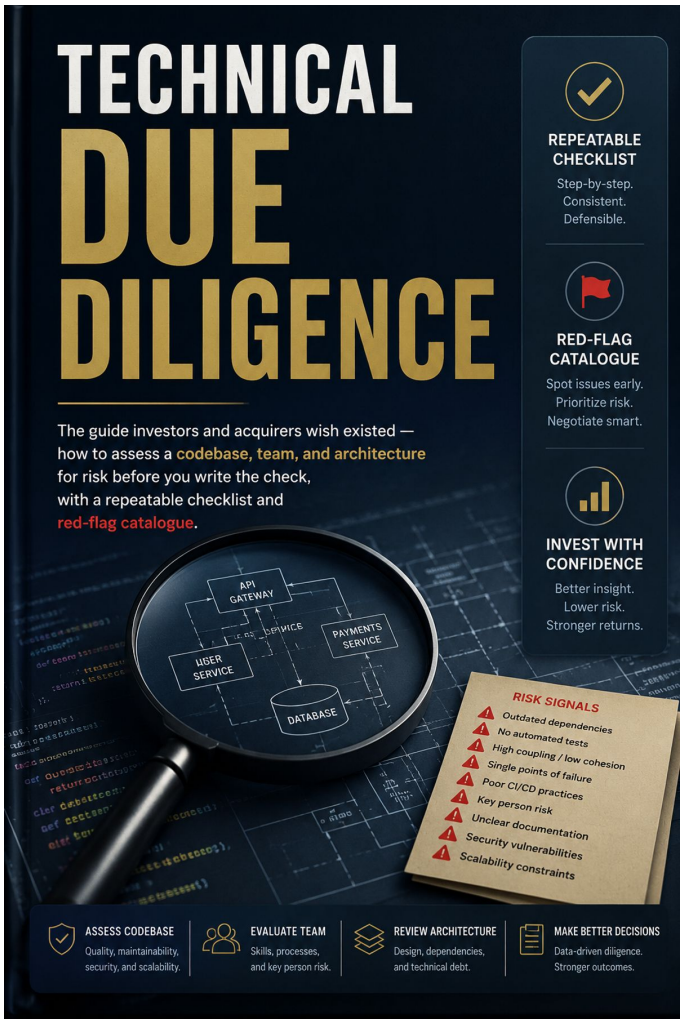

Technical Due Diligence

The guide investors and acquirers wish existed — how to assess a codebase, team, and architecture for risk before you write the check, with a repeatable checklist and red-flag catalogue.

Roger Stringer · rogerstringer.com

June 19, 2026

Contents

What Technical DD Actually Answers

When an investor or acquirer asks for "technical due diligence," they think they want a code review. What they actually need is an answer to one question: **how much risk am I buying, and what will it cost to fix?** Everything in a good technical DD serves that question. The codebase, the architecture, the team, the security posture — none of it matters in the abstract; it matters as risk priced into a deal.

This reframing changes how you do the work. A pure code review asks "is this good code?" — a question with no business meaning. Technical DD asks: *Will this technology support the business case the deal is built on? What's likely to break, and when? What's the gap between where it is and where it needs to be, in time and money? And is there a landmine — a single-person dependency, a security hole, a licensing problem — that could blow up the investment thesis after the check clears?* The deliverable isn't a list of nitpicks; it's a risk-graded picture a non-technical decision-maker can act on.

Who needs this, and when: an investor sizing a round, an acquirer evaluating a target, a board weighing a big technical bet, or a founder getting their own house assessed before raising. The common thread is real money riding on a technology the decider can't evaluate themselves — so they bring in someone who can, to turn opaque technical reality into priced risk.

This guide is the playbook for running that assessment — scoping it, reading the codebase, architecture, team, security, operations, dependencies, and process fast and fairly, cataloguing the red flags, and writing the report that lets a buyer decide. It's the mirror image of the First 90 Days (https://rogerstringer.com/guides/the-first-90-days) guide: that one is the new leader *building* the picture to lead from the inside; this one is the outside expert *assessing* the picture to inform a decision. Both rest on the same skill — reading a system and an organization quickly and honestly — and it's a strong, high-trust thing to be able to offer, the kind of engagement a fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) is uniquely placed to do.

The mindset to hold throughout: **you are not grading homework; you are pricing risk for someone about to spend real money.** Keep asking "so what does this mean for the deal?" and the assessment stays useful. Let's start with scoping it honestly.

Scoping the Engagement

Technical DD is almost always time-boxed and access-limited — you get a week, maybe two, and partial access to a system you've never seen — so scoping it honestly is the difference between a useful assessment and a misleading one. The cardinal sin is implying more certainty than the access allowed.

Time-box and prioritize by risk. You can't review everything, and you shouldn't try. Spend your limited time where the *deal risk* concentrates: the core systems the business depends on, the things most likely to be expensive surprises, the areas the thesis is most sensitive to. A week spent reading every utility function is wasted; a week spent on "can this scale to the growth the deal assumes, and where's the single-person risk" is the job. Prioritize against the *investment thesis*, not against some idea of completeness.

Get the access you need — and note what you didn't. Ideal access is read access to the code, time with the engineers, sight of the architecture, the incident history, the dependency list, and the cloud setup. Often you get less — a curated repo, one guarded conversation, no production access. That's workable, but you must **record the limits of your access in the report**, because "no major issues found" means something very different when you saw everything versus when you saw a demo and a slide deck. Honesty about what you *couldn't* assess is part of the integrity of the assessment.

Be clear about what DD can and can't tell you in the time. In a week you can assess structure, architecture, the obvious risks, the team's apparent capability, the security posture at a glance, and operational maturity — enough to price the big risks and spot the landmines. You *cannot* find every bug, guarantee there's no hidden security hole, or audit code quality line by line. Set that expectation with whoever hired you up front, so your report is read as "the material risks I could identify in the time," not "a guarantee the technology is sound." Overpromising here is how a diligence person ends up blamed for the thing they never had access to find.

Stay independent. You're often assessing a company that *wants the deal to happen* and will present its best face. Your value is precisely that you're not on that team — you report what you find to whoever hired you, plainly, including the inconvenient. The moment you start softening findings to keep the room comfortable, you've stopped being useful.

With the engagement honestly scoped and the access mapped, you start where the substance is: the codebase.

The Codebase Assessment

The codebase is the asset — in an acquisition it's often *the* asset — so reading it fast and fairly is the heart of the work. You're not reviewing it line by line (no time, no point); you're reading its *shape and signals* to judge quality, maintainability, and risk, much like the rapid read in the First 90 Days (https://rogerstringer.com/guides/the-first-90-days) guide but aimed at pricing rather than leading.

The fast, high-signal reads:

- **Get it running, if you can.** The onboarding experience is a measurement: a project that builds in ten minutes signals discipline; one that takes two days and tribal knowledge signals chaos and, often, key-person risk. (You may not get this access — note it if you don't.)
- **Trace one core flow end to end.** Follow the product's main action through the code. You learn the real architecture and the real quality from one honest path faster than from any diagram or any amount of skimming.
- **Read the structure and the signals.** Coherent organization or a sprawl? Tests that run and pass, or none? A consistent style, or five styles suggesting five eras and no standards? Where are the apologetic comments, the `TODO`s, the `// HACK — fix later` clusters? The signals tell the story the marketing won't.
- **Read the git history.** Underrated and revealing. Steady, reviewed commits across several people is a healthy team; everything authored by one person, or giant unreviewed dumps, is risk. The history shows how the software was *actually* built, not how they describe it.

What you're grading it against — always tied to the deal:

- **Maintainability** — can a new engineer be productive here, or does every change require the one person who understands it? The cost-of-ownership question, usually the biggest one.
- **Test coverage and confidence** — not a percentage for its own sake, but: can the team change this safely, or is every deploy a gamble? Low or no testing is a real, priceable risk to future velocity.
- **Tech debt, honestly weighed.** All real codebases have debt; the question is whether it's *manageable debt* (normal, livable) or *load-bearing debt* (the thing that'll need an expensive rewrite to support the growth the deal assumes). Distinguishing the two is most of the judgment.
- **The AI-era wrinkle.** Increasingly, a chunk of the code was AI-generated, which the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide notes runs measurably higher in defects and odd patterns. Look for the tells — inconsistent idioms, code nobody on the team can fully explain, dependencies that don't quite belong — because "generated fast and never really owned" is a risk that's newly common and easy to miss.

The output of this stage isn't a verdict like "good" or "bad" — it's a *characterization*: where the code is solid, where it's fragile, and what the fragile parts would cost to shore up. That feeds straight into the bigger structural question: will the architecture hold?

Architecture & Scalability

Code quality is about the present; architecture is about the future — specifically, whether the system can support where the *deal* expects the business to go. An acquirer paying for 10x growth needs to know the architecture can survive 10x, or what it'll cost to get there. That's the question this stage answers.

Understand the shape first. Get the real architecture — not the idealized diagram, the actual one. What are the major components, how do they communicate, where does the data live, what does it depend on? Often the most useful finding is the gap between the architecture they *describe* and the one that actually exists. Draw it yourself from tracing the system; if you can't, that itself is a finding about how well-understood the system is internally.

Assess against the growth the thesis assumes. This is the crux, and it must tie to the business case:

- **Where are the scaling limits?** Every architecture has a ceiling — the single database that can't be sharded easily, the synchronous design that falls over under load, the component already at capacity. Find the next ceiling the business will hit, and judge whether it's before or after the milestones the deal is priced on. "It'll need work to reach the growth you're paying for, and here's roughly how much" is exactly the priced risk the buyer needs.
- **Is the scaling story credible or hand-wavy?** When you ask "how does this handle 10x?", do you get a concrete answer grounded in how the system actually works, or reassuring vagueness? The quality of the *answer* is as diagnostic as the architecture.
- **Single points of failure.** The one database with no replica, the one service everything routes through, the one external dependency the whole business rests on. Each is a risk to price.
- **Right-sized, or fashion-driven?** Over-engineering is a risk too — a tiny company running a sprawling microservices-and-Kubernetes setup it doesn't need is carrying complexity (and the team to maintain it) it can't afford, the opposite failure from the Boring Stack (https://rogerstringer.com/guides/the-boring-stack) guide's wisdom. Architecture mismatched to the company's stage — in either direction — is a finding.

Cost and lock-in. What does the infrastructure cost to run, and does it scale linearly or worse with growth (a margin problem hiding in the architecture)? And how locked-in is it — deeply wedded to one cloud's proprietary services in a way that constrains the buyer's options? These are business risks wearing technical clothes, which is exactly what the buyer hired you to translate.

The deliverable here is a clear-eyed read of whether the architecture is an *enabler* or a *constraint* for the deal's growth case — and if a constraint, the rough size of the work to fix it. But architecture is built and maintained by people, and the people are often the bigger risk.

The Team & Key-Person Risk

In a technology deal, you're not just buying code — you're buying (or betting on) the *team that can keep building it*, and the people are often where the real risk lives. A pristine codebase nobody left at the company understands is worth far less than messy code with the team that knows it intact. So reading the team is central, not a soft add-on.

Key-person risk is the headline. The single most important question: **who, if they left tomorrow, would take irreplaceable knowledge — or the ability to maintain the system — with them?** Nearly every company being acquired has at least one, usually the founder-engineer or the early hire who built the core. This isn't a criticism of them; it's a concentration risk that can sink a deal's value, because if the acquisition causes them to leave (and acquisitions often do), the buyer may find they purchased a system no remaining person can fully operate. Quantifying this — how many critical systems rest on how few heads — is some of the most valuable work you do.

Read capability and depth, fairly and fast. You're assessing whether the team can deliver what the deal assumes:

- **Depth vs. a house of cards.** Is knowledge spread across the team or concentrated? Could they survive losing any one person? A team where everyone understands their area and there's overlap is robust; one where each system has exactly one owner is fragile.
- **Capability for the road ahead.** Can this team execute the post-deal plan — the scaling, the new features, the integration? Their past (the git history, what they've shipped) is your best evidence, more than résumés.
- **The bus-factor across the org**, not just engineering. Who knows the deployment process? Who has the production credentials? Who understands the gnarly business logic? Risk concentrated in one head anywhere is a finding.

Retention and the deal's effect on it. Acquisitions change incentives — vesting cliffs, culture clash, founders who lose autonomy. Part of the assessment is reading *retention risk*: who's likely to leave after the deal, and how load-bearing are they? A retention package for the two people who hold the system together is often the cheapest risk mitigation in the whole transaction, and flagging that need is a high-value finding.

Be fair and evidence-based. You're judging people fast, on partial information, with their livelihoods affected — the same caution the First 90 Days (https://rogerstringer.com/guides/the-first-90-days) guide urges. Base findings on evidence (what the history and the systems show), not vibes from one tense meeting, and frame them as *risk to the deal*, not verdicts on individuals. The output is a clear map of where the human risk concentrates and what it would take to de-risk — which a buyer can act on with retention, hiring, or a revised price. Next, the risk that can turn a good deal toxic overnight: security.

Security & Compliance

Security is the area where a single finding can change a deal — a serious breach or a glaring hole isn't a line item, it's potentially a deal-breaker, a regulatory liability, or a renegotiation. You're not running a full penetration test in a week (and should say so), but a posture assessment catches the issues that matter most for the transaction.

The posture check — what you can assess quickly:

- **Secrets and credentials handling.** Are secrets in the code or in git history (a shockingly common and serious finding)? Is there a real secrets-management approach, or passwords in a config file? Fast to check and highly diagnostic of overall security maturity — the discipline the Auth (https://rogerstringer.com/guides/auth-done-right) and Boring Stack (https://rogerstringer.com/guides/the-boring-stack) guides insist on. Secrets in the repo is a red flag that predicts others.
- **Authentication and authorization.** Is auth sound, or rolled badly? Critically, is *authorization* enforced — can users access only their own data, or is there the insecure-direct-object-reference hole (any logged-in user reaching anyone's records) the Auth guide warns about? Authorization bugs are common, severe, and exactly the kind of latent liability a buyer needs priced.
- **Data handling and exposure.** What sensitive data does the company hold (personal data, payment info, health data), how is it protected at rest and in transit, and who can access it? A company sitting on a pile of personal data with weak protection is carrying a breach-shaped liability.
- **Known vulnerabilities.** Out-of-date dependencies with known CVEs, unpatched systems, the obvious stuff a scanner finds. Not exhaustive, but a company that's never run a dependency scan is telling you something about its security culture.

Compliance, sized to the business. If the company operates somewhere regulated — health (HIPAA), payments (PCI), EU personal data (GDPR), enterprise contracts (SOC 2) — are they actually compliant, or claiming to be? Compliance gaps are real, priceable liabilities: the cost to *become* compliant, plus the risk of operating without it. A SaaS selling to enterprises with no SOC 2 has a known, quantifiable gap that affects the deal.

The history. Ask about past incidents and breaches. How they answer is as revealing as the answer: a team that discusses a past incident openly with a clear account of what they fixed is mature; evasiveness or "we've never had any issues" (rarely true) is its own signal.

Frame findings by severity and deal impact. Not every security issue is a deal-breaker; your job is to separate the *must-fix-before-close* (a live breach, exposed personal data, a critical auth hole) from the *fix-soon* (out-of-date deps, missing scans) from the *normal hygiene*. The buyer needs that gradation to know what's a renegotiation and what's a Tuesday. Security findings, more than any others, should be ranked ruthlessly by how much they actually threaten the deal. Which leads naturally to whether the company can keep the thing running at all: operations.

Operations & Reliability

A company can have good code and good architecture and still be a mess to *operate* — and operational immaturity is a real risk to a deal, because it predicts outages, slow delivery, and a team that fights fires instead of building. This stage assesses whether they can actually keep the lights on and ship, which matters enormously for whether the post-deal plan is achievable.

Deployment maturity — how does code get to production?

- **Is deploying routine or terrifying?** A team that deploys safely many times a day is operationally mature; a team that deploys manually, rarely, and with held breath is carrying risk (and will struggle to deliver the roadmap the deal assumes). The Boring Stack (https://rogerstringer.com/guides/the-boring-stack) and 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guides treat safe, frequent, reversible deploys as table stakes; their absence is a finding.
- **Can they roll back?** When a bad deploy goes out, can they recover in minutes, or is it an all-hands scramble? Recovery capability is a direct read on operational risk.

Monitoring and incident response.

- **Do they know when something's broken** before customers tell them? A company with no monitoring is flying blind, and "we find out about outages from angry tweets" is a maturity finding.
- **How do they handle incidents?** A process, or chaos? Do they learn from outages (postmortems, fixes that stick) or repeat them? The incident history — if you can see it — is gold: frequency, severity, and whether the same things keep breaking.

The reliability reality — and the backups question. What's the actual uptime, and — the one I'd never skip — **do they have backups, off-site, that they've actually tested restoring?** The Boring Stack (https://rogerstringer.com/guides/the-boring-stack) and Postgres (https://rogerstringer.com/guides/postgres-for-app-developers) guides hammer this because it's the difference between a bad day and a company-ending one. A target with no tested backups is one disaster away from losing the very asset being purchased; that's a finding that belongs near the top of any report.

The bus factor, operationally. Tied to the team chapter: who can actually *run* this system? If only one person knows how to deploy, restore a backup, or fix the production database, the operational risk and the key-person risk are the same risk, doubled.

The output is a read on operational maturity — routine and safe, or fragile and heroic — because it predicts both the reliability the buyer inherits and the team's capacity to execute what comes next. Now, a category of risk that's quietly grown more dangerous: what the software depends on.

Dependencies & Licensing

Modern software is mostly other people's software — a thin layer of your code on a deep stack of dependencies — and that stack carries risks that are easy to miss in a quick look but can be genuinely deal-affecting. Two kinds: supply-chain risk and licensing risk, and both have gotten sharper in the AI era.

The dependency supply chain.

- **How many, how current, how maintained?** A reasonable, current dependency set is healthy. A pile of hundreds of packages, many years out of date, some abandoned (no commits in years, a single unpaid maintainer), is risk — unpatched vulnerabilities and components that could break or go unmaintained. Run a dependency audit; the count of known vulnerabilities and abandoned packages is a quick, hard number for the report.
- **The AI-introduced-dependency problem.** Newly common and worth specific attention: the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide notes AI coding agents reaching for packages that don't quite exist, or pulling in obscure ones nobody actually chose or vets. A codebase built fast with heavy AI assistance may carry dependencies no human consciously decided on — a supply-chain liability that crept in. Look for packages that seem out of place, oddly obscure, or that nobody on the team can explain. "We're not sure why that's in there" is a finding.
- **Concentration on a single vendor or API.** If the whole business rests on one external service (a payments provider, a data source, one cloud's proprietary feature), that's a dependency risk too — what happens to the business if that vendor changes terms, prices, or disappears?

Licensing — the quiet landmine. This one can genuinely blow up a deal, and it's invisible until someone looks:

- **Are the open-source licenses compatible with the business?** Most permissive licenses (MIT, Apache) are fine. But copyleft licenses (GPL and kin) can, in some uses, legally require you to open-source *your own* code — a catastrophic surprise for a company selling proprietary software. A GPL dependency buried in a commercial product is exactly the kind of finding that stops a deal until it's resolved.
- **Do they have the rights to what they ship?** Code copied from elsewhere, an unlicensed library, AI-generated code of murky provenance — the question "do they actually own or license everything in this product they're selling" is one a buyer's lawyers will eventually ask, and better you surface it now than they discover it later.
- **Whose IP is it, really?** Were contractors used without proper IP assignment? Is there code the founders wrote at a previous employer? Ownership of the asset is the most basic thing being purchased; gaps here are serious.

Dependencies and licensing are unglamorous and easy to skip, which is exactly why they're where nasty surprises hide. A clean read here is reassuring; a GPL violation or a tree full of unowned, AI-summoned packages is the kind of finding that earns your whole fee. With the technical pieces assessed, one more lens reveals how the team actually works: process and velocity.

Process & Velocity

The final lens is about *how the team works* — because a deal isn't just buying what exists today, it's betting on the team's ability to keep delivering, and their process and velocity are the best evidence of whether they can. This is where the git history and the ticket tracker quietly tell you the truth the leadership presentation won't.

What the git history reveals. You looked at it for code quality; now read it for *how the team operates*:

- **Velocity and consistency.** Steady, sustained delivery, or feast-and-famine, or a recent slowdown? A team whose output has quietly stalled — maybe people are leaving, maybe morale cracked, maybe the codebase got too painful to change — is a risk the slide deck won't mention but the commit graph will.
- **How work actually flows.** Are changes reviewed (pull requests, approvals) or pushed straight to main? Is there evidence of testing and CI, or do things just ship? The *process* visible in the history tells you whether quality is systematic or accidental.
- **Concentration, again.** If one person authors most commits, that's the key-person risk showing up in yet another place — these signals reinforce each other across the whole assessment, which is how you build confidence in a finding.

What the tickets and tracker reveal. If you can see their project tracker: Is there a runaway backlog of bugs that never get fixed (a quality and morale signal)? Is work planned and tracked, or chaotic? Does the bug-to-feature ratio suggest they're drowning in defects? The tracker is the team's own honest record of what's hard.

Velocity in the AI era — read it carefully. A newly relevant subtlety: a team shipping a *lot* of code fast might be highly productive, or might be generating code faster than they can verify and own it — the exact trap the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide describes, where cheap generation outruns the verification and ownership that make it safe. High raw output is not automatically a good sign; pair it with the quality and defect reads to judge whether the velocity is *healthy* or *debt-accumulating*. "They ship fast" needs the follow-up "— and can they maintain what they ship?"

The maturity read. Pulling it together: does this team have the *discipline* to deliver the post-deal roadmap — the planning, review, testing, and steady execution — or do they run on heroics that won't survive growth or the disruption of a deal? Process maturity predicts future delivery better than any single snapshot of the code.

The output is a read on whether the team can sustain and grow what they've built — the forward-looking complement to all the point-in-time technical findings. With every lens assessed, you assemble the most important section: the risks that actually matter. The red flags.

Red Flags & Deal-Breakers

You've assessed every angle; now comes the part the buyer most needs — the honest catalogue of what's *wrong*, ranked by how much it should worry them. This is the heart of the report, and the discipline is ruthless prioritization: separating the things that should genuinely stop or reprice a deal from the things that are normal and fixable.

The deal-breakers — findings that should stop or fundamentally reprice the transaction:

- **A live, unaddressed security breach or critical exposure** — exposed personal data, a credential leak, a critical auth hole letting users reach others' data. Until resolved, this is a no.
- **No backups, or untested ones** — the company is one disaster from losing the asset being bought.
- **A licensing or IP problem** — a GPL dependency in proprietary code, code the company doesn't actually own, missing contractor IP assignment. The thing being purchased may not be theirs to sell.
- **Catastrophic key-person risk** — the entire system understood by one person who's likely to leave post-deal, with no mitigation. The buyer may be acquiring a system no one will be able to run.
- **An architecture that fundamentally can't support the thesis**, where the cost to fix approaches the cost to rebuild — the deal's growth case is built on something that won't bear it.

The serious concerns — not deal-breakers, but real risk to price in or plan around:

- Significant tech debt that will slow the roadmap; weak or no testing; out-of-date, vulnerable dependencies; operational immaturity (scary deploys, no monitoring); concerning velocity or morale signals; a compliance gap (no SOC 2 where the market needs it). Each is a number — time and money — the buyer factors into the price or the post-deal plan.

The normal stuff — explicitly called out as fine, because saying so builds trust: every real codebase has debt, every team has gaps, no system is perfect. A report that lists only problems and never says "this is normal and healthy" reads as alarmist and loses credibility. Naming what's *fine* is what makes your alarms believable when you raise them.

The principle that makes this section useful: rank everything by deal impact, not by how it offends your engineering taste. A messy code style that bothers you but doesn't threaten the business is a footnote; a quiet licensing violation that does is the headline. The buyer is making a financial decision — your job is to tell them which findings actually move that decision and which are noise. Get that ranking right and the report does its job; get it wrong (burying the deal-breaker under nitpicks, or crying wolf over style) and you've failed the person who hired you. Now, how to write it all down so they can act on it.

Writing the Report

All the assessment in the world is worthless if the report doesn't let the decision-maker act — and the reader is usually *non-technical*, making a financial decision, with limited time. Writing the report well is as much the job as the analysis, and it's the same outcomes-not-architecture discipline the Fractional CTO Field Guide (https://rogerstringer.com/guides/the-fractional-cto-field-guide) and First 90 Days (https://rogerstringer.com/guides/the-first-90-days) guides preach, applied to a buyer instead of a board.

Lead with the answer. The report opens with the bottom line: the overall risk level, the deal-breakers (if any), and the headline risks — in plain language, on the first page. A partner skimming for two minutes should come away knowing whether this is a green light, a green light with conditions, or a problem. Bury the conclusion under thirty pages of technical detail and you've failed the reader who most needs it.

Risk-grade everything. Every finding gets a severity and, ideally, a rough cost or effort to fix. "Critical / High / Medium / Low," or red/amber/green — some consistent scale that lets a non-technical reader sort the landmines from the lint. A finding without a severity is just an observation; a finding *with* one is decision-support. And where you can, attach a number: "roughly N months of engineering to make this scale" turns a vague worry into something the buyer can price.

Translate every technical thing into business impact. Not "the test coverage is 12%" but "the team can't change this system with confidence, which means slower, riskier delivery of the roadmap and a higher chance of customer-facing breakage." Not "there's a GPL dependency" but "a piece of the product may legally require open-sourcing your proprietary code — this needs legal review before close." The reader can't translate technical findings themselves; that translation *is* your value.

Separate fact from assessment, and state confidence. Distinguish what you *observed* (fact) from what you *conclude* (judgment), so the reader can trust your facts even where they weigh your conclusions differently. And be explicit about confidence and the limits of access from the scoping chapter — "I could not assess X because I had no access" is essential honesty, not a weakness. A report that quietly implies certainty it didn't earn is the one that gets the diligence person blamed later.

Be constructive where you can. Most findings come with a path: "this is fixable in roughly this much time and money," "a retention package for these two people mitigates the key-person risk," "this needs legal review." A report that's all problems and no paths is less useful than one that helps the buyer see *what it would take* — because most deals proceed *with conditions*, and your report is what shapes those conditions.

The structure that works: an executive summary with the verdict and headline risks; a risk-graded findings table; then the detail by area (code, architecture, team, security, ops, dependencies, process) for those who want it; and an explicit statement of scope and access limits. Written that way, the report turns a week of technical assessment into something a non-technical buyer can actually decide on — which was the whole point. To make all of this repeatable, the last chapter is the checklist.

The Checklist

Here's the whole assessment as a checklist you can run every time — the repeatable backbone that makes technical DD a craft you get faster and better at rather than reinventing each engagement. Adapt it to the deal; the structure holds.

Scope & access

```
- [ ] Time-box agreed; priorities set against the investment thesis
- [ ] Access secured: code, engineers, architecture, incidents, infra, deps
- [ ] Access LIMITS recorded (what you could not assess)
- [ ] Expectations set: "material risks in the time," not a guarantee
```

Codebase

```
- [ ] Got it running / noted if not
- [ ] Traced one core flow end to end
- [ ] Structure, tests, style, TODO/HACK clusters reviewed
- [ ] Git history read (cadence, reviews, author concentration)
- [ ] Tech debt characterized: manageable vs. load-bearing
- [ ] AI-generated-code tells checked (unexplained code, odd idioms)
```

Architecture & scalability

```
- [ ] Real architecture mapped (vs. the described one)
- [ ] Next scaling ceiling vs. the deal's growth milestones
- [ ] Single points of failure identified
- [ ] Right-sized for stage? (over- or under-engineered)
- [ ] Infra cost and vendor lock-in assessed
```

Team & key-person risk

```
- [ ] Key-person / bus-factor mapped across critical systems
- [ ] Knowledge depth vs. concentration
- [ ] Capability for the post-deal roadmap
- [ ] Retention risk from the deal itself; mitigations noted
```

Security & compliance

```
- [ ] Secrets in code / git history?
- [ ] Auth AND authorization (can users reach others' data?)
- [ ] Sensitive data handling and exposure
- [ ] Known-vuln dependency scan
- [ ] Compliance gaps (SOC2 / GDPR / HIPAA / PCI) vs. market need
- [ ] Incident / breach history
```

Operations & reliability

```
- [ ] Deploy maturity (routine & safe vs. manual & scary)
- [ ] Rollback capability
- [ ] Monitoring / do they know when it breaks?
- [ ] BACKUPS: off-site and tested-restore?
- [ ] Operational bus factor (who can run it?)
```

Dependencies & licensing

```
- [ ] Dependency count, currency, abandoned packages
- [ ] AI-introduced / unexplained dependencies
- [ ] License compatibility (GPL in proprietary code?)
- [ ] IP ownership (contractors, prior-employer code, AI provenance)
```

Process & velocity

```
- [ ] Delivery cadence and any slowdown
- [ ] Review / testing / CI in practice
- [ ] Backlog / bug-ratio health
- [ ] Velocity healthy vs. debt-accumulating (the 70/30 read)
```

Report

```
- [ ] Executive summary: verdict + headline risks, page one
- [ ] Every finding risk-graded, with rough cost where possible
- [ ] Every technical finding translated to business impact
- [ ] Fact vs. assessment separated; confidence and access limits stated
- [ ] Constructive paths / conditions for proceeding
```

Run that, end to end, and you've turned an opaque technology into priced, ranked, actionable risk — which is the entire job. Technical due diligence is one of the highest-trust, highest-value things a senior technical person can offer, precisely because so few people can read a system *and* a business at once and tell a buyer the honest truth. It's the assessment mirror of the First 90 Days (https://rogerstringer.com/guides/the-first-90-days) playbook and a natural extension of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) work. Run the checklist, price the risk, write it plainly — and help someone make a very expensive decision with their eyes open.

About Roger

I'm Roger Stringer — I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email** — roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X** — @freekrai (https://x.com/freekrai)
- **GitHub** — github.com/freekrai (https://github.com/freekrai)
- **LinkedIn** — linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting — follow along wherever suits you.