

MVP: Minimally Viable or Meaningfully Viable?

Welcome to InCogSights™, Issue Four.

Everyone is selling you an MVP

If you're shopping for a LIMS right now, you've probably heard some version of this pitch: "Let's get you live fast with core functionality, then build from there."

It's borrowed from software startups, where the Minimum Viable Product is the smallest thing you can ship to learn whether anyone wants it. And it's not a bad idea. A lab that goes live in six months with a focused scope usually beats one that spends two years chasing every requirement on the wish list.

But there's a catch the pitch leaves out. A startup's MVP can be thrown away. Your LIMS can't. Whatever you configure in phase 1 is what your data lives in, what your auditors review, and what phase 2 has to be built on top of.

I watched this play out at a contract lab whose implementation hadn't been scoped thoroughly enough at the start. To get the LIMS up and running, the team proposed an MVP, with the M meaning Minimum.

It sounded like a great idea. But it came with an unspoken expectation: minimum means fast. Over time, "minimum" became more of a label than a plan. It made the schedule look good without producing a system that would actually deliver what the lab needed to get started.

The lab did eventually go live on the minimum, but only on paper. The system didn't satisfy even the minimum functional requirements, so the project was delayed again by significant rework. In the end, it took three times as long to go meaningfully live as it would have if those pieces had been built into the initial launch.

I've been on every side of these projects: the bench, the customer's project team, the vendor's implementation team, and now as an independent. The pattern is the same everywhere. Some things you can safely leave for later. Some things you pay for now, or you pay for three times later.

The wrong question and the right one

Most MVP scoping sessions ask: what's the least we can go live with?

That question sorts features by how much people want them. Must-haves go in phase 1, nice-to-haves go in phase 2. It sounds disciplined, but it misses the variable that actually drives cost.

The better question is: what is expensive to change after go-live?

That sorts everything by how hard it is to retrofit. And the answer splits a LIMS into two very different layers:

- Features sit on top. Reports, dashboards, extra workflows, secondary instrument interfaces. Add them in phase 2 and nothing underneath has to move.
- Foundations sit underneath everything. The data model, how samples and tests are named, how specs and limits are structured, who can do what. Change them after go-live and you're migrating data, rewriting configuration, and revalidating.

A minimally viable product keeps the features minimal. A meaningfully viable product does that too but gets the foundations right on day one.

Build it now, or rebuild it at 3x

In my experience, a foundation decision fixed after go-live costs at least three times what it would have cost up front, and that's a conservative estimate. And "cost" isn't just money:

- **Time.** The rework runs alongside daily lab operations, so it takes longer than the original build would have.
- **Resources.** The same analysts, QA reviewers and super users who run the lab every day get pulled back into workshops, testing and revalidation.
- **Confidence.** This is the cost nobody budgets for. While the team is redoing things, users start working around the system, keeping side spreadsheets, and second-guessing the data. Once a lab stops trusting its LIMS, that trust is slow to win back.

The reason the bill grows is simple: later, you're not just making the change. You're also undoing the old way, moving the data, retraining people, and revalidating, all while the lab keeps running.

Here's where the bill usually comes due:

Foundation	Get it right now	Fix it later
Data model and sample structure	A few workshops to map how your lab really works	Restructure live data, rebuild configuration, revalidate
Sample and test naming	Agree on conventions before anything is entered	Years of inconsistent records that won't search or trend
Specs and limits	Structure them so products, grades and customers can share them	Duplicate spec sets that drift apart and fail on the COA
Security roles	Define who reviews, approves and releases	Rework roles under audit pressure, with access gaps exposed
Audit trail settings	Decide what is tracked and why, and document it	Explain to an auditor why history is missing for a period
Legacy data	Clean it before it comes over	Clean it inside a validated system, with every change on the audit trail
Report foundations	Make sure the data your COAs need is being captured	Discover at go-live that a required field never existed

Notice what's not on the list: dashboards, extra reports, convenience workflows, the third instrument interface. Those are fine to defer. That's the part of the MVP idea that's right.

Why smart labs still choose "later"

Nobody decides to build on a weak foundation. It happens because of pressures that feel reasonable at the time:

- **The budget cycle.** Phase 1 has a fixed number. Foundation work doesn't show up in a demo, so it's the first thing trimmed to fit.

- The need to show progress. Leadership wants to see samples moving through the system. Data model workshops don't make a good status slide.
- The assumption that the vendor will catch it. Vendors configure what you tell them. A good implementation team will raise concerns, but they're working to a scope and a schedule, and the decisions about how your lab works are yours.
- "We'll clean it up in phase 2." Phase 2 often gets smaller, later, or cancelled once the system is live and the project team goes back to their day jobs.

I've seen every one of these from the vendor side. None of them is a bad reason on its own. Together, they're how a minimum viable product turns into a maximum rework project.

From minimally to meaningfully viable: five moves

You don't have to give up a fast, focused go-live to get this right. You just have to scope it differently.

1. Sort your requirements by retrofit cost, not just priority. Next to every item, ask: if we defer this, what has to change underneath when we add it? If the answer is "nothing," it's a phase 2 candidate.
2. Protect foundation time in the schedule. Data model, naming and spec workshops get dedicated time before configuration starts. Don't let them be squeezed into the build.
3. Design for phase 2 during phase 1. You don't have to build the stability module now, but the sample structure should be able to hold it when you do.
4. Clean legacy data before migration, not after. Outside the validated system, cleanup is a spreadsheet exercise. Inside it, every correction is a documented change.
5. Get a second set of eyes on the foundation. Before you sign off on configuration, have someone who has seen many implementations look at the data model and spec structure. It is the cheapest review you'll ever buy.

Go meaningfully viable, not minimally viable

The MVP approach is a good one. Phase your features, go live sooner, learn from real use. Just keep the minimum on the surface and build the foundation for the system you'll have in five years, not the one you'll have in six months.

Go live small if you need to. Just don't go live shallow.

If you're scoping a phased implementation right now, an independent foundation review before configuration sign-off costs a fraction of the rework it prevents. A Cogsult CogBlock™ of prepaid hours is built for exactly this kind of focused, as-needed check. DM me or schedule a free 30-minute consultation.

Thanks for reading, and I'll see you in the next issue.

Matt