contact@namiq.io · namiq.io

NamiQ

NAMIQ GUIDE

What Technical Due Diligence Looks For

Five findings that sink deals, what to do about each, and what four weeks can and cannot fix

For a founder or CTO about to be read by a stranger with a checklist. What happens, why ugly code is survivable, the five findings, and the sheets to fill in this week.

PREPARED BY	Hung Le, NamiQ	DATE	31 October 2026
FACTS AS OF	31 October 2026	EDITION	1.0
FOR	Founders and CTOs	INSIDE	5 findings, 7 appendices, 4 figures

NamiQ © 2026 NamiQ. All rights reserved. This guide may be shared unaltered, with attribution. Free to read and share.
Web edition: namiq.io/resources/technical-due-diligence

START HERE

Diligence on a page

Technical due diligence is a risk assessment that happens to involve code. Five findings account for most of what costs a round, and none of them is about how the code looks. Each has a first move this week and a proof you can show a stranger.

Five findings, each with the first move and a proof you can show a stranger.

#	Finding	Why it sinks a deal	First move	Proof	§
1	**Licence conflicts**	A legal problem in engineering clothes: it goes to the acquirer's counsel	Run a licence scan over the full tree, transitively	A dated inventory with every licence and the tool's output	3.1
2	**Key-person dependency**	Your commit history shows it whether you say so or not	Print the authorship of your ten most important directories	The map, with the person named and a plan beside it	3.2
3	**No record of why**	Without a record, choices look like accidents	Record ten decisions you already made, dated honestly	Ten dated decision records	3.3
4	**Scaling with no boundary**	The round is raised on growth, and an unbounded claim is unmanaged risk	Write the one thing that breaks first at ten times the load	A one-page bounded estimate with a date	3.4
5	**A security posture you cannot evidence**	Assertion is not evidence	List what runs on every change and who can reach production	A check log with a last-failed date, and an access list with a review date	3.5

Where to start, for what you have today

A term sheet is signed and diligence starts in a month	Page 2, then Chapter 5 for what four weeks can fix, then the order in 5.2
You have ten minutes	Page 2, then Chapter 3.6: score the five
You are about to give the walkthrough	Appendix D, and Chapter 1.4 to check your three descriptions agree
You have been sent the security questionnaire	Chapter 3.5, and our guide on surviving the security questionnaire
You think the code is the problem	Chapter 2, and Chapter 2.4 to find out whether it is
You want to know what the auditor finds first	Chapter 6.2

Edition 1.0, 31 October 2026. The only figures in this guide are two published industry figures, directional rather than audited, with their sources.

Contents

How to use this guide 4
1. What actually happens over four to six weeks 5
2. Ugly code is survivable. Unmanaged risk is not. 6
3. The five findings that actually sink deals 7
4. A delivery record answers the question they are really asking 9
5. What four weeks can fix, and what it cannot 10
6. What to do on Monday 11

APPENDICES

Appendix A. The diligence readiness sheet 12
Appendix B. A decision record 12
Appendix C. The key-person map 12
Appendix D. Questions the walkthrough will ask 13
Appendix E. A bounded scaling statement 13
Appendix F. Glossary 14
Appendix G. Sources, and what this guide does not claim 14

How to use this guide

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 guide is what that stranger is actually checking, in the order it tends to matter, and what you can do about each part before they arrive.

It is written for a founder or a CTO about to go through technical due diligence at Series A. Page 2 is the whole guide on one page: the five findings that sink deals, with the first move and a proof for each. Every chapter ends with a check you can run yourself, and the appendices are sheets to fill in this week.

What happens over four to six weeks, in the order it tends to happen.

1
Architecture walkthrough
Does your explanation match the diagram, and the diagram the scan?

2
Automated scan
Size, duplication, coverage, history, and who wrote what

3
Dependency and licence audit
Every package, transitively, with licence and known vulnerabilities

4
Security questionnaire
Access, secrets, backups, incidents, customer data

5
Interviews
Two or three engineers, separately, compared

The output is a report to the investor, not to you.

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.

How the guide is laid out

Chapter	The question it answers
1	What actually happens over four to six weeks
2	Why ugly code is survivable and unmanaged risk is not
3	The five findings that sink deals, and a check for each
4	Why 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
Appendices A to G	A readiness sheet, ten decision records, a key-person map, the walkthrough questions, a bounded scaling statement, a glossary, the sources

What this is not

It is not legal advice, and it does not tell you what any investor will decide. The two numbers in it are published industry figures, directional rather than audited. It does not claim that doing the six steps makes a company pass: it says what removes the findings that most often cost a round. And it is not a product: if you take all of it and never speak to us, the guide has done its job.

Conventions

- "The auditor" is whoever the investor has hired to run the diligence, usually a former CTO.
- A "proof" is something you can show a stranger, with a date on it. A statement that something is true is not a proof.
- Nothing here quotes a price.

1. What actually happens over four to six weeks

YOU WILL	know the five things that happen in a technical diligence, in the order they happen, and what each is listening for.
KEY IDEA	it is not a code review. It is a risk assessment that happens to involve code, and most of the four to six weeks is not spent reading your source.
PROOF IT WORKED	you can describe your own system three ways (to an auditor, in the diagram, and in a scan) and the three agree.

1.1 A risk assessment, not a code review

Technical due diligence is 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 section 1.3), and most of that time is not spent reading your source.

1.2 The five things that usually happen

1. **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.
2. **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.
3. **A dependency and licence audit.** Every third-party package, transitively, with its licence and its known vulnerabilities. The most mechanical part of the process and the one that most reliably produces a problem.
4. **A security questionnaire.** Access control, secret handling, backups, incident history, what customer data you hold and where. Usually a spreadsheet, and frequently the same spreadsheet your enterprise customers sent you, which is worth noticing.
5. **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".

1.3 The two numbers worth carrying into the process

Figure	Source
68% of audited codebases contain open-source licence conflicts	Technical due diligence benchmarks, 2026 (hypernestlabs.com)

4 to 6 weeks: typical duration of Series A technical due diligence	Technical due diligence guidance, 2026 (verycreatives.com)

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.

1.4 Check it worked

Ask one colleague to draw the system on a whiteboard from memory, a second to describe it aloud, and run the scan yourself. Put the three side by side. Wherever they disagree is where the walkthrough will go wrong.

2. Ugly code is survivable. Unmanaged risk is not.

YOU WILL	spend the weeks before diligence on evidence, not on tidying.
KEY IDEA	tidy code with no evidence around it scores worse than untidy code with evidence. That inversion is the whole reframe.
PROOF IT WORKED	for any change from last month, you can show a test, a second reader and a rollback.

2.1 The wrong fortnight

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 inversion: what an auditor prices.

Ugly code, with evidence

A test covers the refund path, a second person read the change, and you could roll back within the hour.

A cost. A line item. Negotiable.

Tidy code, no evidence

"We tested it locally and Dave usually catches things."

An unknown, priced at its worst plausible value.

2.2 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.

2.3 Why it matters more each year

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 AI SDLC Playbook.

2.4 Check it worked

Pick one change from last month. Without asking anyone, find: the test that covers it, the person who read it, and the way you would roll it back. Three finds is a cost. Fewer is an unknown, and the unknown is what they will price.

3. The five findings that actually sink deals

YOU WILL	know the five findings in order of how often they turn up against how much damage they do, and how to check each one yourself this week.
KEY IDEA	none of them is about how the code looks. Each is about whether risk is managed or merely absent so far.
PROOF IT WORKED	you have found all five before the auditor does, and you have said the awkward ones first.

3.1 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; directional, not audited). The 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.

Check it yourself. Run a licence scan over the full dependency tree, transitively, not only your direct dependencies. Free tooling does it. Write down the date, the tool and every licence it reports.

3.2 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, because 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.

Check it yourself. Print the authorship of your ten most important directories (Appendix C has the sheet). Wherever one name owns a directory outright, you have found it before the auditor.

3.3 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.

Check it yourself. Write down the three choices an outsider would most likely question. For each, can you find a dated record of the reasoning? Appendix B is the template.

3.4 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.

Check it yourself. Write the single thing that breaks first under ten times the load, with your reasoning and today's date (Appendix E). A rough bounded estimate beats a confident unbounded one.

3.5 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 our guide on surviving the security questionnaire covers it.

Check it yourself. List what runs on every change, what it blocks, and when it last failed. List who can reach production and the date you last reviewed that list.

3.6 Ten minutes: score the five

For each of the five, write "found", "not found" or "not checked", with the date. The "not checked" ones are your afternoon.

4. A delivery record answers the question they are really asking

YOU WILL	know what a delivery record is, what it answers, and why its failures are the valuable part.
KEY IDEA	every finding is a variation on one question: is risk managed here, or merely absent so far? Only a record of what actually ran answers it.
PROOF IT WORKED	you can show the last time a check blocked a release.

4.1 One question, five findings

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.

A delivery record, for every change that reached production.

The change what changed ▶ **What checked it** and what each returned ▶ **Who reviewed it** a second reader ▶ **The decision** which one it came from ▶ **What happened next** including the times it was stopped

A record that only ever shows green is evidence that nothing is checked hard enough to fail.

4.2 Read it against the five

- **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.

4.3 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: six checks, each with a recorded result and a stated point at which it runs. 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 our results page with their caveats. 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.

4.4 Check it worked

Find the most recent time a check stopped a change, and what was done about it. If you cannot, either nothing is checked hard enough to fail, or nobody can find the record. Both are findings.

5. What four weeks can fix, and what it cannot

YOU WILL	sort your problems into the ones four weeks can fix and the ones it cannot, and spend the month on the right side.
KEY IDEA	spending the month on the wrong side is worse than spending it on nothing.
PROOF IT WORKED	nothing in your document set has a uniform creation date, and everything dated says what prompted it.

5.1 The line

Assume the term sheet is signed and diligence starts in a month. Be honest about which side of this line each problem sits on.

Four weeks: what moves and what does not.

Not fixable in four weeks

- A genuine key-person dependency: transferring two years of context takes months
- A missing test suite: coverage added under deadline tests what is easy, and the commit dates show it
- Architectural choices that are now load-bearing: disclose and price them
- A history of decisions nobody wrote down: you cannot manufacture a two-year trail

Fixable in four weeks

- The licence position: an afternoon to scan, most conflicts are one swap
- Known vulnerabilities in dependencies: a patching exercise
- Secrets in repository history: rotate them, and say so
- An access review: who can reach production, who left and still can
- A dozen decision records, dated honestly as written now
- The walkthrough itself: rehearse until the explanation and the diagram agree

5.2 A four-week order

Week	Do	Why then
1	Licence scan of the full tree; rotate any secret found in history; print authorship of the ten directories	The mechanical findings, and the ones that need a decision about what to say first
2	Access review (who can reach production, who left and still can); write the first dozen decision records	Both need time to be honest, not time to be long
3	The bounded scaling statement; the list of what runs on every change and when it last failed	The two answers an auditor will test against your claim
4	Rehearse the walkthrough with someone outside the company, twice	By now there is something to rehearse

5.3 One warning

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.

5.4 Check it worked

Sort your documents by creation date. If most share one week, add a line to each saying what prompted it and when it was written, and start the habit that makes the date range wider.

6. What to do on Monday

YOU WILL	do six things, cheapest first, that cost an afternoon each rather than a quarter.
KEY IDEA	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.
TEN MINUTES	put a name and a date beside each of the six.

6.1 The six

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 Chapter 3, 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.

6.2 Where to start

If you want to know what an auditor would find first, give us read access to one system and we come back with what we found, where the risk sits and what we would do about it, before any commitment. If the six steps above are all you needed, that is a good outcome too.

6.3 Ten minutes

Copy the six onto a page. Beside each write who does it and by when. A step with no name will not happen.

APPENDICES

Appendix A. The diligence readiness sheet

One row per area. Fill it in this week and date it. "Where it lives" must be a place you can open in front of a stranger.

Area	What they will ask	What to have ready	Where it lives	Gap	Owner and date
Architecture	Explain the system, then show the diagram	A diagram that matches what you say and what a scan shows			
Codebase	Size, duplication, hot spots, test coverage, who wrote what	The scan you ran yourself, and what you will say about it			
Dependencies and licences	Every third-party package, transitively, with licence and known vulnerabilities	A dated inventory and the tool's output			
Security	Access, secrets, backups, incidents, customer data	The checks that run on every change, the access list with a review date, one incident written up			
People	Who built it, who can explain it, who leaves a hole	The key-person map (Appendix C) and what you are doing about it			
Decisions	Why this and not that	Dated decision records (Appendix B)			
Scale	What breaks first, and when	The bounded scaling statement (Appendix E)			

Appendix B. A decision record

Write one for each choice an outsider would question. A paragraph or two is enough. If you are writing it now about a choice taken earlier, say so in the date line.

```
Decision:        what we chose, in one sentence
Written on:      date. If it is later than the decision: "written now about a decision taken in <month
year>"
Decided by:      names
Context:         what was true then that made this a question
Options:         what else we considered, each in a line
Why this one:    the reasons, including the ones that did not flatter us
Revisit if:      what would make us change our mind (a volume, a price, a limit)
```

Ten to write first. Build or buy for the job queue; the primary database; hosting; how users sign in; monolith or services; the payments provider; how secrets are stored; the language and framework; how releases are made; why a package was replaced or a licence was bought.

Appendix C. The key-person map

One row for each of your ten most important directories. Two commands give you the first three columns:

```
$ git shortlog -sn --no-merges -- path/to/directory
$ git log -5 --format='%an %ad' --date=short -- path/to/directory
```

Directory	Main author	Last change by someone else	A second person who can explain it	What you are doing about it	By when

Wherever the second column is a single name and the third is empty or old, say so first, name the person, and put the plan beside it. The combination of the dependency and a denial is what costs a round.

A licence scan. Use whatever your language has (for example `pip-licenses` for Python or `license-checker` for npm), or the one your CI already runs. Run it on the full tree, write down the date, the tool and its output, and keep the output.

Appendix D. Questions the walkthrough will ask

Rehearse these with someone outside the company. A good answer is specific and can be shown. A warning sign is an answer that is a promise with nothing behind it.

Ask	A good answer sounds like	A warning sign sounds like
Why is the system built this way?	A reason, with a date, and what you rejected	"It was faster at the time"
Where does customer data live, and who can reach it?	A place, a list of people, and the date of the last review	"Only the team"
What breaks first at ten times the load?	One component, a rough bound, and the date you wrote it	"It should be fine, it is all managed"
Who can deploy, and how do you roll back?	Named roles, and a rollback you have done	"Dave usually handles it"
How do you know last month's change did not break refunds?	A test, a second reader and a rollback	"We tested it locally"
What runs on every change, and when did it last fail?	A list, and a date and a reason	"Everything is green"
Who can explain the payments path besides its author?	A name, and what they have changed	A pause
What was your last incident, and what did you do?	A write-up with times	"We have not had one"
Why did you build your own X?	A dated record of the reasoning	"I think it was cheaper"
Which of your dependencies has a licence you have not read?	A scan, a date, and what you did about the findings	"We use popular packages"

Appendix E. A bounded scaling statement

One page. The point is that it is bounded and falsifiable, not that it is exact.

```
Date:               today
Current load:       the number you measure now, and how you measure it
Plan:               the growth the round assumes
First to give way:  the one component, and why
Our estimate:       it holds to about <x> times current volume
Then:               what changes (partitioning, a bigger instance, a rewrite of one path), and about
how long
How we would know:  the signal that tells us we are close
```

"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 statement even if the estimate is rough.

Appendix F. Glossary

Copyleft. A kind of open-source licence that places conditions on distributing software that uses the covered code, which can reach your own source.

Data room. The shared folder of documents an investor's team reads during diligence.

Decision record. A short dated note of what was chosen, what was rejected and why.

Delivery record. For every change that reached production: what changed, what checked it and what the checks returned, who reviewed it, and which decision it came from.

Key-person dependency. Knowledge of an important part of the system that sits with one person.

Remediation item. A finding in the auditor's report that you are expected to fix, with a date.

Transitive dependency. A package your code does not use directly but that one of your dependencies does.

Appendix G. Sources, and what this guide does not claim

Sources.

- Technical due diligence benchmarks, 2026: the 68% licence-conflict figure, at hypernestlabs.com/guides/technical-due-diligence-checklist
- Technical due diligence guidance, 2026: the four to six week range, at verycreatives.com/blog/technical-due-diligence-for-founders

What this guide does not claim.

- It is not legal advice, and it does not predict what any investor or auditor will decide.
- The two figures are published, directional rather than audited, and are not a forecast of your process.
- We have not raised a round, and the guide says so. It is written from the delivery side.
- It does not say the six steps make a company pass. It says what removes the findings that most often cost a round.