now that many candidates use AI tools to polish CV wording. Developers themselves report clear productivity benefits from AI, but much less confidence in its handling of complex work in the Stack Overflow Developer Survey 2024 (https://survey.stackoverflow.co/2024/ai/) and rising workplace use in GitHub's 2024 U.S. developer survey (https://github.blog/wp-content/uploads/2024/08/2024-Developer-Survey-United-States.pdf). So elegant phrasing deserves a little less trust than it used to.

The checklist I come back to is straightforward: **scope, ownership, outcomes, red flags, and strong signals**.

What to read for in every developer CV

- ✓ Scope: how big, complex, or critical was the work?
- ✓ Ownership: did they drive it or just touch it?
- ✓ Outcomes: what changed because of their work?
- ✓ Red flags: where is the wording evasive or inconsistent?
- ✓ Strong signals: what suggests real depth or trust?

Scope tells you the size and shape of the problem. Was this a marketing site, an internal tool, a payments system, a mobile app, or infrastructure other engineers depended on?

Ownership tells you how close they were to the core work. Did they implement assigned tasks, or did they make decisions, handle trade-offs, run releases, or carry production responsibility?

Outcomes tell you whether the work mattered. In engineering, that often means operational results rather than revenue: faster releases, fewer incidents, lower latency, better reliability, less manual work. The DORA 2024 Accelerate State of DevOps Report (https://dora.dev/research/2024/dora-report/2024-dora-accelerate-state-of-devo

ps-report.pdf) is useful here because it gives you concrete language for delivery impact: deployment frequency, lead time for changes, change fail rate, and time to restore service.

Red flags are usually small credibility problems, not dramatic scandals. Vague bullets, inflated verbs, mismatched timelines, or long tool lists with no context all belong here.

Strong signals are details that are hard to fake by accident: clear problem statements, constraints, sensible trade-offs, production responsibility, and outcomes that fit the system being described.

This is why keyword matching is such a weak substitute for judgment. A stack is not a skill level. If two candidates both list Python, React, Amazon Web Services (AWS), and Kubernetes, that only tells you they were somewhere near those tools. It does not tell you what they owned.

A short example makes the point.

One CV line, three very different realities

CV BULLET	COULD MEAN	WHAT TO VERIFY
Built a recommendation service in Python	Designed and owned the service end to end	Architecture, trade-offs, production use
Built a recommendation service in Python	Implemented part of an existing design	Which component, whose design, level of autonomy
Built a recommendation service in Python	Supported nearby work with minor code changes	Exact contribution and decision-making role

Same words, very different substance. Until you know the problem, the system, and the candidate's role in the decisions, you do not know what that line is worth.

The same applies to AI-heavy CVs. "Worked on LLM features" or "did prompt engineering" means little by itself. If the work was serious, you usually see traces of system design around it: evaluation, testing, retrieval design, context handling, versioning, guardrails, and measurable outcomes. That pattern shows up consistently in official guidance from OpenAI (https://platform.openai.com/docs/guides/prompt-engineering/strategies-to-improve-reliability),

Anthropic (https://docs.anthropic.com/en/docs/prompt-engineering), and Microsoft (https://learn.microsoft.com/en-us/azure/developer/ai/advanced-retrieval-augmented-generation).

Read the CV as evidence, not marketing copy. Your goal is to leave the page with a working hypothesis: what this person has likely done, what they likely owned, and what you need to verify next.

Reading for scope and ownership

The biggest reading mistake in developer hiring is confusing participation with responsibility.

A CV can make someone sound busy, modern, and technically adjacent to all sorts of important things without ever telling you what was actually theirs. I read every role through two lenses: **scope** and **ownership**. If you get those right, seniority becomes much easier to judge.

Scope is the size and shape of the work. What product area did they support? Was it customer-facing, internal, core infrastructure, or legacy maintenance? Was the system central to the business, or contained enough that mistakes were survivable before lunch?

Business criticality matters. Keeping a dull but essential system alive often requires more judgment than shipping a shiny feature everyone likes to talk about. Recruiters tend to over-credit visible product work and under-credit operationally important work. I would avoid that habit.

Ownership is more personal. I want to know whether the candidate implemented assigned tasks or drove a meaningful part of the work. Did they define the approach? Make technical

choices? Coordinate with product, design, security, or infrastructure? Handle production issues? Mentor someone else? Push the thing through when it got messy, which it usually does?

Weak phrasing often gives itself away: “involved in,” “participated in,” “worked on,” “helped with.” None of those phrases are useless, but all of them describe proximity better than ownership.

Stronger lines usually contain four things: a problem, a defined responsibility, an action or decision, and an outcome.

Quick screen: scope and ownership

- ✓ What system, product area, or business function did they work on?
- ✓ Was the work central, isolated, internal, customer-facing, or maintenance?
- ✓ What exactly was theirs versus shared with the team?
- ✓ Do I see decisions, trade-offs, or project leadership?
- ✓ Are outcomes tied to the work, not just activities or tools?
- ✓ Does the wording show ownership, or just proximity?

This is also why years of experience can mislead you. Time around engineering is not the same as growing responsibility. In practice, a skills-first reading is more reliable than title matching, which is consistent with LinkedIn's Skills Signal Report 2025 (https://economicgraph.linkedin.com/content/dam/me/business/en-us/talent-solutions/resources/pdfs/the-skills-signal-report-2025.pdf), LinkedIn's Future of Recruiting 2025 (https://business.linkedin.com/content/dam/me/business/en-us/talent-solutions/resources/pdfs/future-of-recruiting-2025.pdf), and Indeed's skills-first hiring guidance (https://www.indeed.com/employers/skillsfirsthiring). Skills-based evaluation is more closely tied to quality hiring than pedigree, titles, or degree filters.

So when you screen a CV, do not ask, "Does this sound impressive?" Ask, "What did this person own, and how big was the problem?" That question alone removes a surprising amount of fog.

Stack, projects, and the story behind the tools

The technology list is usually the noisiest part of a developer CV. It looks concrete, so recruiters often overweight it. But a stack without context tells you very little. It is a list of ingredients, not the meal.

What I look for first is whether the stack is anchored to actual work. A candidate who names a few technologies and explains what they built with them often gives you more signal than someone with a long parade of frameworks. In my experience, ten tools on a CV often means light exposure to most of them.

The project description is where the real signal lives. Read it for clues about the technical environment. Did they build an Application Programming Interface (API), meaning the layer other systems interact with? Did they work on a user interface (UI), where speed and usability matter? Did they mention databases, messaging, deployment pipelines, monitoring, or incident response?

Context matters as much as the tools. “Built internal tools” could mean a one-off script or a business-critical platform. “Worked on enterprise software” could mean deep subsystem ownership or several months of moving buttons through a committee. You need

delivery context: production software or prototype, consulting work or long-term product, startup ambiguity or tightly governed platform work.

Collaboration patterns help too. If a CV mentions Product Managers, designers, security teams, or Site Reliability Engineers (SREs), the engineers responsible for keeping production systems stable and recoverable, that usually points to software with real dependencies and real users.

I also pay attention to signs of technical maturity. Continuous Integration and Continuous Delivery (CI/CD), which means automated systems for testing and shipping code, usually suggests a team that ships real software rather than emailing zip files and hoping for the best. Monitoring, alerts, rollback, release cadence, and reliability language are useful clues. If a candidate says they improved deployment frequency or reduced incidents, that is much better than "optimized processes." Again, the DORA 2024 Accelerate State of DevOps Report (https://dora.dev/research/2024/dora-report/2024-dora-accelerate-state-of-devops-report.pdf) gives you a practical vocabulary for reading these claims.

How to read stack descriptions

WEAK SIGNAL	STRONGER SIGNAL	WHAT IT TELLS YOU
"Java, Spring, AWS"	"Built Spring APIs on AWS for payment flows used in production"	Tool plus business context
"Worked on React app"	"Owned React checkout UI, improved load time and error handling"	Scope and ownership
"Used Docker and Kubernetes"	"Containerized services and supported Kubernetes deployments"	Operational depth
"AI chatbot project"	"Built retrieval pipeline, evals, and model-integrated support workflow"	Applied AI engineering

Portfolio links and GitHub can help if they show craftsmanship: finished work, readable documentation, tests, sensible structure. Fifty abandoned side projects with heroic names are less informative. I say that without judgment. Engineers are allowed to have a graveyard.

AI projects need the same treatment as everything else. You will see bullets about LLMs, copilots, agents, prompts, chatbots, and other fashionable nouns. Some are substantial. Some are decorative.

A serious applied AI project usually leaves traces: evaluation pipelines, model choice, versioning, retrieval-augmented generation (RAG), which means fetching relevant information before the model answers, context management, guardrails, or integration into a real workflow. The official documentation from OpenAI (https://platform.openai.com/docs/guides/prompt-engineering/strategies-to-improve-reliability), Anthropic (https://docs.anthropic.com/en/docs/prompt-engineering), and Microsoft (https://learn.microsoft.com/en-us/azure/developer/ai/advanced-retrieval-augmented-generation) all point in the same direction. Production-grade LLM work is moving away from clever wording and toward system design.

That matters because AI language on CVs is increasing. GitHub's Octoverse 2024 (https://github.blog/news-insights/octoverse/octoverse-2024/) reported strong growth in generative AI project activity, so more candidates now have AI-flavored project titles whether the underlying work was deep or not. Read past the label.

The practical rule is simple: do not recruit the stack in isolation. Recruit the work behind it.

Looking for outcomes and career shape

A developer CV becomes more useful when you stop reading it as a list of jobs and start reading it as a pattern over time.

One role can be noisy. Several roles usually tell a story. I am not looking for a perfectly tidy career. I am looking for repetition, progression, and believable cause and effect.

What keeps repeating? Backend systems, mobile apps, data pipelines, developer tooling, reliability work, messy integrations, customer-facing product delivery? Those patterns usually tell you more than the best-written bullet on the page.

Then ask how ownership changes over time. Do they move from implementing features to owning services, from local delivery to cross-team influence, from assigned tasks to judgment under ambiguity? Seniority often leaves traces even when titles do not.

Outcomes need the same discipline. A metric on a CV is not proof. Some numbers are real, some are rounded, and some look as if several departments collaborated on the fiction. I do not reject metrics. I just read them skeptically.

The useful question is whether the result has a believable mechanism behind it. “Improved performance by 70%” tells me very little. “Reduced API latency by redesigning database queries and caching hot paths” is stronger because the cause and effect make sense.

> ⚠ **Do not confuse numbers with evidence**
>
> A percentage is only useful if the candidate explains what changed, what they owned, and why the result is believable.

Career shape is where recruiters can become unfair. Frequent moves are not automatically a red flag. Long tenure is not automatically a green flag. The real question is whether the moves add depth, responsibility, domain knowledge, or range.

A candidate with several short stints may be random. They may also be someone repeatedly trusted to handle migrations, stabilization, modernization, or first-version product work. Contractor-heavy CVs often look messy until you read for the through-line.

The same goes for startup-to-enterprise shifts, or the reverse. Neither direction is better by default. What matters is whether the candidate expanded capability or just changed logos.

Read across the timeline

- ✓ What technical problems repeat?
- ✓ How does ownership change over time?
- ✓ Are outcomes explained with believable cause and effect?
- ✓ Do the moves add depth, range, or responsibility?
- ✓ Is there a coherent pattern beneath messy formatting?

By the end of CV review, you do not need certainty. You need a grounded hypothesis you can test in the first call.

Red flags, strong signals, and common screening mistakes

A developer CV is full of ambiguity. Your job is not to punish imperfection. It is to separate harmless noise from signs that the story does not hold together.

A real red flag is something that makes the candidate's scope, ownership, credibility, or fit materially less clear. That is different from a CV that is merely annoying, sparse, or badly formatted.

The red flags I care about most are these:

- persistent vagueness across multiple roles
- stack-list inflation with no visible depth
- big impact claims with no mechanism behind them
- timelines or promotions that do not match the work described
- repeated adjacency to important work without clear accountability

Messy formatting is not a red flag. Imperfect English is not a red flag unless the role depends heavily on written communication. A sparse CV is not a red flag. Some excellent engineers write CVs as if under mild protest.

Strong signals are usually plain. Clear ownership language helps: designed, implemented, migrated, introduced, reduced, operated. So does thoughtful project framing: problem, constraint, decision,

outcome. Evidence of shipping also matters. Developers create value by delivering things that reached real users, real teams, or real systems.

I also like technical range with depth somewhere. Breadth shows adaptability. Depth shows they can stay with a hard problem long enough to solve it.

For AI-heavy CVs, the same rule applies. Do not over-credit titles like “AI engineer” or bullets like “used ChatGPT” or “worked with Copilot.” AI tool use at work is becoming normal according to GitHub’s 2024 U.S. developer survey (https://github.blog/wp-content/uploads/2024/08/2024-Developer-Survey-United-States.pdf), and more AI projects are appearing generally, as GitHub’s Octoverse 2024 (https://github.blog/news-insights/octoverse/octoverse-2024/) shows. That makes labels cheaper. Stronger signals are still evaluation, success criteria, retrieval design, grounding, versioning, guardrails, and measurable outcomes, which align with guidance from OpenAI (https://platform.openai.com/docs/guides/prompt-engineering/strategies-to-improve-reliability), Anthropic (https://docs.anthropic.com/en/docs/prompt-engineering), and Microsoft (https://learn.microsoft.com/en-us/azure/developer/ai/advanced-retrieval-augmented-generation).

> ⚠ **A warning worth keeping nearby**
>
> Do not confuse a missing keyword with a missing capability. Do not confuse a famous employer with strong ownership. Do not confuse polished wording with technical depth.

The screening mistakes I see most often are predictable: rejecting candidates for not matching the wish list exactly, overweighting employer brands, treating every GitHub link as meaningful, assuming every AI title reflects depth, and mistaking confidence for competence because the bullets sound expensive.

This is where structured screening helps. Greenhouse's guidance on structured hiring (https://www.greenhouse.com/guidance/how-to-implement-structured-hiring-steps-2-and-3-defining-your-scorecard-and-planning-your-interview?lang=en) gets the core idea right: define what you need to verify before interviewing, then assess it consistently. For developer CVs, that scorecard can stay simple.

Quick screening checklist

- ✓ Is the candidate clear about what they owned?
- ✓ Do the stack claims show depth somewhere?
- ✓ Are outcomes described with believable context?
- ✓ Do dates and seniority align with the work described?
- ✓ Are you reacting to true risk or just imperfect presentation?
- ✓ What follow-up question would confirm or break the story?

Conclusion

If you take one thing from this book, let it be this: a developer CV is not there to impress you. It is there to give you enough evidence to form a hypothesis.

Read for scope, ownership, outcomes, and credibility. Treat stacks as context, not proof. Be skeptical of glossy wording, especially now that AI can polish bullets in seconds. And resist the lazy shortcuts: keyword bingo, title worship, employer-brand worship, and the belief that a neat CV must describe neat work.

In my experience, good technical recruiting gets better the moment you stop trying to decode perfection and start looking for substance. That is true in this book, and it is a theme in the rest of the series as well. The tools change, the buzzwords rotate, and every year a new category of software arrives to save us from thinking. The job, annoyingly, still requires thinking.

If this book helps you ask sharper follow-up questions and reject fewer strong candidates for the wrong reasons, it has done its job.

Glossary

- **AI (Artificial Intelligence)** — Software systems that perform tasks such as generating text, writing code, or making predictions.
- **API (Application Programming Interface)** — The part of a system that lets other systems communicate with it.
- **AWS (Amazon Web Services)** — A widely used cloud platform where companies run applications, databases, and infrastructure.
- **CI/CD (Continuous Integration / Continuous Delivery)** — Automated processes that help developers test and ship code more reliably.
- **DORA metrics** — Common software delivery measures: deployment frequency, lead time for changes, change fail rate, and time to restore service.
- **GitHub** — A platform used to store code and collaborate on software projects.
- **Guardrails** — Rules or controls that limit unsafe, inaccurate, or unwanted behavior in an AI system.
- **Kubernetes** — Software used to run and manage containers across servers.
- **LLM (Large Language Model)** — A type of AI model that generates and understands text.
- **Monitoring** — Tools and processes used to observe system health, performance, and failures in production.

- **RAG (Retrieval-Augmented Generation)** — An AI approach where the system retrieves relevant documents or data before generating an answer.
- **React** — A popular tool for building web user interfaces.
- **SRE (Site Reliability Engineer)** — An engineer focused on keeping production systems reliable, observable, and recoverable.
- **Stack** — The set of technologies used to build and run a software system.
- **UI (User Interface)** — The part of a product a user sees and interacts with directly.
- **Versioning** — Keeping track of software or model versions so changes can be managed, tested, and rolled back if needed.

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.

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.