Modern Technology Diligence

When a private equity firm is about to spend serious money acquiring a company, they hire consultants to spend time examining that company's technology — its architecture, its security posture, its development practices, its technical debt. They want to know what they're buying. They want to know what's going to break, what's going to cost money to fix, whether the company is well positioned to realize its strategic roadmap, and where the bodies are buried.

This makes sense. It's a lot of money. You'd be crazy not to look.

But here's the odd part: the company being acquired almost never knows the answers to these questions about itself. The things that investors want to know — things like "how coupled is your architecture?" and "what's your deployment frequency?" and "do you actually have a disaster recovery plan that works?" — are things the company should have known all along. They're not esoteric questions that only matter during an acquisition. They're questions that matter every single day, because the answers determine whether the company can ship fast, stay secure, scale when it needs to, and not hemorrhage money on cloud infrastructure nobody's watching.

I've been doing technology diligence for over twenty-five years, starting in 1999 when Java was still new and "Agile" didn't exist yet. Object-oriented programming was getting established, and the big methodology debates were about whether you needed to write a three-hundred-page requirements document before writing a line of code. The world has changed enormously since then, but the fundamental problem hasn't changed at all: most companies don't really understand the state of their own technology.

This isn't because they're stupid. It's because the day-to-day pressure to ship features, fix bugs, and keep customers happy leaves no room for the kind of systematic self-assessment that would reveal the real picture. The CTO knows some things. The VP of Engineering knows other things. Individual developers know things nobody else knows. But nobody has the whole picture, and without a structured way to assemble it, the whole picture never materializes. What you get instead are vibes. "I think our architecture is pretty good." "We're probably okay on security." "Our tech debt isn't that bad." These are not findings. They're feelings.

I think of technology diligence the way a doctor thinks of a physical exam. You can feel fine and still have high blood pressure. The point of the exam isn't to find out whether you feel healthy — it's to find out whether you are. And just as you don't wait until someone's buying your body to check your cholesterol, you shouldn't wait until someone's buying your company to check your architecture.

So why don't more companies do this for themselves?

Part of the answer is that they don't know how. When I started doing this work, there was no established framework for it. Each diligence engagement was essentially bespoke — a consultant would show up, ask a bunch of questions based on personal experience, poke around in the code, and write a report. The quality varied wildly depending on who you hired and what they happened to think was important. Over the years, I developed a more systematic approach — eighty-eight controls across ten domains, each with anchored rubrics that define exactly what a score of 1 through 5 means. Not "is your security good or bad?" but "do you have continuous scanning in your CI/CD pipeline with zero critical vulnerabilities in production, or have you literally never run a penetration test?" The specificity matters. Without it, everyone gives themselves a 4.

The ten domains cover the territory you'd expect: architecture and infrastructure, the software development lifecycle, security and compliance, data management, operations and reliability, product and roadmap, people and organization, cost structure, AI and ML capabilities, and governance. But the structure is more interesting than the list might suggest. Some domains are stack-dependent — they need to be evaluated separately for each product, because your Python microservices platform and your legacy PHP monolith are going to have very different scores. Others are organization-level — the security program, the hiring pipeline, the cost structure apply to the whole company. Getting this distinction right matters a lot when you're trying to understand a multi-product business.

In fact, the technology diligence investors pay for before writing a check is exactly the technology hygiene companies should be doing for themselves, continuously. The same eighty-eight questions that tell an investor whether to buy tell a CTO whether the machine is healthy. The same red flags that might kill a deal — no disaster recovery plan, a recent security breach, critical IP ownership gaps, key-person dependencies — are things any competent technology leader should be tracking without needing an outside examiner to discover them.

Consider tech debt, which is something almost every company acknowledges and almost no company manages. When I say "manages," I mean treats as a first-class concern with the same rigor applied to feature delivery. Most companies treat tech debt the way people treat flossing — they know they should do it, they feel guilty about not doing it, and they don't do it until something hurts. But tech debt compounds. An architectural shortcut that saves a week today costs a month next year and a quarter the year after. By the time it's painful enough to address, it's also painful enough that addressing it feels impossible. The features keep winning the prioritization battle because they have customers attached to them, and tech debt has nobody lobbying for it.

This is why I think companies need to invest in technology self-assessment the way they invest in annual financial audits. Not because they're about to be acquired, but because you can't manage what you don't measure. And the thing about technology is that its problems are invisible until they're catastrophic. A database with no backup strategy doesn't look any different from one with a solid backup strategy — right up until the moment the disk fails. Architecture that can't scale looks fine when you have a hundred users. Security that's held together with hope looks fine until the breach.

The good news — and this is genuinely exciting — is that the technology available today makes DIY technology diligence far more accessible than it was when I started. In 1999, evaluating a company's code quality required manually reading code. Today, static analysis tools can automatically assess code complexity, coupling, test coverage, dependency health, and known vulnerabilities. In 1999, understanding deployment practices required lengthy interviews. Today, you can look at CI/CD pipeline configurations and deployment logs. In 1999, security assessment meant hiring a specialist. Today, automated scanners, compliance monitoring tools, and cloud security posture management platforms do much of the heavy lifting.

The tools don't replace judgment. I want to be clear about that. They replace data gathering. And that's what makes DIY diligence possible. The hard part of a technology assessment was never the judgment calls — those require experience, but they're relatively quick once you have the facts. The hard part was getting the facts. Pulling together evidence from six different teams, reviewing documentation that may or may not exist, reconciling what people say with what the code actually shows. Automation can compress this dramatically.

But the real step change isn't static analysis or CI/CD dashboards. Those have been around for a while. The real step change is large language models, and I don't think most people in diligence have caught up to what they make possible.

Think about what a technology diligence engagement actually involves. You're trying to answer eighty-eight specific questions about a company's technology. Each question has defined rubrics — concrete descriptions of what a score of 1, 2, 3, 4, or 5 looks like. The answers live in documents, codebases, configuration files, architecture diagrams, Jira backlogs, incident reports, and the heads of engineers. The consultant's job is to find the evidence, evaluate it against the rubric, and form a judgment. Most of the time — maybe 70% — is spent on the finding, not the judging.

This is exactly what LLMs are good at.

Consider a control like "SDLC Process Maturity." The rubric says a score of 3 means "Scrum or Kanban practiced, regular ceremonies, definition of done exists." A score of 1 means "no defined process, ad-hoc development, no sprints, no planning ceremonies." To score this, a human consultant would interview the VP of Engineering, ask to see sprint boards, review retrospective notes, maybe sit in on a standup. It takes half a day.

An LLM with access to the company's project management tool can answer this in minutes. It can scan thousands of Jira tickets, identify whether sprints exist, whether they have consistent cadences, whether stories have acceptance criteria, whether retrospectives are logged, whether velocity is tracked. It's not guessing. It's reading the actual artifacts — the same artifacts the human consultant would review — and mapping what it finds to the rubric. The evidence is cited, traceable, and reproducible.

Or take architecture. One of the hardest things to assess in diligence is software architecture quality — the degree of coupling, modularity, separation of concerns. Traditionally, this requires a senior architect to review code, trace dependencies, and form an expert opinion. It's expensive and subjective. But tools like GitHub Copilot, Sourcegraph Cody, and Amazon CodeGuru can now analyze entire repositories. They can identify tightly coupled modules, trace dependency graphs, flag circular dependencies, assess code complexity metrics at scale, and even evaluate whether the code follows the patterns described in the architecture documentation. An LLM can read an architecture decision record and then check whether the codebase actually implements what the ADR says. That gap — between what a company says its architecture is and what the code shows it actually is — is where some of the most important diligence findings live.

Security is another area where this is transformative. Tools like Snyk and Semgrep have been doing static security analysis for years, but the integration of LLMs takes it further. Instead of just flagging a known vulnerability pattern, an LLM-enhanced scanner can assess context — is this vulnerable code path actually reachable in production? Is the mitigation the team claims to have in place actually effective? Checkmarx and SonarQube have both added AI-assisted capabilities that go beyond pattern matching into something closer to reasoning about security posture. These aren't replacing the human security reviewer. They're doing the first pass — the evidence gathering — so the human can focus on judgment.

What makes this particularly powerful for diligence is the evidence framework. In the system I've built, every score has an associated evidence quality rating from 1 to 5. A "5" means system-generated data independently verified. A "1" means hearsay — a verbal claim with nothing behind it. The gap between these two is enormous in practice, and it's exactly the gap that LLMs can close. When an LLM scans a codebase and reports

"deployment frequency averages 3.2 times per week based on 847 merged pull requests over the last six months," that's a 5 — verified, system-generated, independently confirmable. When a VP of Engineering says "we deploy several times a week," that's a 2 — attested, with some supporting context, but unverified. The LLM doesn't just find the answer faster. It finds a better-evidenced answer.

The implications for self-assessment are even more interesting. A PE firm's diligence consultant shows up for two to four weeks, and the clock is ticking. They can't read every file, review every ticket, or interview every engineer. They sample. They triangulate. They use judgment to fill in gaps. But a company assessing itself has no time constraint and full access to its own systems. An LLM-powered assessment tool can be pointed at the entire codebase, the entire ticket history, the entire documentation library. It can run continuously. It can flag changes — "your deployment frequency dropped 40% this quarter" or "three new critical dependencies were added without security review." The assessment stops being a periodic snapshot and becomes a living signal.

I realize there's a reasonable objection here, which is that LLMs might hallucinate. They might make things up. How can you trust an LLM to assess your architecture if it might confabulate findings? This is a fair concern, and the answer is that the framework itself is the safeguard. Each finding is tied to specific evidence — a file, a commit, a ticket, a configuration — and that evidence is cited. You're not asking the LLM to render an opinion. You're asking it to find artifacts that match rubric criteria and show its work. If it claims your test coverage is 80%, you can check. If it says your deployment pipeline has no approval gates, you can verify. The LLM is doing research, not rendering verdicts. The judgment still belongs to the human. But the research that used to take days now takes hours, and the evidence trail is more complete than any human could produce manually.

There's another dimension here that most people miss when they think about LLMs and code. The popular narrative is about code generation — autocomplete on steroids, copilots that write functions for you. That's real and useful, but the more powerful application is comprehension. An LLM can read an entire repository and produce human-readable documentation that explains what the code actually does — not what someone intended it to do three years ago when they wrote a README that nobody updated, but what it does today, in practice, with all its warts and workarounds. This is transformative for diligence because one of the most common findings is that documentation doesn't match reality. The architecture diagram on the wiki shows a clean microservices topology. The code shows a distributed monolith with services that can't be deployed independently. An LLM that reads the code and generates accurate, current documentation closes that gap.

And here's where it gets recursive in a useful way: the documentation an LLM produces from reading code isn't just valuable to human engineers. It's also valuable to LLMs themselves. The next time you point an LLM at that codebase — for a diligence assessment, for a security review, for onboarding a new developer — it has better context to work with. Good documentation makes LLMs more effective, and LLMs can produce good documentation. This is a virtuous cycle that didn't exist before, and it fundamentally changes the economics of keeping a codebase well-documented. The excuse "we don't have time to write docs" loses its force when an LLM can generate a first draft that's 80% right in an afternoon.

This dual-audience principle extends to code generation itself. When an LLM writes new code — and yes, this is where code generation genuinely matters — the best practice is no longer just "write clean code." It's "write code that both humans and LLMs can reason about." That means meaningful variable names, yes, but also inline comments that explain intent, not just mechanism. It means docstrings that describe not just what a function does but why it exists and what assumptions it makes. It means architecture decision records that capture the trade-offs considered, not just the choice made. Traditionally, developers resisted writing all this because it was tedious and because the primary audience was other humans who might never read it. But now there's a second audience that will absolutely read it: the LLM that runs the next security audit, the next diligence review, the next onboarding walkthrough. Writing for both audiences simultaneously isn't twice the work — it's the same work, done slightly more carefully. And the payoff compounds. A codebase that is well-documented and well-commented doesn't just help the next human developer. It makes every future LLM interaction with that codebase more accurate, more contextual, and more useful. You're not just writing for today's team. You're writing for every machine that will ever need to understand what you built.

And this changes the calculus on maintenance and extensibility in a way that wasn't possible before. The perennial nightmare of software — the system nobody dares touch because the person who understood it left three years ago — starts to dissolve when the codebase carries its own explanation. Code generated with both audiences in mind is code that can be safely modified, extended, refactored, and debugged by whoever comes next, human or machine. An LLM that encounters well-annotated code can propose extensions that respect the original design intent. It can refactor a module without inadvertently violating assumptions buried in some engineer's head that never made it into a comment. The maintenance burden that slowly crushes most software organizations — the reason systems calcify and rewrites get proposed — drops substantially when the code itself is a living, readable artifact that any competent reader, carbon-based or silicon-based, can pick up and work with. This is perhaps the most underappreciated consequence of the

dual-audience principle: it doesn't just make today's code better. It makes tomorrow's changes safer.

The same comprehension capability applies to finding and fixing flaws. An LLM reviewing code isn't just looking for known vulnerability patterns the way a traditional static analyzer does. It can reason about logic errors, identify race conditions, spot places where error handling is incomplete, and flag code paths that violate the assumptions documented elsewhere in the system. More importantly, it can propose patches — not just say "this is wrong" but show what "right" looks like. The combination of identification and remediation in a single pass is something no traditional tool offers. A senior engineer still needs to review and approve the fix, but the time from discovery to resolution compresses dramatically.

This is why I think the framing of LLMs as "code generation tools" fundamentally undersells what they make possible. Code generation is about producing new code faster. But the applications that matter most for technology health — documentation, comprehension, flaw detection, evidence gathering, compliance verification — are about understanding existing code better. They're about diligence in the original sense of the word: careful, persistent effort applied to understanding what you have and what state it's in. When you use LLMs not just to write code but to read it, to document it, to interrogate it, to heal it, the entire practice of technology management becomes more thorough. You can afford to be more diligent because the cost of diligence just dropped by an order of magnitude.

Evidence-based research informs good judgment. That's always been the core principle of sound diligence — on an architecture, on a codebase, on a company's entire technology platform and organization. What's changed is that the research step, the painstaking gathering and organizing of facts, is now something machines can do with a thoroughness and speed that humans simply can't match. The judgment remains irreducibly human. But the foundation on which that judgment rests just got a lot stronger.

Emerging technologies allow a CTO to run a structured technology assessment quarterly, using the same framework and the same rigor that a PE firm's diligence consultant would use, but in a fraction of the time and at a fraction of the cost. Not because the company is for sale, but because knowing the health of your own technology should be table stakes for running a technology company. The assessment becomes a dashboard, not a dossier. You track your Tech Health Index the way you track your ARR. You know your scores by domain. You know where your red flags are. And when that investor does come knocking, you already have the answers — not because you crammed for the test, but because you've been doing the homework all along.

There's a deeper strategic point here that most technology leaders miss. It's not enough to build what customers ask for. You also have to build what customers don't know they need — which is, in large part, the invisible infrastructure that keeps the whole thing from falling over. Customers will never ask you to refactor your database schema. They'll never request a disaster recovery test. They'll never put "fix your dependency management" on a feature request. But these things determine whether you're still in business in five years. The companies that get this right are the ones where technology leadership has the credibility and the data to say "we're spending 20% of our capacity on platform health this quarter" and actually mean it.

I've seen this pattern play out hundreds of times. The companies that invest in self-knowledge — that maintain architecture strategies, that track their core metrics, that regularly assess their security posture, that keep their tech debt under control — are the ones that score well in diligence, yes, but more importantly, they're the ones that ship faster, break less, scale more smoothly, and retain their best engineers. The diligence score is just a symptom. The underlying health is the cause.

When an investor hires a consultant to evaluate a company's technology, what they're really asking is: does this team know what they're doing, and can we trust what they've built? The irony is that the best way to answer "yes" to both questions is to have been asking them of yourself all along.