



BEYOND EASY CONNECTION

The API Economy's Hidden Cost: Why Integration Programmes Grow More Complex

How microservices redistribute coordination across complex enterprises

Giovanni Leonardi

November 2016

The API Economy's Hidden Cost: Why Integration Programmes Grow More Complex

Programme · No. 191

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

"Integration debt is therefore not the number of interfaces in an estate. It is the number of obligations those interfaces create without a durable owner."

1. The Dependency Review

The programme board had been told that the new customer service would be ready for the November release. The front end was complete, the mobile screens had passed acceptance testing, and the API gateway demonstration had gone well. Yet the dependency review told a different story. One service needed a customer identifier held in a twenty-year-old policy system. A second depended on an overnight batch that could not be moved without changing the finance reconciliation. A third exposed an address in JSON but still relied on an XML message whose meaning differed by product line.

On the plan, these were three interfaces. In practice, they were three negotiations about ownership, meaning and risk.

This is the recurring contradiction at the heart of the API economy. We speak of interfaces as instruments of speed: publish a service, open a channel, connect a partner, let innovation happen at the edge. The ambition is sound. But the integration programme grows more complex precisely because the interface makes more combinations possible. Every new connection is technically smaller than the monolithic change it replaces, yet the whole estate can become harder to reason about.

The important question is therefore not whether APIs are useful. It is why organisations repeatedly mistake easier connection for simpler change.

2. The Promise Beneath the Phrase

By late 2016, the language of the API economy carries a powerful proposition. Capabilities once buried inside applications can be made available to mobile channels, business partners and cloud services through stable contracts. RESTful interfaces and lightweight JSON messages promise an escape from some of the weight of earlier

web-service programmes. Microservices push the argument further: if a business capability can be separated into a small, independently deployable service, teams need not wait for a large application release to move.

There is genuine progress here. A well-designed API can prevent five channels from building five different routes into the same policy system. A service boundary can contain change. Clear versioning can allow consumers to move at different speeds. None of this should be dismissed as fashion.

But the phrase API economy encourages an economic metaphor that is only half-complete. It emphasises exchange while hiding production. An API may look like a neat product on a catalogue, but behind it sit data quality, operational support, security decisions, release discipline and a system of record that may have been designed long before external consumption was imagined.

A market works because its participants can rely on rules, definitions and remedies. An API estate needs the equivalent. Without them, the organisation has not created an economy. It has created a bazaar of promises.

The interface is the visible edge of a much larger operating agreement. When that agreement is absent, programme complexity does not disappear; it migrates into exceptions, meetings and release delays.

3. Architecture Does Not Abolish Coordination

The earliest diagrams of an API programme often look reassuring. Boxes become smaller, arrows become cleaner, and the enterprise service bus is no longer drawn as the centre of everything. The picture suggests that decomposition has reduced the problem.

What it has actually done is redistribute it.

A large application coordinates many decisions internally. Once it is divided into services, those decisions must be coordinated through contracts, ownership and runtime behaviour. The dependencies remain, but they become less visible to any one team. This is why microservices can reduce the size of a code change while increasing the number of parties required to make it safe.

The mechanism is straightforward:

- Semantic dependency: two services can agree on message syntax while disagreeing on what “active customer”, “available balance” or “effective date” means.
- Release dependency: independent deployment is promised, but a breaking contract change still forces several consumers into the same release window.
- Operational dependency: a transaction crosses four services, yet monitoring is organised application by application. When it fails, each component appears

healthy.

- Control dependency: identity, consent, audit and retention decisions are repeated at every boundary unless common rules are established.
- Commercial dependency: a partner-facing API creates service expectations that the internal system owner was never funded to meet.

The old integration programme concentrated complexity in a central team and a visible queue. The newer model distributes complexity across product teams and hides part of the queue in their backlogs. Distribution can be healthier, but only if authority and capability travel with it.

Earlier promise	What the programme discovers
Smaller services mean smaller changes	The code change is smaller; the coordination path may be longer
Reusable APIs prevent duplication	Reuse increases the number and diversity of consumers
Independent deployment removes release trains	Contract discipline becomes a prerequisite, not an option
A gateway creates control	A gateway controls traffic; it does not resolve meaning or ownership
A service catalogue creates discoverability	Discovery without trustworthy support information accelerates poor reuse

4. The Debt That Does Not Sit in Code

Technical debt is commonly described as a compromise in software that makes later change more expensive. Integration debt is broader. It is the accumulated cost of connections whose ownership, semantics or operational obligations were never made explicit.

Some of it is visible in code: point-to-point mappings, duplicate transformations, brittle adapters. The more consequential portion often sits elsewhere:

- a customer status translated differently by two programmes;
- an API owner accountable for availability but unable to change the source system;
- a test environment whose reference data bears little resemblance to production;
- a consumer that copied a response into its own database and now treats yesterday's value as authoritative;
- an interface described as reusable but funded only until the originating project closes.

This debt compounds because an interface acquires consumers. The first shortcut may save a fortnight. The fourth consumer turns that shortcut into a constraint, and the seventh turns it into a governance dispute.

Consider a composite programme built around a new intermediary channel. The initial scope covered 23 business capabilities across six existing applications. The architecture catalogue listed 41 APIs, of which 17 were described as reusable. By the second release, the programme had created 64 services and 138 production interfaces. Only nine services had a named business owner; the rest belonged, by default, to whichever delivery team had first built them.

The programme did not fail dramatically. It slowed by degrees. A change to the definition of an eligible account required eleven consumer impact assessments. End-to-end testing occupied seven weeks of a twelve-week release cycle. Two teams built separate customer-search services because the first service's support hours did not meet the second channel's needs. The integration architect became the practical bottleneck, not because the architecture was unusually poor, but because one person was carrying decisions the operating model had never allocated.

The turning point was not a new tool. It was a decision to classify each service by business capability, name an accountable owner, publish a support promise, and refuse a new consumer until its data meaning and failure behaviour were agreed. The next release contained fewer new APIs—twelve rather than twenty-seven—but end-to-end testing fell from seven weeks to four. The apparent reduction in output produced an increase in usable change.

Integration debt is therefore not the number of interfaces in an estate. It is the number of obligations those interfaces create without a durable owner.

5. The Serious Case for Moving Quickly

The strongest argument against caution deserves respect. Markets are being reshaped by mobile expectations, specialist digital entrants and partners that will not wait for a twelve-month integration cycle. A programme that insists on perfect enterprise definitions before exposing any capability will protect coherence by making itself irrelevant. Central integration functions have often earned their reputation for delay: long design queues, heavyweight standards and a preference for theoretical reuse over a working customer journey.

Microservices and API-first delivery offer a credible correction. Give a small team control of a bounded capability, make the contract explicit, automate its build and test process, and let consumers integrate against it. Local autonomy can produce learning faster than a committee can produce certainty. In a fast-moving service, that learning has real economic value.

The mistake is not speed. It is treating speed as proof that the boundary is sound.

A team can deploy independently only when it owns enough of the service's data, runtime and decisions to act independently. If every meaningful change still requires approval from a mainframe owner, an information-security function, a central database team and three consuming channels, the microservice boundary is architectural theatre. The organisation has drawn autonomy on a diagram without creating it in practice.

“A small service is not autonomous because its codebase is small; it is autonomous when the decisions required to change it are contained.”

The answer is not to rebuild central control under a newer name. It is to distinguish guardrails from gates. Authentication patterns, versioning rules, logging requirements and minimum support information can be established once as guardrails. Business meaning, service levels and investment choices must remain owned by the capability that creates the obligation. Central teams should make safe delivery easier, not become the place where every decision waits.

6. Small Services Can Create a Large Programme

Programme leaders often count APIs as deliverables because they are visible and easy to report. The measure is seductive: 30 designed, 22 built, 18 live. Yet it says almost nothing about whether the organisation can change safely.

A more revealing view asks what happens when a contract changes, a source system is unavailable, or a new consumer arrives. If the answer depends on personal knowledge and a sequence of emergency meetings, the estate is not modular. It is merely fragmented.

Four tests expose the difference:

1. Can the owner decide? The named service owner must be able to resolve priorities, semantics and service levels, not simply coordinate people who hold those powers elsewhere.
2. Can the consumer move safely? Contracts need explicit versioning, representative test data and a defined period of coexistence. “Non-breaking” must be demonstrated from the consumer’s perspective.
3. Can operations see the journey? Logs and measures must allow a failed business transaction to be traced across service boundaries, not only show that each server is running.
4. Can the service survive its project? Funding, support and documentation must persist after the delivery team disperses. A reusable API without a lifecycle is an unfunded liability.

These tests change the programme conversation. Instead of asking whether the gateway is installed, we ask whether a consumer can discover a service and trust its terms. Instead of asking how many microservices have been deployed, we ask how many can be changed without coordinated release. Instead of celebrating reuse, we examine whether the reused service has the capacity and support model its additional consumers require.

7. A Different Measure of Progress

The API economy does not remove the need for integration discipline. It changes its object. The discipline is no longer primarily the movement of messages between applications; it is the management of commitments between capabilities.

That requires programme measures closer to the mechanisms of change:

- proportion of services with an accountable business and technical owner;
- elapsed time from a proposed contract change to safe consumer adoption;
- number of consumers per version, including those no longer actively maintained;
- percentage of critical transactions traceable across every service hop;
- reuse accompanied by an agreed service level and funded capacity;
- defects caused by semantic disagreement rather than transport failure.

These measures may initially make progress look worse. That is useful. A catalogue of sixty APIs can appear impressive until it reveals twenty-three competing customer definitions and fourteen services with no support owner. Visibility is not a setback; it is the beginning of control.

The deeper lesson is that integration is not a temporary workstream between otherwise stable systems. In an organisation opening more capabilities to more channels and partners, integration becomes a permanent feature of the operating model. The programme can establish the first contracts and patterns, but it cannot own every obligation indefinitely.

8. The Boundary We Are Really Designing

The debate is often framed as old integration against new architecture: enterprise service bus against lightweight API, monolith against microservice, central standards against team autonomy. These contrasts are useful until they conceal the more important question.

We are designing boundaries of responsibility.

A good boundary places a business decision, the information needed to make it, and the authority to act close enough together that change can occur without institutional theatre. A poor boundary exposes a clean REST interface while leaving those elements scattered across departments, suppliers and release calendars.

The API economy will reward organisations that make capabilities easier to combine. But combination without responsibility produces a larger field of hidden dependencies. The winners will not be those with the greatest number of APIs, nor those that divide their applications into the smallest services. They will be those that understand every interface as a promise—about meaning, availability, change and ownership—and build the programme around keeping it.

That is the gap between transformation intent and transformation reality. The intent is to open the enterprise through technology. The reality is that openness works only when the organisation is equally explicit about who carries the consequences.