



BEYOND PORTABILITY

Vendor Lock-In 2.0: How the Cloud Rebuilt the Cage We Spent a Decade Escaping

The dependency did not disappear when we left the data centre — it moved up the stack, changed its texture, and learned to feel like progress

Giovanni Leonardi

November 2016

Contents

1. Executive Summary	3
2. The bill that could not be argued with	3
3. A dependency we have met before	4
4. The promise we made ourselves	5
5. Why the cage reassembles	5
6. The strongest case for not caring	6
7. The false exits	7
8. Conscious dependency	7
9. Coda	8

Vendor Lock-In 2.0: How the Cloud Rebuilt the Cage We Spent a Decade Escaping

Transformation · No. 249

Originally written November 2016 | Re-edited and re-rendered for the WhereBeyond Edition, 2025

"You do not need to own the cage for it to hold you."

1. Executive Summary

For most of a generation, vendor lock-in had a name and a logo. Organisations that had spent years inside a mainframe estate, an Oracle licence, or a proprietary Unix learned to treat dependency on a single supplier as the original sin of technology strategy, and the long move to commodity hardware and open standards was, in large part, a campaign to escape it. The public cloud was sold as the completion of that campaign: no more capital tied up in machines, no more three-year hardware refresh, no more supplier holding the estate hostage at renewal. Rent what you need, hand back what you do not, walk away whenever you like.

This essay argues that the escape was genuine and the freedom was not. What organisations left behind when they closed their data centres was a particular form of dependency — on physical assets and the vendors who sold them. What they acquired was a newer and subtler one, higher up the stack and far more comfortable to live inside. The old lock-in was visible on a balance sheet and hated accordingly. The new lock-in arrives disguised as productivity, priced as convenience, and defended by the very engineers it binds. It is harder to see precisely because it feels like progress.

I do not argue that this is a scandal, or that the cloud was a mistake, or that dependency can be engineered away. The portability fantasy — the dream of an architecture so cleanly abstracted that any provider is interchangeable — is, in most cases, a tax paid for a freedom never used. The argument is narrower and, I think, more useful: that dependency has not disappeared but changed shape, that the new shape is easy to acquire unconsciously, and that the only mature response is to choose it deliberately, price it honestly, and know exactly what it would cost to leave.

2. The bill that could not be argued with

The moment of recognition, when it comes, is usually a number. An organisation three years into an enthusiastic cloud migration sits down to model what it would take to move a single large workload to a different provider — not because it wants to, but because a board member has asked the reasonable question of whether it could. The compute is portable enough. The trouble starts with the data. Several hundred

terabytes now live in the provider's object store, and the price of reading them back out across the public internet — the egress charge that nobody costed at the beginning because nobody was ever going to leave — turns out to be a five- or six-figure line item on its own, before a single engineer's time is counted. Then someone lists the proprietary services the application has quietly come to depend on: the managed database with the provider's own operational model, the message queue, the identity layer, the data warehouse that made last year's analytics programme possible. None of it has a drop-in equivalent elsewhere. Re-platforming would mean re-engineering.

The board's question — could we move — receives its honest answer: yes, in the way that one could move a house brick by brick. The organisation is not trapped by a contract. It is trapped by gravity.

The old lock-in was written into a licence agreement and hated at every renewal. The new lock-in is written into an architecture and loved at every sprint review.

3. A dependency we have met before

It helps to remember how thoroughly we understood the old version, because the new one rhymes with it more closely than we like to admit.

The mainframe era taught a whole generation of technology leaders what dependency felt like from the inside. The hardware, the operating system, the database, and often the applications came from one supplier, and the supplier knew it. Migration was theoretically possible and practically unthinkable; the estate had accreted two decades of undocumented dependency, and the skills to run it were scarce and ageing. Renewal negotiations were not really negotiations. The relational-database and enterprise-application vendors of the following era refined the same model into an art: proprietary extensions that were genuinely convenient, that developers adopted willingly, and that turned out to be the bars of the cage when the invoice arrived. Even the move to "open" Unix delivered less freedom than advertised, because each vendor's flavour diverged just enough that portability remained a slide rather than a fact.

The lesson we drew from all of this was a good one, and it drove twenty years of strategy: prefer commodity over proprietary, prefer open standards over a single supplier's cleverness, keep the exit cheap even when you have no intention of taking it. That instinct built the case for the cloud in the first place. The irony is that it is precisely that instinct which the cloud has most quietly eroded.

Dimension	The old lock-in (mainframe / licence era)	The new lock-in (cloud era)
Where it lives	Hardware, operating system, licence agreement	Managed services, data gravity, operating model
How it feels	A cost centre and a constraint, resented	A productivity gain, welcomed by engineers

Dimension	The old lock-in (mainframe / licence era)	The new lock-in (cloud era)
When you notice	At every contract renewal	Only when you try to leave
What binds you	Capital, contracts, scarce legacy skills	Proprietary APIs, egress economics, muscle memory
Who defends it	The vendor's account team	Your own delivery teams

4. The promise we made ourselves

The cloud's liberation narrative was not a lie, and it is worth being precise about what it did deliver, because the argument here is not that the freedom was fake. Capital expenditure genuinely became operating expenditure. The three-year hardware refresh genuinely disappeared. Capacity that once took a quarter to procure genuinely arrived in minutes, and the ability to hand it back when demand fell was genuinely new. For organisations whose growth was throttled by the physical supply of infrastructure, this was not a marginal improvement; it was a change in what was possible.

But the narrative carried a second, softer promise that has aged less well: that because you were renting rather than owning, you were free. Ownership had been the thing that trapped you before — the sunk capital, the depreciating asset, the estate you could not walk away from — so its removal felt like the removal of the trap itself. This was the sleight of hand, though no one performed it deliberately. Dependency had never really been about ownership. It had been about the cost of leaving. And the cost of leaving, it turned out, could be rebuilt just as effectively out of proprietary services and accumulated data as it had ever been built out of hardware and licences. You do not need to own the cage for it to hold you.

5. Why the cage reassembles

The most important thing to understand about the new lock-in is that no one imposes it. The old version was pressed on you by a vendor with leverage; the new version is assembled, enthusiastically, by your own teams, one sensible decision at a time. That is what makes it so durable. Several forces drive it, and each is individually rational.

1. The proprietary services are genuinely better. The provider's managed database, its data warehouse, its message service: these are not traps laid to ensnare you. They are excellent, and they remove real toil — the "undifferentiated heavy lifting" the providers rightly promise to take off your hands. An engineer who reaches for the managed service instead of running the open-source equivalent herself is making the correct decision for this sprint. The dependency is the by-product of a hundred correct decisions.
2. Data has gravity. Once a large body of data lives in a provider's store, everything else is drawn towards it. Compute wants to be near the data to avoid latency and

egress cost; new services are built where the data already is; and the data itself grows faster than anyone's appetite for moving it. Gravity is not a contract term. It is physics, and it compounds silently.

3. The economics punish leaving, not staying. Ingress is free; egress is charged. Long-term commitments — reserved capacity bought a year or three ahead — lower the running cost while raising the cost of change. The pricing is not malicious, but its gradient always points the same way: cheap to deepen the relationship, expensive to unwind it.
4. The operating model binds hardest of all. This is the one organisations consistently underestimate. Over three years, teams do not merely adopt a provider's services; they build their skills, their tooling, their automation, their on-call runbooks, and their instincts around that provider's particular way of working. The muscle memory is the deepest dependency of the lot, and it appears on no architecture diagram. Re-platforming the software is a project. Re-platforming the people is a culture change.

Put these together and the pattern is clear: the new lock-in is not something that happens to an organisation. It is something an organisation does to itself, for good reasons, without noticing the aggregate. Every step is defensible. The destination is a dependency as complete as any mainframe estate, arrived at with enthusiasm rather than resentment.

6. The strongest case for not caring

Here I have to put the opposing view at full strength, because it is held by serious people and it is not wrong.

The case runs like this. Lock-in is a bogeyman inherited from an era that no longer applies. The whole point of the cloud is to stop treating infrastructure as something you hedge against and start treating it as a source of leverage. The proprietary services are where the value is; refusing to use them in the name of some hypothetical future migration is refusing the entire benefit of being there. Portability is not free — architecting for it means constraining yourself to the lowest common denominator across providers, forgoing exactly the high-value services that justified the move, and paying a permanent tax in velocity to preserve an option you will almost certainly never exercise. How many organisations, in practice, actually switch primary cloud providers? Vanishingly few. So why impoverish every day of your delivery to insure against an event that will probably never happen? Depend deeply, extract the full value, and stop worrying about a cage you have no intention of leaving.

This is a strong argument and I concede most of it. The portability tax is real, the lowest-common-denominator trap is real, and an organisation that cripples its use of the cloud to preserve a theoretical exit has usually made a worse decision than one that commits wholeheartedly. If the choice were simply between anxious, expensive portability and confident, productive dependency, I would choose dependency most of the time.

But the argument has a hole, and it is the same hole every time: it conflates depending deeply with depending unconsciously, and it treats the exit cost as irrelevant because

the exit is unlikely. Those are two different claims, and the second does not follow from the first.

7. The false exits

Before I say what conscious dependency looks like, it is worth clearing away the responses that pretend to solve the problem and mostly do not, because a great deal of energy at the end of 2016 is being spent on them.

The most popular is multi-cloud as a portability strategy — the idea that by spreading workloads across two or more providers you neutralise dependence on any one. In practice, run as a genuine portability play, this tends to deliver the worst of both worlds: you still depend on each provider for the workloads that live there, you forgo the deep proprietary services on all of them in the name of parity, and you double your operational surface, your skills burden, and your integration complexity. There are excellent reasons to use more than one provider — a specific capability, data residency, an acquisition you have to absorb — but "so that we are not locked in" usually produces a more expensive dependency, not a smaller one.

The second is the abstraction layer: the internal platform that promises to hide the provider behind a neutral interface so that the choice underneath becomes swappable. Sometimes, for narrow and stable needs, this works. More often the abstraction leaks the moment anyone wants a service the layer does not expose, becomes a bespoke product that must itself be maintained forever, and ends up being its own form of lock-in — this time to a homegrown platform that only your team understands and that no market of skills supports.

Containerising everything is sometimes offered as the third exit — package the workloads so they run anywhere. Portability of the runtime is real and worth having, and the container tooling maturing this year makes it more real than it was. But it addresses the shallowest layer of the dependency. It does nothing about the managed database you are calling, the data sitting in the provider's store, or the operating model your teams have built. It moves the code. It does not move the gravity.

"Portability of the runtime is the easy tenth of the problem. The data and the muscle memory are the other nine, and neither fits inside a container."

8. Conscious dependency

If the fantasy of the frictionless exit is a dead end, and unconscious dependency is a slow trap, what is left is a discipline rather than an architecture. Conscious dependency means depending as deeply as the value justifies while refusing to do so by accident. In

practice it comes down to a few honest habits.

Know the price of leaving, and revisit it. Not because you intend to leave, but because a dependency whose exit cost you have never calculated is one you do not actually understand. The number — egress plus re-engineering plus retraining — is the truest single measure of how bound you are, and watching it grow tells you more about your strategic position than any architecture review.

Distinguish the dependencies worth deepening from the ones worth containing. Not all lock-in is equal. Binding yourself to a provider's excellent data warehouse to unlock an analytics capability you could not otherwise have may be an excellent trade. Binding your core identity and data layers so tightly that the whole enterprise has a single point of commercial failure is a different kind of decision, and it should be made at the level that owns enterprise risk, not settled by default in a backlog.

Make the deep dependencies deliberate and visible. The problem is almost never that an organisation depends on its provider. The problem is that it never decided to, and so no one owns the decision, prices it, or reviews it. A dependency chosen with open eyes, costed, and signed off is a strategy. The same dependency arrived at through a hundred unexamined sprint decisions is a liability wearing the mask of progress.

9. Coda

We did escape the mainframe. That should not be waved away; a generation of technology leaders spent their careers trying to, and the cloud finished the job. But we should be honest that we escaped a form of dependency, not dependency itself — and that the form we walked into is more comfortable, more productive, and for exactly those reasons harder to see than the one we left. The old cage was made of steel and resented. The new one is made of convenience and loved. The task is not to smash it, which would be foolish, nor to pretend we are standing in open country, which would be a lie. The task is to know we are inside it, to have chosen it on purpose, and to keep an honest account of what the door would cost to open. That is not freedom. But it is the closest thing to freedom that anyone renting their foundations has ever actually had.