

GUIDE · FREE TO READ

What technical due diligence actually looks for

Your term sheet is signed and an investor has hired a former CTO to read your codebase. You have spent three years building something that works and about four hours thinking about how it would look to a stranger with a checklist. This is what that stranger is actually checking, in the order it tends to matter.

No email required. Nothing gated. If you take all of it and never speak to us, the guide has done its job.

Contents

- 1 What actually happens over four to six weeks
- 2 Ugly code is survivable. Unmanaged risk is not.
- 3 The five findings that actually sink deals
- 4 A delivery record answers the question they are really asking
- 5 What four weeks can fix, and what it cannot
- 6 What to do on Monday

Where this comes from, and where it does not

We have not raised a round and have not sat on the wrong side of a funding diligence ourselves. This is written from the delivery side: from the requests auditors make of engineering teams, and from what a delivery record can and cannot answer when one arrives. Treat it as a description of the questions, not as a war story about surviving

them. If you want the war story, ask a founder two rounds ahead of you — they will give you a better one than we can.

1 What actually happens over four to six weeks

Technical due diligence is not a code review. It is a risk assessment that happens to involve code, run by someone whose job is to tell an investment committee what could go wrong after the money lands. At Series A it typically runs four to six weeks — the sourced range is in the box below — and most of that time is not spent reading your source.

Five things usually happen, roughly in this order.

- **An architecture walkthrough.** Ninety minutes with whoever can explain the system. They are listening for whether your explanation matches the diagram, and whether the diagram matches what a scan will show them later. Two of those three disagreeing is common and forgivable. All three disagreeing is a finding.
- **An automated scan of the codebase.** Size, language mix, duplication, complexity hot spots, test coverage, commit history. This takes them an afternoon and gives them the questions for everything that follows. It also tells them who wrote what, which matters more than most founders expect.
- **A dependency and licence audit.** Every third-party package, transitively, with its licence and its known vulnerabilities. This is the single most mechanical part of the process and the one that most reliably produces a problem.
- **A security questionnaire.** Access control, secret handling, backups, incident history, what customer data you hold and where. Usually a spreadsheet. Frequently the same spreadsheet your enterprise customers sent you, which is worth noticing.
- **Interviews with the people who built it.** Not just the CTO. They will ask to speak to two or three engineers separately, and they are comparing the answers. A team that describes the same system three different ways is telling them something.

The output is a report to the investor, not to you. You will usually see a summary and a list of remediation items. What you will not see is the sentence that actually moves the valuation, which is some version of “this team either does or does not know what it has built”.

The two numbers worth carrying into the process

- **68%** of audited codebases contain open-source licence conflicts — [Technical due diligence benchmarks, 2026](https://hypernestlabs.com/guides/technical-due-diligence-checklist) (<https://hypernestlabs.com/guides/technical-due-diligence-checklist>)
- **4–6 weeks** typical duration of Series A technical due diligence — [Technical due diligence guidance, 2026](https://verycreatives.com/blog/technical-due-diligence-for-founders) (<https://verycreatives.com/blog/technical-due-diligence-for-founders>)

Published industry figures, directional rather than audited. Your own process may be shorter or considerably longer; the range is what to plan against, not what to expect precisely.

2 Ugly code is survivable. Unmanaged risk is not.

Founders prepare for diligence by tidying. They rename variables, delete dead files, and lose a fortnight making the codebase look like something they would be proud to show a stranger. It is the wrong fortnight, spent on the wrong axis.

“Auditors rarely fail a deal on ugly code. They fail it on unmanaged risk.”

The difference is easier to see with one concrete module. Imagine a four-thousand-line billing service written in a hurry two years ago. The logic is duplicated in three places, half the functions take seven arguments, and the person who wrote it named things after the tickets rather than the concepts.

An auditor looks at that and writes down a number: this will cost roughly this much engineering time to make maintainable. It is a line item. It is negotiable. It has never on its own killed a round.

Then they ask the real question: *how do you know that last month's change to this service did not break refunds?* If the answer is a test suite that covers the refund path, a review record showing a second person read the change, and a deployment you could roll back within the hour, the ugly code is a cost. If the answer is “we tested it locally and Dave usually catches things”, the ugly code has become an unknown — and an unknown gets priced at its worst plausible value, because the auditor has no basis for pricing it at anything else.

Tidy code with no evidence around it scores worse than untidy code with evidence. That inversion is the whole reframe.

This matters more each year, not less. A codebase where a significant share of the lines were generated by a model is not inherently worse — but nobody can claim to have read all of it, and “a person reviewed every line” stops being a credible answer. What replaces it has to be something a machine did, repeatably, with a record. We have written about that shift [in the playbook](#).

3 The five findings that actually sink deals

In rough order of how often they turn up, against how much damage they do when they do.

Licence conflicts, because they are a legal problem wearing an engineering costume

68% of audited codebases contain open-source licence conflicts — [Technical due diligence benchmarks, 2026](https://hypernestlabs.com/guides/technical-due-diligence-checklist) (<https://hypernestlabs.com/guides/technical-due-diligence-checklist>) (directional, not audited). That figure is high because the mechanism is invisible: you install a package, that package depends on four others, and one of those carries a copyleft licence that has an opinion about distributing your own source code. Nobody chose it. Nobody read it. It is three levels down a dependency tree and it took eleven seconds to arrive.

This sinks deals because it is not an engineering judgement call that can be argued about — it goes to the acquirer’s counsel, who will ask whether the exclusive rights being purchased are actually exclusive. AI coding assistants have made this sharper: dependencies now enter a codebase suggested rather than chosen, and a suggestion carries no licence review with it.

Key-person dependency, because your commit history proves it whether you disclose it or not

One person understands the payments path. One person can explain why the scheduler works the way it does. Every founder knows this about their own team, and most assume it

is not visible from the outside. It is the first thing the scan reports: for each important directory, who has changed it, and how recently anyone else did.

What makes it fatal is not the dependency itself — every small company has one — but the combination of the dependency and a denial. Say it first, name the person, and show what you are doing about it. An auditor who discovers it themselves after you said the knowledge was well spread has learned something about you that no amount of good code will offset.

No record of why anything was decided, because it makes choices look like accidents

They will ask why you built your own job queue instead of using a managed one. There is often a genuinely good answer: the managed option did not support a thing you needed, or the cost model did not work at your volume, or you tried it and hit a limit. If that reasoning was written down at the time, with a date, it reads as engineering judgement.

If it was not, the honest reconstruction — “I think it was faster at the time” — reads as drift. An auditor cannot tell a deliberate trade-off from an unconsidered default by looking at the code, because both produce the same code. The only difference is whether a record exists.

Scaling assumptions with no boundary, because the round is being raised on growth

You are raising money on a plan to grow usage by an order of magnitude. The audit exists partly to ask whether the system can absorb that, and the failure mode is not “we have not load tested”. Plenty of companies pass without a load test.

The failure mode is not knowing which component gives way first. “The primary database write path is the constraint; we estimate it holds to roughly ten times current volume and then needs partitioning, which is about a quarter of work” is a good answer even if the estimate is rough, because it is bounded and it is falsifiable. “It should be fine, it is all on managed infrastructure” is an unbounded claim, and an unbounded claim about the thing the investment thesis depends on is exactly the shape of unmanaged risk.

A security posture you cannot evidence, because assertion is not evidence

Nobody expects a Series A company to hold a formal security certification, and not having one rarely blocks a round on its own. What is expected is that you can show, rather than state, the controls you claim to have. Which checks run on every change. Who has production access and how that list is reviewed. Where secrets live. Whether you have had an incident, and what you did.

“We take security seriously” is the weakest sentence in the English language when addressed to someone whose job is verification. If your enterprise customers already put you through a vendor security review, you have most of this written down already — that work is reusable here, and [we have written about it separately](#).

4 A delivery record answers the question they are really asking

Every finding above is a variation on one question: is risk managed here, or is it merely absent so far? Documents can answer parts of it. An architecture diagram answers what the system is. A security questionnaire answers what your controls are meant to be. Neither answers whether the controls actually ran.

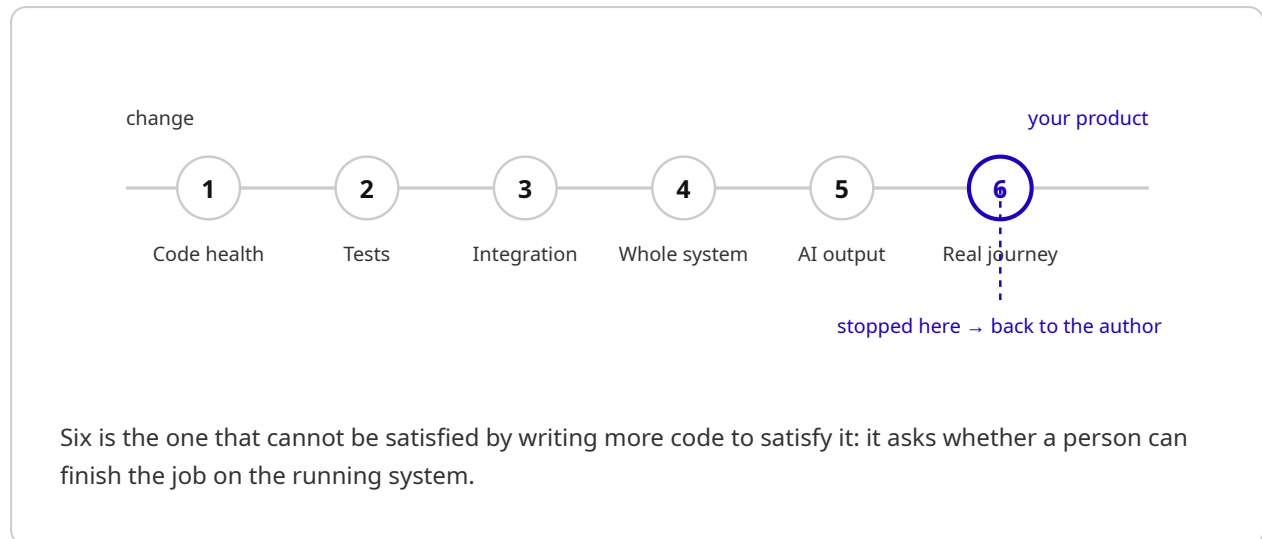
A delivery record does, and it is the one artifact that answers all five findings at once. By delivery record we mean something unglamorous: for every change that reached production, what changed, what checked it and what those checks returned, who reviewed it, and which decision it came from.

Read that against the list. Key-person dependency: the record shows who has touched what, and if a second reviewer signed off on the payments path, that is direct evidence the knowledge is not held by one head. Decision history: the reasoning is attached to the change rather than remembered. Security: rather than asserting that scans run, you show the last hundred times they ran and the times they failed and stopped a release. Licences: a dependency check that runs on every change means the answer to “when did you last audit this” is a date this week rather than a shrug.

The failures are the valuable part. A record that only ever shows green is not evidence of quality — it is evidence that nothing is being checked hard enough to fail. An auditor who

sees a check that blocked a release last Tuesday learns more about your process than one who sees an unbroken run of passes.

For what it is worth, this is the shape of what we produce for the teams we build with: 6 checks, each with a recorded result and a stated point at which it runs — two on every change, two on integration, two before release.



We publish our own numbers on the same basis, including the ones that read badly — a large share of our work was later redone, and queueing time rose as we ran more in parallel. Both are on [the results page](#) with their caveats, as of 2 August 2026. We show them because a record that can only report good news tells you nothing on the day something goes wrong, which is the only day you urgently needed it. The same logic is why an auditor trusts a record with failures in it.

5 What four weeks can fix, and what it cannot

Assume the term sheet is signed and diligence starts in a month. Be honest with yourself about which side of this line each problem sits on, because spending the month on the wrong side is worse than spending it on nothing.

Not fixable in four weeks

- A genuine key-person dependency. Transferring two years of context takes months, and pairing sessions

Fixable in four weeks

- The licence position. Scanning the dependency tree takes an afternoon, and most conflicts are

booked last week do not change the commit history.

- A missing test suite. Coverage added under deadline pressure tends to test what is easy rather than what is risky, and an auditor can see the commit dates.
- Architectural choices that are now load-bearing. If the coupling is real, disclose it and price it rather than starting a rewrite you will not finish.
- A history of decisions nobody wrote down. You can record what you remember; you cannot manufacture a two-year trail, and attempting it is worse than the gap.

resolved by swapping one package or buying a commercial licence.

- Known vulnerabilities in dependencies. Largely a patching exercise, and leaving them unpatched reads as inattention rather than as risk appetite.
- Secrets in the repository history. Rotate them, and say that you rotated them.
- An access review. Who can reach production, who left and still can, and one page recording that you checked.
- A dozen decision records for the choices you already made, each dated honestly as written now about a decision taken then.
- The walkthrough itself. Rehearse it with someone outside the company until the explanation and the diagram agree.

One warning on the right-hand column. Do not produce a documentation set that was all created in the same week and present it as standing practice. Auditors read timestamps, and a file tree with a uniform creation date is a finding of its own — it converts a documentation gap, which is ordinary, into a credibility question, which is not. Date things truthfully and say what prompted them. “We wrote these down while preparing for diligence, and we have kept doing it since” is a fine sentence, and it has the advantage of being checkable in three months.

6 What to do on Monday

Cheapest first, and cheap here means an afternoon rather than a quarter. The first three cost almost nothing and cover the finding that turns up most often.

1. **Run a licence scan on your full dependency tree.** Transitively, not just your direct dependencies. Free tooling does this. On the sourced figure in section three, assume

yours turns something up.

2. **Print the authorship of your ten most important directories.** One command against your version control history. Wherever one name owns a directory outright, you have found your key-person risk before the auditor does.
3. **Write down the single thing that breaks first under ten times the load,** with your reasoning and today's date. A rough bounded estimate beats a confident unbounded one.
4. **Record ten decisions you already made.** A paragraph each: what you chose, what you rejected, why, and when. Start with the three an outsider would most likely question.
5. **Make your checks visible in one place.** What runs on every change, what it blocks, and when it last failed. If nothing has failed in six months, that is worth investigating rather than celebrating.
6. **Have someone outside run the architecture walkthrough with you.** A friendly technical founder, an hour, permission to be rude. They will find the two questions you cannot answer, which is exactly two more than you want the auditor finding.

None of this requires anyone's tooling, ours included, and a team that does the six steps above will handle diligence better than a team that hires help and skips them. The reason to build this habit before you need it is that the record has to have been accumulating to be worth anything — assembled in the month before an audit, it is a document; accumulated over a year, it is evidence.