What Slow Building Was Doing

This piece extends Sustaining the Product Engineer. While that work identified human comprehension as an implicit yield of slow code production, the same structural mechanism is responsible for the logical discovery that iterative building relies upon. This piece isolates that second yield.

1. Two kinds of discovery

Iterative development begins before the problem is fully understood, and this has always been the rational way to work. Some of what a piece of work requires cannot be known in advance. It appears only once building is under way: how the parts resist each other, how the system behaves once it exists, what the problem really was rather than how it first looked. Knowledge of this kind cannot be acquired any earlier, because earlier it does not exist. Starting with a partial understanding and letting the building supply the rest was never a concession. It was the only way to reach the understanding at all.

There is a second kind of understanding mixed in with the first, and it behaves differently. Some of what gets discovered during a build was knowable all along: a contradiction between two requirements, a term used in two senses, an edge case nobody had named. Nothing about it required the build. It surfaced during construction only because construction ran into it, and because letting it surface there felt free. This is the understanding this piece is about. It used to appear during the build. It now has to appear before.

What changed is not the understanding but the build. Producing working software used to take weeks, and across those weeks the knowable detail surfaced on its own, paid for by time that was being spent anyway. AI has collapsed that span. The build no longer lasts long enough to surface anything incidentally. So the detail it used to reveal has nowhere left to hide, and surfaces at the front instead. This is not new work. It is the same discovery, relocated, because the slow process that used to host it is gone.

2. Why discovery was deferred

The iterative model was a rational response to a specific condition: building was the expensive part. Perfect foresight is both unreliable and costly to produce. If a slow construction process is going to absorb the logical gaps anyway, there is little reason to pay for foresight up front. Deferral was simply cheaper.

But cost was not the only thing pulling in this direction. Deferral also licensed impatience.

Starting is inherently more satisfying than specifying. A team eager to see a system run would always rather begin building than sit longer with a document. The slow build was what made this impatience safe. It acted as a high-friction buffer, catching what eagerness skipped and surfacing

omissions before they could do real damage. The buffer held so reliably that the urge to begin rarely had to be examined.

So discovery was deferred into the build, not because deferral was virtuous, but because a slow build was a forgiving place to host it. The team could learn what it needed to learn in installments, folding the logic back into the code as it went. This is the familiar, choppy rhythm of iterative development: build, bump into a contradiction, revise the understanding, build again.

This rhythm relied entirely on an assumption so foundational it was never stated: that building would always take long enough for the discovery to happen along the way. The deferral was safe because the build was slow.

3. The exposed constraint

Understanding what to build was always slow and effortful. Working out what a system should do, resolving the contradictions in what stakeholders ask for, defining the terms precisely enough that the definitions hold under pressure, this was never quick. But it was never the binding constraint, because building was slower still. A team limited by build time does not experience the slowness of understanding as the thing holding it back. It experiences that slowness as work that gets done gradually, paid down across the long build, never the gate.

AI collapses the build. Producing working software, which took weeks, can now be done in a fraction of the time. This does not make understanding any faster. Working out what to build is exactly as slow and effortful as it always was. But it is now the slowest remaining part of the process, which means it has become the binding constraint. The slowness of understanding was always present. It was masked by the greater slowness of building, and the collapse of building has exposed it.

This is what is happening when a specification session presses on detail that would once have been left for later. The detail is no different, and no more costly to resolve, than it ever was. What has gone is the slow build that used to carry it. Understanding cannot be handed off to something faster than itself, and nothing downstream is slow enough any more. So the resolving has to happen at the front, because the front is the only place left where it can be done cheaply.

4. The objection

There is an obvious objection, and it is the strongest one available. If building is now cheap, why front-load anything? Cheap building should make iteration more attractive, not less. The entire cost argument for specifying up front rested on building being expensive. Remove the expense and iteration becomes cheaper than it has ever been. Build a rough version in an afternoon, see what breaks, adjust, build again before lunch. The rational response to cheap building looks like more iteration, not less.

This objection is correct about cost and wrong about mechanism. It assumes that the value iteration produced came from the iterating, from the loop of build, learn, revise, build again. If the value came from the loop, then making each pass through the loop cheaper would indeed make iteration more attractive. But the value did not come from the loop. It came from something the loop organised but did not itself produce.

5. What the loop was organising

Iterative building did not generate discovery because iteration is inherently generative. It generated discovery because building slowly forced contact with detail. Every hour spent constructing a system was an hour of friction against reality, and that friction is what surfaced the gaps. The edge case became visible because the code that handled it had to be written. The contradiction became visible because both requirements could not be implemented at once. The bad assumption became visible because the system built on it did not behave as expected. The discovery was produced by the slow drag of construction across every detail, which made the problems impossible to skate past.

The loop did not produce this. The loop organised it into manageable units. Iteration was the structure that took the slow contact with detail and broke it into chunks a team could plan around. The discovery was a function of the slowness, and the loop was a way of managing the slowness, not the source of what the slowness produced.

This is the same mechanism that produced comprehension as the yield of slow code production. There, the slow pace of typing forced an engineer into sustained contact with each unit of code, and understanding accumulated as a by-product. Here, the slow pace of building forces a team into sustained contact with the details of the system, and discovery accumulates as a by-product. In both cases the medium was producing something valuable that nobody had set out to engineer, and in both cases the value was a function of the pace.

Collapse the pace and the by-product stops. Iteration is still possible, and each pass is faster than ever, but nothing drags slowly across the detail any more. The build skates over it, finished before the friction that used to surface the gaps has time to act. Fast iteration is not slow iteration sped up. It is a different activity that shares a name, and it has lost the property that made the original valuable. A team can now iterate ten times as fast and discover less per pass, because the discovering was never the loop. It was the slowness the loop contained.

6. Where discovery goes

Discovery does not disappear. It relocates. The work of surfacing the gaps, the contradictions, the bad assumptions still has to happen, because the gaps are still there. What changes is where in the process it happens. Under slow building, it happened throughout, dripped across the long construction. Under fast building, it has to happen before, concentrated at the front, because there is no longer a slow build to host it.

This is what the front-loaded specification work is doing. The pressing for clarity, the surfacing of contradictions, the demand for tighter definitions: this is the discovery work, concentrated at the front and done by design. It does up front, through questioning, the contact with detail that slow building used to force incidentally, through construction. The front-loading is not a delay before the real work. It is the relocated form of work that used to be spread across the build.

This returns to the distinction the start of this piece drew. Of the two kinds of discovery, only one has moved. The first, the understanding that genuinely could not have been reached earlier, still cannot be. How users respond, how the system behaves under real load, what the market does, these emerge only when the world is given something to react to, and no amount of front-loading can surface them before they exist. This discovery still happens during and after the build, and the iteration that exists to capture it remains rational. The collapse of build time does not touch it. What has moved is the second kind: the discovery that was always possible up front but was cheaper to postpone while the slow build picked it up for free. That deferral has lost its justification. The detail was available all along. Only the slow build that used to surface it is gone.

7. Closing

The choice is not between cheap iteration and expensive front-loading. That framing assumes fast iteration still delivers what slow iteration delivered, and it does not. The discovery that iterative building produced was a function of the slowness of building, and the slowness is gone. Iterating faster does not recover it. It produces more passes, each surfacing less, until the gaps that slow building would have caught are shipped instead of solved.

So the front-loaded understanding is not better because foresight is virtuous or because planning is superior to building. It is better because the alternative has quietly stopped working. The discovery has to happen somewhere. Slow building used to be where it happened, and slow building is gone. Resolving the understanding before the build is not inefficiency. It is the work the medium used to do slowly and throughout, now done deliberately at the front, the only place left that can host it.